Coalgebraic effects and their cohandlers in programming languages

(Efekty koalgebraiczne oraz ich kohandlery
w językach programowania)

Mateusz Urbańczyk

Praca inżynierska

Promotor: dr Maciej Piróg

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

1 września 2020

Abstract

. . .

. . .

Contents

Chapter 1

Introduction

My algebraic methods are really methods of working and thinking; this is why they have crept in everywhere anonymously. ∼ Emmy Noether

TODO This requires a rework

In this thesis we present experimental programming language Freak, which is an implementation of Continuation Passing Style for Effect Handlers paper [10], with the addition of a few basic constructs.

1.1 Problem Analysis

TODO add citations

Algebraic effects, while not being a new concept, in the recent years have received a lot of attention, both from theoretical and practical side.

TODO refactor the following

They give developer necessary tools for declarative programming, writing programs in a way to strive from exposing implementation details, while still preserving a lot of flexibility and having strong theoretical background and ways of formal reasoning.

In fact, that very flexibility can be troublesome. Allowing developer for too much freedome may lead to undesired behaviour and incorrect programs, for that very reason, we develop type systems, to detect errors before they occur.

1.2 Problem Statement

TODO Think about merging Problem Analysis with Statement

Multi-shot resumptions in effect handlers are a powerful tool to describe fairly complex logic in a concise way. That being said, they give rise to issues that would not occur in standard control flow, especially when composing various effects together.

We will explain this in greater details and think about possible ways to address the issue of flexibility of effects in the next sections.

TODO actually point out which sections you talk about

1.3 Thesis Outline

TODO Guide reader through thesis, probably later than sooner
TODO Clearly divide categorical from operational approach for less algebraically fluent readers

TODO it's outdated We start by presenting the related work, then discuss syntax and operational semantics. Basic usage guide for playing with the language is provided, as well as implementation details and examples. We conclude by stating what are the possible augmentations, that are intended to be made in the future.

Chapter 2

Background

2.1 Computational Effects

Since the rapid development of computational theory in 1930s by A. Turing, K. Godel and A. Church, we have a well-established notion of what can and what cannot be done through algorithmic means, which we can almost directly translate to being computable by our machines. Through next years we have developed mainstream languages that are used almost everywhere, with a great success.

Under these circumstances one may pose a question, why do we still bother with development of languages theory, since so much has been done already. Is there anything that drives us towards further research? Indeed, one active branch revolves around equational theory to asses equality of two programs, which we know that in general setting is undecidable. Proof methods may include extensional, contextual or logical equality. However, there is no doubt that these formal ways of reasoning about programs, while being crucial for assessing correctness, do not bring direct benefits for everyday use cases, as they are rarely accessible by a common developer.

Other branch of languages theory, that we shall investigate more in this thesis, is about taming complexity of programs. Various methods of static analysis has been developed for various use cases, most notably, type systems.

What is the source of complexity? Thanks to strong and static type systems along with their implementations, we have solid tools to work efficiently on functions that are pure. That being said, we claim that the core complexity of programs comes from side effects, which we cannot avoid in writing anything useful.

We need to have a good way for handling computational effects. One of the ways to model them, can be done through monads [14]. However, they were found to be, to say the least, cumbersome to work with when the number of different effects increases. It is perhaps not a coincidence that many functional programming languages do not have them, because of the non-composable nature of them, or at least not composable in their implementations.

Let's put the following functions

$$\begin{gathered} f : a \to b,\ g : b \to c \\ f' : a \to m\ b,\ g' : b \to m\ c \\ f'' : a \to m'\ m\ b,\ g'' : b \to m'\ m\ c \end{gathered}$$

where m, m' are monads. Functions f, g can be composed using standard composition, for f', g' we can use Kleisli composition operator from monad m. What in case of adding one more effect?

It turns out, that it becomes complex and unpleasent to combine two functors together to form a new monad, for which purpose, monad transformers arose in Haskell. Most of the common languages avoid this by not expressing computational effects in the type system, and instead one may think about functions as being implicitly embed in a Kleisli Category over a functor T, where T is a hidden signature over all possible side effects that occur in our program.

TODO Think about describing what is Kleisli Category

Not only we would like to bring back effects to our type system, but also do it in a way that is composable. This is where algebraic effects comes to the rescue. From theoretical point of view, we need to develop equational theory about our effects to assert correctness of our langauge as well as to have the right hammer to reason about our programs. That is the point where we would like to introduce to unfamiliar reader a notion of algebraic effects.

2.2 Algebraic Effects

Algebraic effects can be thought as an public interface for computational effects. Declarative approach allow us to write programs in which the actual semantic of source code is dependent on handler that defines the meaning of a subset of effects.

This is really a breakthrough from practical point of view, as we may substitute logic depending on execution environment. As an example, fetching for resources can behave differently as we run tests, debug our code, or run it on production. In the same manner, they neatly allow us to abstract over implementation details.

Patterns like these are well known in programming, for which other alternatives arose. One can mention interfaces from object-oriented programming, which are also a way to describe the API of a certain component. In order to abstract from implementation details, we pass an object around which represents a certain interface. While this is certainly better than having no abstractions at all, we need to pass this interface into every function that is going to use it. This practice is called dependency injection, and while it sounds like it solves some of the issues, we end up in functions that need to carry representants of the interfaces, and pass them in every subcall that they make. It also can't be ommited that it's not really a substitute, as it's solving only one particular issue that algebraic effects address.

TODO add link to paper with plenty of examples

- Exceptions - Async/await - Flipping coin - Backtracking? - Operating on file

2.3 Categorical Setting of Universal Algebra

Algebraic effects can be described via operational means, however, for the purpose of presenting the duality between algebra and coalgebra, we allow ourselves to wander a bit deeper into category theory and describe effects from denotational point of view.

2.3.1 Algebraic Theories

Definition 2.1. A *signature* Σ is given by a collection of operation symbols op_i with associated parameters P_i and arities A_i, where P_i and A_i are objects in the category of our interest. We will write an operation as $op_i : P_i \rightsquigarrow A_i$

Definition 2.2. Collection of *Σ-terms* is a free algebra with a generator X for a functor μH_Σ that maps objects into trees over a given signature Σ and morphisms into folds over trees.

Definition 2.3. A *Σ-Equation* is an object X and a pair of *Σ-terms* $l, r \in Tree_\Sigma(X)$, written as

$$X \mid l = r$$

Definition 2.4. An *algebraic theory* $T = (\Sigma_T, \mathcal{E}_T)$, is given by a signature Σ_T and a collection $\mathcal{E}_T$ of *Σ_T-equations*. For clarity, we will usually omit T subscript.

Definition 2.5. An *interpretation I over a given signature* Σ is given by a carrier object $|I|$ and for each $op_i : P_i \rightsquigarrow A_i$ in Σ a map

$$[\![op_i]\!]_I : P_i \times |I|^{A_i} \to |I|$$

Interpretation may be naturally extended to *Σ-terms*, such that a given *Σ-term* $X \mid t$ is interpreted by a map which sends variables into projections from environment and terms into map composition over each subterm.

Definition 2.6. A *model M* of an algebraic theory T is an interpretation of the signature Σ_T which validates all the equations $\mathcal{E}_\mathcal{T}$. That is, for every equation $X \mid l = r$ the following diagram commutes:

TODO this diagram doesn't work

Definition 2.7. Free F-algebra on an object A (of generators) in $\mathcal{C}$ is meant an algebra

$$\varphi_A : FA^{\#} \longrightarrow A$$

together with an universal arrow $\eta_A : A \longrightarrow A^{\#}$. Universality means that for every algebra $\beta : FB \longrightarrow B$ and every morphism $f : A \longrightarrow B$ in $\mathcal{C}$, there exists a unique homomorphism $\overline{f} : A^{\#} \longrightarrow B$ extending f, i.e. a unique morphism of $\mathcal{C}$ for which the diagram below commutes:

TODO we can also say it's an initial algebra over initial object.

Definition 2.8. *Free model* is just a model which is free algebra.

Lemma 2.9. *Free models form monads*

Definition 2.10. Let L, M be models of a theory T. A *T-homomorphism* $\phi : L \to M$ is a map such that the following diagram commutes:

TODO finish definition

2.4 Duality

2.4.1 Comodels

Definition 2.11. Comodel in $\mathcal{C}$ is just a Model in $\mathcal{C}^{op}$.

We could end by stating this definition, and everything else would follow directly from duality. However, expanding definitions is going to give us better intuition, as well as give more solid ground for implementation.

2.4.2 Cooperations

Derive from Models duality

2.4.3 Coalgebraic Effects

2.4.4 Coinductive Reasoning

Induction and Coinduction

Induction is a way of constructing new structures. Recursion is a way of folding inductively defined structure in an terminating way. Recursive functions should shrink the argument in each call, meaning that it eventually ends up terminating in a base case.

Coinduction is literally a dual notion to induction. We *observe* possibly infinite structures, by doing deconstruction. Corecursion, however, is a way of productively defining new, possibly enlarged structures. Due to infinity, the evaluation should be lazy, whereas in induction it may be eager.

Here is a table that summarizes difference between these two methods of reasoning:

feature	**induction**	**coinduction**
basic activity	construction	deconstruction
derived activity	deconstruction	construction
functions shape	inductive domain	coinductive codomain
(co) recursive calls	shrinks the argument	grows the result
functions feature	terminating	productive
evaluation	possibly eager	necessarily lazy

Reader that focuses on pragmatism may pose a question, why do we even want to reason about infinite structures? They never appear in practice! In fact, it's very common to operate on never-ending transition systems or streams of data without, wher efinitary means of reasoning are of no use, as we can't expect an end to stream!

Simple example of the duality can be expressed through induction over finite list and coinductive observation of infinite streams.

Doing the latter may involve modification of the internal state of the machine that is generating the infinite streams, or in more concrete scenario, alternation of the external resource that is providing us the data.

TODO Write a coinductive proof for divergent terms using coinductive proof principle.

2.4.5 Coinduction Coalgebra something section

TODO Rethink this section, back by examples

This is one of the cases where interaction with external resource multiple times, or more specifically in case of algebraic effects, invoking resumption more than once, may lead to unexpected behaviour that would not be expected in standard control flow. This captures the excessive generality of effects, and is the issue that we would like to address with coalgebraic effects.

TODO On it's own this snippet a bit dry and of course requires additional examples, for instance the classic one with operating on closed file and/or on streams of data

2.5 Related Work

2.5.1 Algebraic Effects

In fact, libraries for algebraic effects also arose in mainstream languages, such as C or Python. In Python, effects are implemented through generators, using built-in feature of sending value when doing *yield* operation. Resumptions in handlers that are sending value, are one-shot and tail-recursive, therefore we do not need to handle coalgebraic part, at the cost of flexibility.

TODO add reference to C and Python lib

Except from Links language [10], on which the implementation is based, there are currently many other alternatives available. One may take a look at Frank [13], which provides a support for multihandlers, Koka [12], Helium [6] or Eff [3]. Except from separate languages, many libraries arose for existing ones like Haskell, Idris, Scala or Multicore OCaml.

As can be seen in the J. Yallop repository [7], algebraic effects and handlers are now trending branch in the programming languages theory.

2.5.2 Coalgebraic Effects

Ahman and Bauer [1]

Chapter 3

Potential Solutions

TODO smth smt initialization before finalization.

It can be seen, that issues lies with the fact, that interacting with coalgebras may lead to change of their internal state system. Examples include change of state in NFA, taking next element in infinite stream or a closing file.

One of the ways to approach this problem, caused by extensive generality of effects, is extension of the calculus with Coalgebraic effects, also named coeffects. Difference is that resumptions in coeffects handlers are one-shot only, which means that continuation can be invoked at most once.

Lemma 3.1. *Problem of detection whether resumption is called only once is undecidable.*

Proof. Follows directly from Rice's Theorem, as checking whether function was called is a nontrivial semantic property of a program. □

3.1 Dynamic Constraints Checking

One approach is to dynamically during execution check against contract that continuation may only be called once. In this setting we are sure that our program will not accidentaly go into wrong state, however by definition we lack static analysis to prevent from errors before they occur.

3.2 Linear Types

Solve the issue through introduction of linear type system.

TODO Explain it can get complex and unfamiliar for users.

3.3 Data-Flow Analysis

Another static analysis of the program.

TODO Describe what is data-flow analysis

TODO Give example for constant folding

TODO Propose analysis on imaginary CFG Not sure if it makes sense?

3.4 Cohandlers as Separate Constructs

Simplify semantics by separating coalgebraic effects into a new, restricted construct in programming language

Chapter 4

(Co) Effectful Programming

4.1 Examples

In this section we present a few examples to show the capabilities of the language. The ideas have been based on [4], and thus will not be described in great details. More exemplary programs in Freak language can be found under https://github.com/Tomatosoup97/freak/tree/master/src/programs.

4.1.1 Choice

The first example will be based on modelling (nondeterministic) choice in the program. We will make two decisions, which will affect the computation

result:

```
let c1 <- do Choice () in
let c2 <- do Choice () in
let x <- if c1 then return 10 else return 20 in
let y <- if c2 then return 0 else return 5 in
    return x - y
```

With that in hand, we may want to define effect handlers:

```
handle ... with {
    Choice p r ->
        let t <- r 1 in
        let f <- r 0 in
        <PLACEHOLDER> |
    return x -> return x
}
```

where in the `<PLACEHOLDER>` we can define on what to do with the computation. For example, min-max strategy for picking the minimum value:

```
if t < f then return t else return f
```

where the code evaluates to `5`. Another example is a handler that collects all possible results, which can be achieved by putting `return (t, f)` in the `<PLACEHOLDER>`, which evaluates to `((10, 5), (20, 15))`.

4.1.2 Exceptions

Exceptions are simply algebraic effect handlers which drop the resumption.

```
handle
    if x == 0 then do ZeroDivisionError ()
              else return 1/x
with {
    ZeroDivisionError p r -> return 42 |
    return x -> return x
}
```

Where we imagine that x variable has been bound previously.

4.1.3 Taming Side effects

The complexity of the programs and their performance usually comes from side effects. Algebraic effects allow us to define code in a declarative manner, and hence neatly tame the side effects that they produce. This gives us a lot of flexibility in the actual meaning without duplicating the code. Let's consider the following very basic code snippet:

```
let x <- do Fetch () in
-- operate on x
```

The code is dependent on a context in which it is executed, which here is the handler that defines the behaviour of the algebraic `Fetch` effect. In the imperative, or even functional approach, we would need to provide the interface for fetching the data by doing dependency injection or even embedding the operation directly. Here we are just stating what operation we are performing, leaving the interpretation up to the execution context, which could do the fetching or mock the external resource.

TODO subset of this section's content is explained already

These implications are straightforward when looking from a categorical standpoint, where effects are viewed as free models of algebraic theories [16], and handlers are homomorphisms preserving the model structure [15]. Nevertheless, the results are very exciting for programming use cases. The current Freak implementation does not support I/O.

4.2 Coexamples

Examples for cohandlers

4.3 Usage guide

As of this day, two implementations are available, one based on the curried translation and Appel [2], and the second one based directly on the uncurried translation with continuations as explicit stacks from paper. More details can be found in Section 6. All commands are available within `src` directory.

4.3.1 Build and install

- Install dependencies: `make install`
- Select implementation: `make link-lists` (default) vs `make link-appel`
- Compile: `make build`
- Link to PATH: `sudo make link`
- Remove artifacts: `make clean`

After compiling and linking program to PATH, one may evaluate program as follows: `freak programs/choicesList.fk`. The actual code is described in Section 4.1.1

4.3.2 Running tests

Test cases are available here, they include both inline and file-based tests. For more details about writing tests, one may refer to *HUnit documentation* [11].

- Run tests: `make tests`
- Run code linter: `make lint`
- Compile, run linter and tests: `make check`

Chapter 5

Calculus of Freak language

5.1 Syntax

TODO Extend language with basic constructs for usability

The syntax for the calculus is shown below. *nat n* represents an integer n, $V \oplus W$ and $V \approx W$ are respectively binary and relational operators, where we support basic arithmetic and comparison operations. **if** V **then** M **else** N is a standard branching statement. The other constructs are just as in Links, with slight syntax modifications. Actual programs in Freak can be found in Section 4.1.

$$
\begin{aligned}
&\langle \textit{Values } V,\ W \rangle ::= x \mid \textit{nat}\ n \\
&\quad \mid \backslash x : A \to M \mid \textbf{rec}\ g\ x \to M \\
&\quad \mid V \oplus W \mid V \approx W \\
&\quad \mid \langle\rangle \mid \{\ell = V; W\} \mid [\ell\ V]^R
\end{aligned}
$$

$$
\begin{aligned}
&\langle \textit{Computations } M,\ N \rangle ::= V\ W \\
&\quad \mid \textbf{if}\ V\ \textbf{then}\ M\ \textbf{else}\ N \\
&\quad \mid \textbf{let}\ \{\ell = x; y\} = V\ \textbf{in}\ N \\
&\quad \mid \textbf{case}\ V\{\ell\ x \to M; y \to N\} \mid \textbf{absurd}\ V \\
&\quad \mid \textbf{return}\ V \mid \textbf{let}\ x \leftarrow M\ \textbf{in}\ N \\
&\quad \mid \textbf{do}\ \ell\ V \mid \textbf{handle}\ M\ \textbf{with}\ \{H\}
\end{aligned}
$$

$$\langle \textit{Handlers } H \rangle ::= \textbf{return}\ x \to M \mid \ell\ p\ r \to M, H$$

$$\langle \textit{Binary operators } \oplus \rangle ::= + \mid - \mid * \mid /$$

$$\langle \textit{Relational operators } \approx \rangle ::= < \mid \leqslant \mid > \mid \geqslant \mid == \mid != $$

5.2 Typing Rules

5.3 Operational Semantics

The source language's dynamics have been described extensively by providing small-step operational semantics, continuation passing style transformation [10] as well as abstract machine [8], which was proved to coincide with CPS translation. That being said, Freak introduces new basic constructs to the language, for which we shall define the semantics.

TODO Rewrite here operational semantics from Links paper

Extension of the evaluation contexts:

$$\mathcal{E} ::= \mathcal{E} \oplus W \mid \mathit{nat}\ n \oplus \mathcal{E} \mid \textbf{if}\ \mathcal{E}\ \textbf{then}\ M\ \textbf{else}\ N$$

Small-step operational semantics:

$$\begin{array}{ll} \textbf{if}\ \mathit{nat}\ n\ \textbf{then}\ M\ \textbf{else}\ N \rightsquigarrow M & \text{if } n \neq 0 \\ \textbf{if}\ \mathit{nat}\ n\ \textbf{then}\ M\ \textbf{else}\ N \rightsquigarrow N & \text{if } n = 0 \end{array}$$

$$\begin{array}{ll} \mathit{nat}\ n \oplus \mathit{nat}\ n' \rightsquigarrow n'' & \text{if } n'' = n \oplus n' \\ \mathit{nat}\ n \approx \mathit{nat}\ n' \rightsquigarrow 1 & \text{if } n \approx n' \\ \mathit{nat}\ n \approx \mathit{nat}\ n' \rightsquigarrow 0 & \text{if } n \not\approx n' \end{array}$$

5.4 Continuation Passing Style Transformation

...

Chapter 6

Implementation

The Freak implementation is available at https://github.com/Tomatosoup97/freak, written purely in Haskell. While the paper provided a good overview of the language and the translation, the lower-level details were omitted. That being said, two inherently different takes at the implementations were made. The first one is based on curried translation and A. Appel [2] book, and the second one directly on the uncurried translation to target calculus with continuations represented as explicit stacks from the paper. We start by presenting core data structures, and afterwards move to actual translation details.

6.1 Abstract Syntax Tree

The language's AST is defined without surprises, just as syntax is:

```
data Value
    = VVar Var
    | VNum Integer
    | VLambda Var ValueType Comp
    | VFix Var Var Comp
    | VUnit
    | VPair Value Value
    | VRecordRow (RecordRow Value)
    | VExtendRow Label Value Value
    | VVariantRow (VariantRow Value)
    | VBinOp BinaryOp Value Value

data Comp
    = EVal Value
    | ELet Var Comp Comp
    | EApp Value Value
    | ESplit Label Var Var Value Comp
    | ECase Value Label Var Comp Var Comp
    | EReturn Value
    | EAbsurd Value
    | EIf Value Comp Comp
    | EDo Label Value
    | EHandle Comp Handler
```

Similarly for the target calculus data structure. However, as one may notice, for convenience the **let** translation is homomorphic, as opposed to be to lambda abstracted with immediate application:

```
data UValue
    = UVar Var
    | UNum Integer
    | UBool Bool
    | ULambda Var UComp
    | UUnit
    | UPair UValue UValue
    | ULabel Label
    | URec Var Var UComp
    | UBinOp BinaryOp UValue UValue

data UComp
    = UVal UValue
    | UApp UComp UComp
    | USplit Label Var Var UValue UComp
    | UCase UValue Label UComp Var UComp
    | UIf UValue UComp UComp
    | ULet Var UComp UComp
    | UAbsurd UValue
```

The final answer, common to both evaluations, is represented as a `DValue`, where the meaning of the coproduct is as one would expect:

```
type Label = String
type FuncRecord = [DValue] -> Either Error DValue
data DValue
    = DNum Integer
    | DLambda FuncRecord
    | DUnit
    | DPair DValue DValue
    | DLabel Label
```

6.2 Target calculus

6.3 Curried translation

The first take was heavily inspired by A. Appel's Compiling with Continuations [2], which provides a translation for a simplified ML calculus. The calculus was extended and translation adapted to handle algebraic effects and their handlers. The translation is based on the curried first-order translation. That being said, the source code diverged a lot from the paper on which it was based, leading to a different transformation for which the correctness and cohesion with operational semantics should be proved separately. Indeed, while the interpreter worked well on the use cases defined in tests, the evaluation had a part which was not tail-recursive. What's more, nested handlers were not supported, and the implementation was found to be trickier than it should, as it was not obvious on how to adopt the technique proposed in the paper.

In terms of improving the performance of the evaluation, uncurried higher-order translation should be adapted, so that administrative redexes are contracted and proper tail-recursion is obtained. The core data structure, into which the source program is transformed, is defined as follows:

```
data ContComp
    = CPSApp CValue [CValue]
    | CPSResume CValue ContComp
    | CPSFix Var [Var] ContComp ContComp
    | CPSBinOp BinaryOp CValue CValue Var ContComp
    | CPSValue CValue
    | CPSLet Var CValue ContComp
    | CPSSplit Label Var Var CValue ContComp
    | CPSCase CValue Label Var ContComp Var ContComp
    | CPSIf CValue ContComp ContComp
    | CPSAbsurd CValue
```

Most of the terms at the end have a coinductive reference to itself, which represents the rest of the computation that needs to be done. For more clarification, one may take a look into the book mentioned above [2]. The source code for curried translation and evaluation can be found respectively in `CPSAppel.hs` and `EvalCPS.hs`.

6.4 Uncurried translation

Having in mind the drawbacks mentioned above, alternative translation was written, that coincides with the translation from the paper. Namely, with the uncurried translation to target calculus with continuations represented as explicit stacks. The target calculus was described in Section 6.1, for which the evaluation is straightforward. The continuations are represented as `Cont`, with syntactic distinction between pure and effectful computations, which occupy alternating positions in the stack. Explicit distinction gave more control in the source code.

```
type CPSMonad a = ExceptT Error (State Int) a

type ContF = UValue -> [Cont] -> CPSMonad UComp

data Cont = Pure ContF
          | Eff ContF
```

Where `CPSMonad` is a monad transformer over `Either` and `State`. `State` was required to generate labels for fresh variables that came from the translation. The core code is split into five functions:

```
cps     :: Comp    -> [Cont] -> CPSMonad UComp
cpsVal  :: Value   -> [Cont] -> CPSMonad UValue
cpsHRet :: Handler -> Cont
cpsHOps :: Handler -> Cont
forward :: Label   -> UValue -> UValue -> [Cont] -> CPSMonad UComp
```

Where the first two are implementing cps for computations and values. `cpsHRet` and `cpsHOps` are yielding pure and effectful continuations, based on a given handler. The last one is responsible for forwarding the computation to the outer handler.

This results in an implementation that finally supports nested handlers, as can be seen by evaluating `programs/complexNestedHandlers.fk` program. Unfortunately, following closely translation from the paper resulted in a behaviour, in which invoked resumption forgets its pure continuation. This means, that the following code, evaluating correctly on the Appel-based translation, returns 0 rather than 1:

```
handle do Drop () with { Drop p r -> let t <- r 0 in return 1}
```

Nevertheless, working out this issue appears as less demanding than coping with discrepancies created in the first translation. The source code for uncurried translation and evaluation can be found respectively in `CPSLists.hs` and `EvalTarget.hs`.

TODO Drop of continuations is still an issue

6.5 Cohandlers

...

TODO Simple implementation for cohandlers

6.6 Source Code Structure

The source code is divided into a number of modules, where the most crucial parts have already been described.

```
AST.hs           - AST data structures
CommonCPS.hs     - Common functions for CPS translation
CommonEval.hs    - Common functions for evaluation
CPSAppel.hs      - Appel-based CPS translation
CPSLists.hs      - Uncurried CPS translation
EvalCPS.hs       - Evaluation of the Appel's CPS structure
EvalTarget.hs    - Evaluation of the target calculus
Freak.hs         - API for the language
Main.hs          - Main module running evaluator on given filename
Parser.hs        - Parser and lexer
TargetAST.hs     - AST for the target calculus
Tests.hs         - Tests module
Types.hs         - Common types definition
programs/        - Exemplary programs used in tests
```

Chapter 7

Conclusion

7.1 Summary

What was achieved, what was described, what needs to be done or researched.

7.2 Future work

7.2.1 Abstract machine

The Links language also provides small-step operational semantics and an abstract machine [8]. Implementing another way of evaluation could serve as a way to empirically assert correctness, as opposed to formally.

7.2.2 Type inference and row polymorphism

The type system as of this day is not implemented, as the focus has been put on CPS transformation. Further work is required here, especially considering the fact that a huge advantage of algebraic effects is that they are explicitly defined in the type of a computation.

7.2.3 Multiple instances of algebraic effects

The Freak language is limited to a single instance of an effect. We would need to support cases where many instances of the algebraic effects, with the same handler code, could be instantiated. The current state of the art introduces a concept of resources and instances, as in Eff [4], or instance variables, as in Helium [5].

7.2.4 Selective CPS

Other languages, like Koka [12], or even the core of the Links, are performing selective CPS translation, which reduces the overhead on code that does not perform algebraic effects. Our current translation is fully embedded in the CPS.

7.2.5 Exceptions as separate constructs

Exceptions are a trivial example of algebraic effect where the resumption is discarded, and as described in §4.5 [10], they can be modeled as a separate construct to improve performance.

7.2.6 Shallow handlers

Shallow and deep handlers while being able to simulate each other up to administrative reductions, have a very different meaning from a theoretical point of view. Implementing them as defined by Lindley et al. [9] could be another way of enhancing Freak.

Bibliography

[1] Danel Ahman and Andrej Bauer. „Runners in Action". In: *Lecture Notes in Computer Science* (2020), 29–55. ISSN: 1611-3349. DOI: 10.1007/978-3-030-44914-8_2. URL: http://dx.doi.org/10.1007/978-3-030-44914-8_2.

[2] Andrew W. Appel. „Compiling with Continuations". In: (1992).

[3] Andrej Bauer and Matija Pretnar. „An Effect System for Algebraic Effects and Handlers". In: *Logical Methods in Computer Science* 10.4 (2014). DOI: 10.2168/LMCS-10(4:9)2014. URL: https://doi.org/10.2168/LMCS-10(4:9)2014.

[4] Andrej Bauer and Matija Pretnar. „Programming with Algebraic Effects and Handlers". In: *CoRR* abs/1203.1539 (2012). arXiv: 1203.1539. URL: http://arxiv.org/abs/1203.1539.

[5] Dariusz Biernacki et al. „Binders by Day, Labels by Night: Effect Instances via Lexically Scoped Handlers". In: *Proc. ACM Program. Lang.* POPL (2020).

[6] Dariusz Biernacki et al. „Handle with Care: Relational Interpretation of Algebraic Effects and Handlers". In: *Proc. ACM Program. Lang.* 2.POPL (Dec. 2017), 8:1–8:30. ISSN: 2475-1421. DOI: 10.1145/3158096. URL: http://doi.acm.org/10.1145/3158096.

[7] *Collaborative bibliography of work related to the theory and practice of computational effects.* URL: https://github.com/yallop/effects-bibliography.

[8] Daniel Hillerström and Sam Lindley. „Liberating Effects with Rows and Handlers". In: TyDe 2016. ACM, 2016.

[9] Daniel Hillerström and Sam Lindley. „Shallow Effect Handlers". In: *Programming Languages and Systems.* Springer International Publishing, 2018.

[10] Daniel Hillerström et al. „Continuation Passing Style for Effect Handlers". In: *FSCD.* 2017.

[11] *HUnit: A unit testing framework for Haskell.* URL: https://hackage.haskell.org/package/HUnit.

[12] Daan Leijen. „Type Directed Compilation of Row-typed Algebraic Effects". In: POPL. ACM, 2017.

[13] Sam Lindley, Conor McBride, and Craig McLaughlin. „Do be do be do". In: *CoRR* abs/1611.09259 (2016). arXiv: `1611.09259`. URL: `http://arxiv.org/abs/1611.09259`.

[14] Eugenio Moggi. „Notions of Computation and Monads". In: *Inf. Comput.* 93.1 (July 1991), 55–92. ISSN: 0890-5401. DOI: `10.1016/0890-5401(91)90052-4`. URL: `https://doi.org/10.1016/0890-5401(91)90052-4`.

[15] Gordon Plotkin and Matija Pretnar. „Handlers of Algebraic Effects". In: *Programming Languages and Systems*. Ed. by Giuseppe Castagna. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 80–94. ISBN: 978-3-642-00590-9.

[16] Gordon D. Plotkin and John Power. „Adequacy for Algebraic Effects". In: *Proceedings of the 4th International Conference on Foundations of Software Science and Computation Structures*. FoSSaCS '01. Berlin, Heidelberg: Springer-Verlag, 2001, 1–24. ISBN: 3540418644.