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

Algebraic effects are a great way to model computational effects in a composable way. Nonetheless, they give rise to issues with idiomatic resource management. From mathematical side, they form free models over algebraic theories, with handlers being homomorphisms preserving the model structure. Dualizing this notion gives us comodels, which turned out to be suitable for expressing state-passing transitions in the code. We present a calculus with algebraic and coalgebraic effects, to address the issues with external state management and provide abstraction over resources.

Efekty algebraiczne są dobrym rozwiązaniem na modelowanie efektów obliczeniowych, w sposób który dobrze się składa. Niemniej, powstają przy nich problemy z idiomatycznym zarządzaniem zasobami. Z matematycznej strony, tworzą one wolne modele nad teoriami algebraicznymi, a ich handlery homomorfizmy zachowujące strukturę modelu. Dualnym pojęciem są komodele, które okazały się odpowiednim narzędziem do wyrażania w kodzie przejść z przekazywaniem stanu. Prezentujemy rachunek z efektami algebraicznymi i koalgebraicznymi, aby zaadresować problemy z zarządzaniem zewnętrznym stanem.

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*

In this thesis we discuss issues arising from combining algebraic effects and handlers with resource management, and present a solution through coalgebraic means along with presentation of Freak, experimental programming language with (co)algebraic effects and (co)handlers. Implementation is based on Continuation Passing Style translation.

1.1 Problem Statement

Algebraic effects, while not a new concept, have received a lot of attention [10] in recent years, both from the theoretical and practical side.

They give the developer necessary tools for declarative programming with computational effects, on a high level of abstraction, by defining effects as an API in the code. At the same time, algebraic effects preserve a lot of flexibility and have a strong theoretical foundations.

In fact, that very flexibility can be troublesome. Allowing developer for too much freedom may lead to undesired behaviour and incorrect programs.

In particular, multi-shot resumptions in effect handlers and the possibility to drop the resumption allow one to express fairly complex logic in a concise way, but at the same time, they give rise to issues that would not occur in standard control flow, especially when composing various effects together. Unwanted interaction with external resources multiple times or not invoking finalisation code are cases that may occur while using algebraic effects and handlers. Let's consider the following example:

```
handle
    let fh <- do Open "praise.txt" in
    let c <- do Choice () in
    let text <- if c then return "Guy Fieri" else return "is cool" in
    let _ <- do Write (fh, text) in
    do Close fh
with {
    Choose _ k ->
        let t <- r 1 in
        let f <- r 0 in
        return (t, f) |
    return x -> return x
}
```

In the above code we open file, and based on nondeterministic choice, we write to the file, and then close it. This piece of code looks harmless, however, as we invoke resumption for the second time, we are attempting to write to the closed file, and then close it once again. In fact, dropping resumption is also considered harmful, as we would not attempt to close the file.

This captures the excessive generality of effects, and is the issue that we would like to address with coalgebraic effects.

1.2 Thesis Outline

In the following chapter we provide a background about effects, define algebraic effects in the categorical setting, as well as point out dual coalgebraic effects and cohandlers, finishing with showing related work in this area. Chapter 3 describes possible ways to approach the issue of excessive generality, Chapter 4 shows the Freak language through examples along with a usage guide, and then Chapter 5 describes more formally the language's syntax, operational semantics and CPS translation. In the next one, Chapter 6, implementation details are revealed. We conclude in Chapter 7 by stating what the possible augmentations are, which 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 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 assess equality of two programs, which we know that in the 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. 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, or more generally, computational 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 [39, 29]. However, they were found to be, to say the least, cumbersome to work with when the number of different effects increase. 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, f'' and g''?

It turns out, that it becomes complex and unpleasant to combine two functors together to form a new monad, and for this purpose, monad transformers [26] 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 [22], where T is a hidden signature over all possible side effects that occur in our program.

TODO Reconsider reference to ncatlab on Kleisli Cat, often it's not the best resource

Not only we would like to bring back effects to our type system [37], but also do it in a way that is composable. This is where algebraic effects come to the rescue. From theoretical point of view, we need to develop equational theory about our effects to assert correctness of our language as well as to have the right hammer to reason about our programs. From practical side, we need to have the way of taming computational effects in our programs. That is the point where we would like to introduce to unfamiliar readers a notion of algebraic effects.

2.2 Algebraic Effects

Algebraic effects can be thought of 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 an incredible feature from practical point of view, as we may substitute logic depending on the 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.

That being said, it's only one particular issue that algebraic effects address. There are many other great sources for getting familiarized with effects from practical side [34, 7, 23], so we will omit further explanations. More examples can be found in Chapter 4.

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 [5].

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. *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:

$$\begin{tikzcd}
FA^{\#} \arrow[r, "\varphi_A"] \arrow[d, "F\overline{f}"] & A^{\#} \arrow[d, "\overline{f}"] & A \arrow[l, "\eta_A"'] \arrow[dl, "f"] \\
FB \arrow[r, "\beta"'] & B &
\end{tikzcd}$$

Definition 2.3. 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 2.4. 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 2.5. 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*. We might omit T subscripts in case our theory of interest is obvious.

Definition 2.6. 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 2.7. 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:

$$\begin{tikzcd} {|M|^k} \arrow[r, bend left, "{[\![X|l]\!]}"] \arrow[r, bend right, "{[\![X|r]\!]}"'] & {|M|} \end{tikzcd}$$

Definition 2.8. Let L, M be models of a theory T. *T-homomorphism* $\varphi : L \to M$ is a map such that, for every operation symbol op_i in T the following diagram commutes:

$$\begin{tikzcd} {|L|^{ar_i}} \arrow[r, "{[\![op_i]\!]_L}"] \arrow[d, "\varphi^{ar_i}"'] & {|L|} \arrow[d, "\varphi"] \\ {|M|^{ar_i}} \arrow[r, "{[\![op_i]\!]_M}"'] & {|M|} \end{tikzcd}$$

We denote here A^n as *n-ary* product of A.

Definition 2.9. *Free model* is just a model that is free algebra.

Lemma 2.10. *Free models form monads*

Proof. Let M be a free model of an algebraic theory T in category $\mathcal{C}$. We have an endofunctor F_T, which takes objects into free model and maps to unique homomorphisms for which the following diagram commutes:

$$\begin{tikzcd}
X \arrow[r, "\eta_X"] \arrow[d, "f"'] & F_T(X) \arrow[d, "\overline{f}"] \\
Y \arrow[r, "\eta_Y"'] & F_T(Y)
\end{tikzcd}$$

which immediately gives us η natural transformation for a monad. We can now define μ for a monad as a unique morphism for which the diagram commutes:

$$\begin{tikzcd}
F_T(X) \arrow[r, "\eta_{F_T(X)}"] \arrow[dr, "id"'] & F_T(F_T(X)) \arrow[d, "\mu"] \\
 & F_T(Y)
\end{tikzcd}$$

F_T is an endofunctor, therefore it sends X into an object in $\mathcal{C}$. From any object we have a unique map sending an object into a model, which is $\eta_{F_T(X)}$. From free property of our model we get a unique map μ such that the diagram commutes, therefore we have a monad (F_T, η, μ).

$\square$

Definition 2.11. *Handler* is a *T-homomorphism* between free models

$$H : |F_T(X)| \to |F_{T'}(X')|$$

that is, a map between carriers that preserves the structure of theory T.

Concluding from the theory that we have built, algebraic effects are just free models over computation trees, where the signature is the set of effects, and equational theory is defining rules of rewriting expressions with effects. The latter is usually irrelevant from implementation perspective, but may be important for reasoning about programs.

Effect handlers on the other hand, are transformations from one computation tree over given signature of effects into another one, where handler may serve effects and also propagate new ones.

2.4 Duality

2.4.1 Comodels

Definition 2.12. *Comodel* in $\mathcal{C}$ is just a Model in $\mathcal{C}^{op}$.

We could end by stating this definition of comodels [33], 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. For this part, we are going to assume we operate in **Set** category.

2.4.2 Cooperations

We have defined interpretation of an operation as a morphism:

$$[\![op]\!]_M : P \times |I|^A \to |I|$$

following now straight from Yoneda Lemma, we have:

$$[\![op]\!]_M : |I|^A \to |I|^P$$

Let's now take a look on how it dualizes. Since we operate in **Set**, we can represent A^B exponentials as B-*ary* products over A, where for each argument we select one result:

$$[\![op]\!]_M : \prod_{a \in A} |I| \to \prod_{p \in P} |I|$$

We want now to embed this morphism in $\mathbf{Set^{op}}$. Every arrow in opposite category is reverted, thus, products become coproducts and our morphism is reverted:

$$[\![op]\!]_M : \coprod_{p \in P} |I| \to \coprod_{a \in A} |I|$$

It turnes out, that this map can be further simplified.

Lemma 2.13. *In* ***Set*** *category, the following isomorphism holds* $\coprod_{b \in B} A \cong B \times A$

Proof. We can think about B-*ary* coproducts (tagged unions) over the same set such that we have B replicas of A and we select which replica we want to pick. Putting it this way, it immediately follows that it's the same, up to isomorphism, as a cartesian product over indexes from B and corresponding A sets. For a more detailed and generalized results, see [30]. $\square$

Model turns into a comodel [33], which we shall also call *world*. Putting this together with the above lemma, we obtain the following map:

$$[\![op]\!]^W : |I| \times P \to |I| \times A$$

which is called a *cooperation.* Meaning of that morphism, is that based on coalgebra carrier and a parameter, we obtain new state of the coalgebra and a value generated by coalgebra. This state-alike result that we have obtained by dualizing models is indeed surprising, and it's expected that reader may feel astonished after seeing it for the first time. This observation was made in [30].

2.4.3 Coalgebraic Effects

We have now derived a dual concept to algebraic effects and handlers, which in the literature are also called comodels and runners [38, 2]. As we have seen, comodels turned into state-passing (co)operations. In programming languages, they may be a way to model interaction with external resources, where based on current configuration and a parameter, we obtain a new configuration along with it's result.

2.4.4 Coinductive Reasoning

Definition 2.14. *Initial algebra* is an initial object in **Alg**, the category of algebras and homomorphisms.

Lemma 2.15. *Free algebra over an initial object is initial algebra. Dually, cofree coalgebra over a final object is final coalgebra.*

Proof. Follows immediately after putting initial object as generator in the diagram for free algebra. □

Due to close connections presented above of initial algebras and free algebras, from which algebraic effects are arising, we could not spare a few words a about induction and coinduction, where the latter may give us more intuition about coalgebraic effects.

Induction is a way of constructing new structures. *Recursion* is a way of folding inductively defined structure in a terminating way. Recursive functions (not to be confused with recursive computability class), 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* 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

From categorical standpoint, coinduction is formed over a final coalgebra, where corecursion is the mediator between any coalgebra into a final one, and coinductive proof principle corresponds to the uniqueness of the mediator. Recall that coalgebraic effects arises from a particular type of a final coalgebra, namely, cofree coalgebra.

TODO Not sure which paper to cite here for a deeper dive into the upper topic. Book *Initial Algebras, Terminal Coalgebras, and the Theory of Fixed Points of Functors* by Adámek, Milius and S. Moss from which I have studied is no longer available on the internet, as it was a draft under construction.

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, where finitary means of reasoning are of no use, as we can't expect an end to stream.

Operating on coinductive structures 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.

For a better understanding and intuition, we shall illustrate now the difference based on inductive and coinductive relation. Recall that the set Λ of λ-*terms* is given by the following grammar:

$$e ::= x \mid \lambda x.e \mid e_1 e_2$$

where $\Lambda^0 \subseteq \Lambda$ is a set of *closed* λ-*terms*, meaning, terms without free variables. Let's now define an inductive, and coinductive predicate. Relation $\Downarrow\ \subseteq \Lambda^0 \times \Lambda^0$ (convergence) for call-by-value λ-*calculus* is defined as follows:

$$\frac{}{\lambda x.e \Downarrow \lambda x.e} \qquad \frac{e_1 \Downarrow \lambda x.e_0 \qquad e_0[e_2/x] \Downarrow e'}{e_1 e_2 \Downarrow e'}$$

which means that $\Downarrow$ is the *smallest* predicate *closed forward* under the above rules. Relation can be inductively built from an empty set and then in a fixed-point manner we can add more terms. Let's now see a relation $\Uparrow\ \subseteq \Lambda^0$ of *divergent* terms, which is coinductively defined as follows:

$$\frac{e_1 \Uparrow}{e_1 e_2 \Uparrow} \qquad \frac{e_1 \Downarrow \lambda x.e_0 \qquad e_0[e_2/x] \Uparrow}{e_1 e_2 \Uparrow}$$

which means that $\Uparrow$ is the *greatest* predicate *closed backwards* under the above rules, relation can be coinductively obtained by starting with Λ_0, and then by removing elements that do not fit the rules.

For more detailed introduction into coinduction and other examples, see [20, 36] or for a deeper dive [1].

2.5 Related Work

2.5.1 Algebraic Effects

Except from Links language [18], on which the implementation is based, there are currently many other alternatives available. One may take a look at Frank [27], which provides support for multihandlers, Koka [25], Helium [9] or Eff [6]. Except from separate languages, many libraries arose for existing ones like Haskell, Idris, Scala or Multicore OCaml.

In fact, libraries for algebraic effects also arose in mainstream languages, such as C [24] or Python [12]. In Python, effects are implemented through generators [21], using built-in feature of sending value when doing *yield* operation. Resumptions in handlers that are sending value are one-shot and tail-recursive.

As can be seen in the J. Yallop repository [10], algebraic effects and handlers are now a trending branch in programming languages theory.

2.5.2 Coalgebraic Effects

From theoretical side, research on comodels and its relation to state dates to papers by John Power et. al. [33, 30], and on the runners of comodels to T. Uustalu [38].

Practical part that is closest to topic of the thesis, is work done by Danel Ahman and Andrej Bauer [2], as well as their implementation of a λ_{coop} language [11] and *haskell-coop* [3] library, which is a calculus with runners of coalgebraic effects that ensures linear usage of resources as well as execution of code handling finalisation.

Chapter 3

Potential Solutions

It can be seen, that issues lie with the fact, that interacting with coalgebras may lead to change of their internal state system. Examples include change of state in NFA, taking the next element in an infinite stream or a closing file. Finalization must occur after initialization, thus we conclude that resumption should be called at least once. However, modifying state twice the same way can also lead to unwanted behaviour, as it was seen in Section 1.1, therefore it should be invoked exactly 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 given function is going to be called is a nontrivial semantic property of a program. □

One of the ways to approach this problem, caused by extensive generality of effects, is the introduction of dual Coalgebraic effects, also named coeffects, for which we shall discuss various methods of implementation.

3.1 Dynamic Constraints Checking

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

3.2 Linear Types

Imposition of a single call of resumption, can be thought in a way that continuation is 'consumable', such that it's not available after one takes use of it. Putting it in that way, it immediately reminds of linear logic [14], where premises of linear implications are no longer of use in conclusion, and in case of linear type theory, binding variable from expression A, disallows use of variables in A. Implementation of coalgebraic effects might be simplified in languages that already support linear type system, such as Rust [35].

In fact, there is a bijective translation between language with algebraic effects and calculus with linear type theory, and *every monad embeds in a linear state monad* [28].

3.3 Data-Flow Analysis

Robust resource management could be also achieved by developing a static analysis for checking whether resumption is called only once and that configuration is properly passed.

Data-flow analysis may be the right tool for detecting such anomalies, however, further work in this area is reserved for the future.

3.4 Cohandlers as Separate Constructs

Having observed in Section 2.4 that cooperations correspond to state-passing transitions, we propose a calculus which exactly achieves this semantic. We divide the solution into three parts, discussed below.

3.4.1 State passing

State passing can be done by embedding given coalgebraic theory in a state monad over configurations, which in simpler words means, that for a set of coeffects that we handle, we are going to store current configuration in a state. Configuration is retrieved before performing a coeffect, and new configuration is saved after one is handled. Passing of configuration does not leak to the user, which creates a handy abstraction over operations that manage external state.

3.4.2 Linear usage

Linear usage of resumptions is not going to be addressed by crafting operational semantic that ensures that, as in [2], but rather on a syntactic level where we verify that resumptions in cohandlers are one-shot and tail recursive.

3.4.3 Combining effects with coeffects

Despite of the above means, algebraic effects that occur in the source code may drop the resumption and thus finalization code is not going to be executed. We address this issue by not allowing effects to escape the cohandler, which means that we are not going to allow to discard the execution of cohandler.

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 [7], and thus will not be described in great details. More exemplary programs in Freak language can be found in `src/programs` directory [13].

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 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.

These implications are straightforward when looking from a categorical standpoint, where effects are viewed as free models of algebraic theories [32], and handlers are homomorphisms preserving the model structure [31]. Nevertheless, the results are very exciting for programming use cases.

4.2 Coexamples

We are going to give here informal introduction to programming with coeffects in Freak, based on what we have stated in Section 2.4 and Section 3.4. Elimination of effects has the following syntax:

```
cohandle <Algebraic Theory> using <Initial state> at
    <computation>
through <handler>
```

Where algebraic theory is a name for signature of coeffects that we handle, with initial coalgebra carrier holding `<Initial state>` value. Handler is syntactically same as in the case of standard algebraic effects, however, the parameter passed to cohandler is always a pair of current configuration and actual argument passed by user to effect.

Introduction rule for coeffects is written as `observe Coeffect p`, which references observations from coinduction described in Subsection 2.4.4.

4.2.1 File handling

The first example we are going to show is file handling. As per Section 1.1, we do not want to expose file handle to the user, thus we hide it behind carrier of FileIO coalgebraic theory, for which the signature is composed of Open, Write and Close operations.

```
cohandle FileIO using "filename.txt" at
    let _ <- observe Open () in
    let _ <- observe Write "Karma Chameleon" in
    let _ <- observe Close () in
    return 42
through {
    Open p r ->
        let filename <- return fst p in
        let fh <- do OpenBI filename in
        r (fh, ()) |
    Write p r ->
        let fh <- return fst p in
        let _ <- do WriteBI fh in
```

```
            r (fh, ()) |
        Close p r ->
            let fh <- return fst p in
            let _ <- do CloseBI fh in
            r ((), ()) |
        return x -> return x
    }
```

In the above code, we initialize the coalgebra carrier with filename that we want to handle, we open a file, write to it once and then close it. Notice that file handle is not present in the user code, but only accessible in the runner of FileIO as a first element of a pair.

4.2.2 Nondeterministic Finite Automata

This example shows how one can implement NFA using coeffects. We abstract over internal state of the automata, and based on letters from the alphabet, we change automata configuration and return whether word is accepted or not. The code below returns whether a given NFA accepts the word "abba".

```
cohandle Automata using 0 at
    let _ <- observe NFA "a" in
    let _ <- observe NFA "b" in
    let _ <- observe NFA "b" in
    observe NFA "a"
through {
    NFA p r ->
        let state <- return fst p in
        let letter <- return snd p in
        <code transitions here> |
    return x -> return x
}
```

Full and working example with implementation of automata can be found under `programs/coeffects/NFA.fk` in Freak repository [13].

4.3 Usage guide

As of this day, two implementations are available, one based on the curried translation from Appel [4], and the second one based directly on the uncurried translation with continuations as explicit stacks [17]. More details can be found in Section 6.

All commands are available within the `src` directory. Please note, however, that Apple-based version is no longer supported, and is left as a reference.

4.3.1 Build and install

- Install dependencies: `make install`
- Select implementation: `make link-lists` (recommended) 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:

```
Usage: freak [options] [filename]
Options:
                       Evaluate program
   -c, --cps           CPS-translate program
   -h, --help          Print this message and exit
   -p, --parse         Parse program
   -ds, --desugar      Parse and run AST transformations
   -v, --validate      Run available static analysis
```

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* [19].

- Run tests: `make tests`
- Run code linter: `make lint`
- Validate Freak code from `programs` dir: `make lint`
- Run all above commands: `make check`

Chapter 5

Calculus of Freak language

5.1 Syntax

The syntax for the calculus is shown below. *nat* n represents an integer n, 'string' a string value, $V \oplus W$ 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 [17], with slight syntax modifications. Actual programs in Freak can be found in Section 4.1.

Cohandlers are as they were introduced in Section 4.2, where *AlgTheory* is an identifier representing our signature.

$\langle \textit{Values } V,\ W \rangle ::= x$
$\quad | \ \textit{nat } n \ | \ \texttt{'string'}$
$\quad | \ \backslash x : A \to M$
$\quad | \ \textbf{rec } g\ x \to M$
$\quad | \ V \oplus W \ | \ V \approx W$
$\quad | \ () \ | \ (V, W)$
$\quad | \ \textbf{fst } V \ | \ \textbf{snd } V$

$\langle \textit{Computations } M,\ N \rangle ::= V\ W$
$\quad | \ \textbf{if } V \textbf{ then } M \textbf{ else } N$
$\quad | \ \textbf{return } V$
$\quad | \ \textbf{absurd } V$
$\quad | \ \textbf{let } x \leftarrow M \textbf{ in } N$
$\quad | \ \textbf{do } \ell\ V$
$\quad | \ \textbf{handle } M \textbf{ with } \{H\}$
$\quad | \ \textbf{observe } \ell\ V$
$\quad | \ \textbf{cohandle } \textit{AlgTheory} \textbf{ using } V \textbf{ at } M \textbf{ through } \{H\}$

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

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

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

5.2 Dynamics

Semantic of Freak is heavily based on Links language [17], for which the source language's dynamics have been described extensively by providing small-step operational semantics, continuation passing style transformation [18] as well as abstract machine [15], 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.

Extension of the evaluation contexts:

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

Small-step operational semantics:

$$\begin{aligned} &\mathbf{if}\ nat\ n\ \mathbf{then}\ M\ \mathbf{else}\ N \rightsquigarrow M && \text{if } n \neq 0 \\ &\mathbf{if}\ nat\ n\ \mathbf{then}\ M\ \mathbf{else}\ N \rightsquigarrow N && \text{if } n = 0 \end{aligned}$$

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

5.2.1 Cooperations

Semantics for cooperations is brought from denotational one that we have introduced in Section 2.4, meaning, we denote them as state passing transitions in a code, represented similarly as algebraic effects, through (co)free (co)models of algebraic theories along with their morphisms. Let's put the following cooperation:

$$[\![op]\!]^W : |C| \times P \to |C| \times X$$

Operationally, we embed transitions between coeffectful computations in a standard state monad $S_C X \triangleq (C \Rightarrow X \times C)$. We achieve this by transforming AST of the source code, such that configurations are passed among cooperations. For a given coalgebraic theory T, handler of the state monad in Freak looks as follows:

```
{
    TPut s' r -> return (\s -> let g <- r () in g s') |
    TGet _ r -> return (\s -> let g <- r s in g s) |
    return x -> return (\s -> return x)
}
```

Having that defined, we expand computation

```
    observe Coeffect p
```

into the following one (modulo α-*conversion*):

```
    let Tconf <- observe TGet () in
    let (TnewConf, Tres) <- observe Coeffect (Tconf, p) in
    let _ <- observe TPut TnewConf in
    return Tres
```

where before executing a coeffect, we obtain current configuration, pass it to coeffect, put new config and then return the actual result. This is done behind the scenes and the user does not know about state passing. Note that for readability we have done pattern matching on pairs, while currently in Freak it needs to be deconstructed using projections.

5.2.2 Cohandlers

Cohandlers needs to be aware of configuration passing, therefore, every cohandler accepts $C \times P$ as a parameter where C is current configuration and P parameter passed to coeffect. In a similar way, value passed to resumption also needs to be a pair $C \times X$ of new configuration and value returned down to computation.

For a full currently implemented example of transformation, one may take a look at `programs/transform/effCase` directory in source code.

Chapter 6

Implementation

The Freak implementation is available on github [13], written purely in Haskell. Two inherently different takes at implementations were made. The first one is based on curried translation from A. Appel [4] book, and the second one on Links language [18, 17], on the uncurried translation to target calculus with continuations represented as explicit stacks. We start by presenting core data structures, and afterwards move to actual translation details. Development of the former Appel's version is now suspended.

6.1 Abstract Syntax Tree

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

```
data EffLabel = EffL String
              | CoeffL String

data Value
    = VVar Var
    | VNum Integer
    | VStr String
    | 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
    -- Algebraic effects
    | EOp EffLabel Value
    | EHandle Comp Handler
    -- Coalgebraic effects
    | ECoop EffLabel Value
    | ECohandle Comp Handler
    -- Intermediate representation for a cohandler
    | ECohandleIR AlgTheoryName Value Comp Handler

data Handler
    = HRet Var Comp
    | HOps AlgebraicOp Handler

data AlgebraicOp = AlgOp EffLabel Var Var Comp
```

Where under `Value` in `ECohandleIR` we store initial configuration of a coalgebra, and under `AlgTheoryName` name of the theory over which we operate. We use `EffLabel` structure to syntactically distinguish between effects and coeffects in the code.

Data structures for the calculus are also driven by syntax. 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
    | UStr String
    | UBool Bool
    | ULambda Var ([Cont] -> CPSMonad UComp)
    | UUnit
    | UPair UValue UValue
    | ULabel Label
    | UEffLabel EffLabel
    | URec Var Var UComp
    | UBinOp BinaryOp UValue UValue
```

```
data UComp
    = UVal UValue
    | UApp UComp UComp [Cont]
    | USplit Label Var Var UValue UComp
    | UCase UValue Label UComp Var UComp
    | UIf UValue UComp UComp
    | ULet Var UComp UComp
    | UAbsurd UValue
    | UTopLevelEffect EffLabel UValue
```

Careful reader may notice the correspondence between `UApp` having a stack of continuations, and `ULambda` being a `[Cont] -> CPSMonad UComp` function, which follows directly from translation proposed by Lindley et al˙ [17].

The final answer, common to both evaluations, is represented as a `DValue`, where the meaning of the coproduct is as one would expect. At this point, we no longer treat effect labels as a special case.

```
type Label = String
type EvalMonad a = ExceptT Error (StateT Int IO) a
type FuncRecord = [DValue] -> [Cont] -> EvalMonad DValue

data DValue
    = DNum Integer
    | DStr String
    | DLambda FuncRecord
    | DUnit
    | DPair DValue DValue
    | DLabel Label
```

Evaluation of target calculus is typical to call-by-value with syntactic distinction between values and computations. For more details, refer to [17]. That being said, there is one part that requires more attention, namely, `UTopLevelEffect Label UValue`, which represents an unhandled algebraic effect along with it's parameter. Even in case language has coalgebraic effects, special treatment is required for doing IO. For that reason, we define default handling for a few effects for printing to console and handling files, where semantic is just as one would expect from naming. We abuse haskell's notation to define types for algebraic effects living in Freak language:

```
type Filename = String
Print :: String -> ()
ReadLine :: () -> String
ReadFile :: Filename -> String
```

```
WriteFile :: (Filename, String) -> ()
AppendFile :: (Filename, String) -> ()
```

6.2 Curried translation

The first take was heavily inspired by A. Appel's Compiling with Continuations [4], 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 [4]. The source code for curried translation and evaluation can be found respectively in `CPSAppel.hs` and `EvalCPS.hs`. Development of this version of translation is discontinued.

6.3 Uncurried translation

Having in mind the drawbacks mentioned above, alternative translation was written, that coincides with the translation from [17]. Namely, with the uncurried translation to target calculus with continuations represented as explicit stacks. The target calculus was described in Section 6.1. The continuations are represented as `Cont`, with syntactic distinction between pure, effectful and coeffectful computations. Pure and (co)effectful continuations occupy alternating positions in the stack. Explicit distinction was made to provide more control in the source code.

```
type CPSMonad a = ExceptT Error (StateT Int IO) a

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

data Cont = Pure ContF
          | Eff ContF
          | Coeff ContF
```

Where `CPSMonad` is a monad transformer over `Either`, `State` and `IO`. `State` is required to generate labels for fresh variables that came from the translation, `Either` for handling exceptions that may occur during translations, and `IO` for handling input output of the source language.

```
initialPureCont :: ContF
initialPureCont v ks = (return . UVal) v

initialEffCont :: ContF
initialEffCont (UPair (UEffLabel l) (UPair p r)) ks =
    return $ UApp (UVal r) (UTopLevelEffect l p) k

initialContStack :: [Cont]
initialContStack = [Pure initialPureCont, Eff initialEffCont]
```

Initial pure continuation is lifting values into computations, and effectful one is propagating previously mentioned `UTopLevelEffect`. Notice that `initialContStack` does not have initial cohandler, as handling of default coeffects is embed in evaluator. The core code is split into a few functions:

```
data EffT = EffT | CoeffT
type EvalResMonad a = IO (Either Error a)

cps     :: Comp     -> [Cont]  -> CPSMonad UComp
```

```
cpsVal  :: Value    -> [Cont]  -> CPSMonad UValue
cpsOp   :: EffLabel -> Value   -> [Cont] -> CPSMonad UComp
cpsHRet :: Handler  -> Cont
cpsHOps :: EffT     -> Handler -> Cont
forward :: EffLabel -> UValue  -> UValue -> [Cont] -> CPSMonad UComp
runCPS  :: Comp     -> EvalResMonad UComp
```

Where the first two are implementing cps for computations and values. `cpsOp` is doing translation for an (co)effect that occurred in program, `cpsHRet` and `cpsHOps` are yielding pure and effectful continuations, based on a given handler. `forward` is responsible for forwarding the computation to the outer handler, and the last one for running the translation. This results in an implementation that finally supports nested handlers, which can be seen by running tests. Links language [17] also supports shallow handlers [16], whereas our language implements only deep ones.

6.4 Cohandlers

Similarly as in Section 3.4, we are going to divide this into three parts.

6.4.1 State passing

State passing is defined as it was described in Subsection 5.2.1. From implementation standpoint, transformation is done by manipulating AST of the language to inject constructs handling state management.

6.4.2 Linear usage

TODO it's not implemented yet. Be more clear what's happening here

Linear usages of resumptions is asserted by analyzing parsed AST tree, where in every leaf there must be a return statement. It's value is then passed into continuation of a cohandler.

6.4.3 Combining effects with coeffects

Guards for not letting effect pass cohandler is implemented in CPS translation, more specifically, code that translates cohandler ensures that argument passed in resumption is not an effect. This is done on a static level before evaluation, which is a desired behaviour.

6.5 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
CPSLists.hs     - Uncurried CPS translation
CPSMonad.hs     - Definition of CPSMonad
DValue.hs       - Evaluation result data structures
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
Transform.hs    - Cohandlers transformation described in Calculus chapter
Types.hs        - Common types definition
programs/       - Exemplary programs covered in tests

CPSAppel.hs     - [deprecated] Appel-based CPS translation
EvalCPS.hs      - [deprecated] Evaluation of the Appel's CPS structure
```

Chapter 7

Conclusion

7.1 Summary

We have showed derivation of cooperations, based on previous work, as well as presented a calculus combining algebraic effects with their dual notion. This resulted in a programming language which allows user to seamlessly operate on resources based on solid mathematical foundations. It can seen now, that (co)algebraic effects are an abstract concept that captures plenty of patterns in everyday programming, which already had ad-hoc and more restricted implementations in many existing languages, through constructs such as `try, except, finally`, or by more low-level mechanisms like handling interrupts in operating systems.

This relatively new concept in theory of programming languages yet needs to be researched, and we propose in the next section a few directions into which Freak could go.

7.2 Future work

7.2.1 Abstract machine

The Links language, on which CPS translation Freak was based, also provides small-step operational semantics [18] and an abstract machine [15]. Implementing another way of evaluation could serve as a way to empirically assert correctness, as opposed to formally. Coeffects are living in a syntactic world, which are then translated to algebraic effects, thus we may not need to extend the abstract machine to accommodate them.

7.2.2 Static analysis

The type system as of this day is not implemented, as the focus has been put on CPS transformation and dynamics of the calculus with effects and coeffects. 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. We have been considering row polymorphism as the tool of choice.

Another part could involve statics through data-flow analysis, which we have already mentioned in Section 3.3.

7.2.3 Multiple instances of algebraic effects

Languages with algebraic effects tend to have an mechanism, in which one can instantiate multiple versions of the effects under the same handler code. The current state of the art introduces a concept of resources and instances, as in Eff [7], or instance variables, as in Helium [8]. Work in this area could involve diving into how this concept is related to coalgebras, and whether to implement that in Freak.

7.2.4 Selective CPS

Other languages, like Koka [25], 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 CPS transformation.

7.2.5 Exceptions and signals

Exceptions are a trivial example of algebraic effect where the resumption is discarded, and as described in §4.5 [18] they can be modeled as a separate construct. On the one hand, it may lead to performance improvements, on the other, we could use this notion to ensure that finalisation code for coeffects is executed even in the case of a failure, as described in [2].

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. [16] could be another way of enhancing the language.

Bibliography

[1] *Advanced Topics in Bisimulation and Coinduction.* Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2011. DOI: 10.1017/CBO9780511792588.

[2] 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.

[3] *An experimental Haskell library for programming with effectful runners.* URL: https://github.com/danelahman/haskell-coop.

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

[5] Andrej Bauer. "What is algebraic about algebraic effects and handlers?" In: *CoRR* abs/1807.05923 (2018). arXiv: 1807.05923. URL: http://arxiv.org/abs/1807.05923.

[6] 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.

[7] 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.

[8] Dariusz Biernacki et al. "Binders by Day, Labels by Night: Effect Instances via Lexically Scoped Handlers". In: *Proc. ACM Program. Lang.* 4.POPL (Dec. 2019). DOI: 10.1145/3371116. URL: https://doi.org/10.1145/3371116.

[9] 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.

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

[11] *Coop, language with coalgebraic effects.* URL: https://github.com/andrejbauer/coop.

[12] *Effect: library in Python implementing algebraic effects.* URL: `https://github.com/python-effect/effect`.

[13] *Freak, functional programming language with (co)algebraic effects and (co)handlers.* URL: `https://github.com/Tomatosoup97/freak`.

[14] Jean-Yves Girard. "Linear logic". In: *Theoretical Computer Science* 50.1 (1987), pp. 1 –101. ISSN: 0304-3975. DOI: `https://doi.org/10.1016/0304-3975(87)90045-4`. URL: `http://www.sciencedirect.com/science/article/pii/0304397587900454`.

[15] Daniel Hillerström and Sam Lindley. "Liberating Effects with Rows and Handlers". In: *Proceedings of the 1st International Workshop on Type-Driven Development*. TyDe 2016. Nara, Japan: Association for Computing Machinery, 2016, 15–27. ISBN: 9781450344357. DOI: `10.1145/2976022.2976033`. URL: `https://doi.org/10.1145/2976022.2976033`.

[16] Daniel Hillerström and Sam Lindley. "Shallow Effect Handlers". In: *Programming Languages and Systems*. Ed. by Sukyoung Ryu. Cham: Springer International Publishing, 2018, pp. 415–435. ISBN: 978-3-030-02768-1.

[17] Daniel Hillerström, Sam Lindley, and Robert Atkey. "Effect Handlers via Generalised Continuations". In: *Journal of Functional Programming* 30 (2020). DOI: `10.1017/S0956796820000040`.

[18] Daniel Hillerström et al. "Continuation Passing Style for Effect Handlers". In: *2nd International Conference on Formal Structures for Computation and Deduction (FSCD 2017)*. Ed. by Dale Miller. Vol. 84. Leibniz International Proceedings in Informatics (LIPIcs). 2017. DOI: `10.4230/LIPIcs.FSCD.2017.18`.

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

[20] Bart Jacobs and Jan Rutten. "A Tutorial on (Co)Algebras and (Co)Induction". In: *EATCS Bulletin* 62 (1997), pp. 62–222.

[21] Satoru Kawahara and Yukiyoshi Kameyama. "One-shot Algebraic Effects as Coroutines". In: (2020).

[22] *Kleisli category definition on ncatlab.* URL: `https://ncatlab.org/nlab/show/Kleisli+category`.

[23] Daan Leijen. *Algebraic Effects for Functional Programming*. Tech. rep. MSR-TR-2016-29. 2016, p. 15. URL: `https://www.microsoft.com/en-us/research/publication/algebraic-effects-for-functional-programming/`.

[24] Daan Leijen. *Implementing Algebraic Effects in C*. Tech. rep. MSR-TR-2017-23. 2017. URL: `https://www.microsoft.com/en-us/research/publication/implementing-algebraic-effects-c/`.

[25] Daan Leijen. "Type Directed Compilation of Row-Typed Algebraic Effects". In: *Proceedings of Principles of Programming Languages (POPL'17), Paris, France*. 2017. URL: https://www.microsoft.com/en-us/research/publication/type-directed-compilation-row-typed-algebraic-effects/.

[26] Sheng Liang, Paul Hudak, and Mark Jones. "Monad Transformers and Modular Interpreters". In: POPL '95. San Francisco, California, USA: Association for Computing Machinery, 1995, 333–343. ISBN: 0897916921. DOI: 10.1145/199448.199528. URL: https://doi.org/10.1145/199448.199528.

[27] 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.

[28] Rasmus Mogelberg and Sam Staton. "Linear usage of state". In: *Logical Methods in Computer Science* 10 (Mar. 2014). DOI: 10.2168/LMCS-10(1:17)2014.

[29] 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.

[30] Gordon Plotkin and John Power. "Tensors of Comodels and Models for Operational Semantics". In: *Electronic Notes in Theoretical Computer Science* 218 (Oct. 2008), pp. 295–311. DOI: 10.1016/j.entcs.2008.10.018. URL: https://doi.org/10.1016/j.entcs.2008.10.018.

[31] 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.

[32] 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.

[33] John Power and Olha Shkaravska. "From Comodels to Coalgebras: State and Arrays". In: *Electronic Notes in Theoretical Computer Science* 106 (Dec. 2004), pp. 297–314. DOI: 10.1016/j.entcs.2004.02.041. URL: https://doi.org/10.1016/j.entcs.2004.02.041.

[34] Matija Pretnar. "An Introduction to Algebraic Effects and Handlers. Invited Tutorial Paper". In: *Electron. Notes Theor. Comput. Sci.* 319.C (Dec. 2015), 19–35. ISSN: 1571-0661. DOI: 10.1016/j.entcs.2015.12.003. URL: https://doi.org/10.1016/j.entcs.2015.12.003.

[35] *Rust language webpage*. URL: https://www.rust-lang.org/.

[36] Davide Sangiorgi. *Introduction to Bisimulation and Coinduction*. Cambridge University Press, 2011. DOI: 10.1017/CBO9780511777110.

[37] J.P. Talpin and P. Jouvelot. “The Type and Effect Discipline”. In: *Information and Computation* 111.2 (1994), pp. 245 –296. ISSN: 0890-5401. DOI: `https://doi.org/10.1006/inco.1994.1046`. URL: `http://www.sciencedirect.com/science/article/pii/S0890540184710467`.

[38] Tarmo Uustalu. “Stateful Runners of Effectful Computations”. In: *Electronic Notes in Theoretical Computer Science* 319 (Dec. 2015), pp. 403–421. DOI: `10.1016/j.entcs.2015.12.024`. URL: `https://doi.org/10.1016/j.entcs.2015.12.024`.

[39] Philip Wadler. “The Essence of Functional Programming”. In: POPL ’92. Albuquerque, New Mexico, USA: Association for Computing Machinery, 1992, 1–14. ISBN: 0897914538. DOI: `10.1145/143165.143169`. URL: `https://doi.org/10.1145/143165.143169`.