Continuation Passing Style for Effect Handlers

Mateusz Urbańczyk

31 January 2020

Abstract

We provide an implementation of algebraic effects and handlers by doing continuation passing style transformation of the functional programming language Freak, which is based on the existing Links language [9].

1 Introduction

In this article we present experimental programming language Freak, which is an implementation of Continuation Passing Style for Effect Handlers paper [9], with the addition of a few basic constructs. 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.

2 State of the art

Except from Links language [9], on which the implementation is based, there are currently many other alternatives available. One may take a look at Frank [12], which provides a support for multihandlers, Koka [11], Helium [5] or Eff [2]. 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 [6], algebraic effects and handlers are now trending branch in the programming languages theory.

3 Syntax

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 7.

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

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

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

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

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

4 Operational semantics

The source language's dynamics have been described extensively by providing small-step operational semantics, continuation passing style transformation [9] as well as abstract machine [7], 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:

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

Small-step operational semantics:

$\textbf{if } \textit{nat } n \textbf{ then } M \textbf{ else } N \rightsquigarrow M \qquad \text{if } n \neq 0$
$\textbf{if } \textit{nat } n \textbf{ then } M \textbf{ else } N \rightsquigarrow N \qquad \text{if } n = 0$

$\textit{nat } n \oplus \textit{nat } n' \rightsquigarrow n'' \qquad \text{if } n'' = n \oplus n'$
$\textit{nat } n \approx \textit{nat } n' \rightsquigarrow 1 \qquad \text{if } n \approx n'$
$\textit{nat } n \approx \textit{nat } n' \rightsquigarrow 0 \qquad \text{if } n \not\approx n'$

5 Usage guide

As of this day, two implementations are available, one based on the curried translation and Appel [1], 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.

5.1 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 7.1

5.2 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* [10].

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

6 Implementation

The Freak implementation is available here, 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 [1] 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 Trees

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 Curried translation

The first take was heavily inspired by A. Appel's Compiling with Continuations [1], 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
```

Each term at the end has 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 that the book cited above. The source code for curried translation and evaluation can be found respectively in `CPSAppel.hs` and `EvalCPS.hs`.

6.3 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 computation, 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 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`.

6.4 Source code structure

The source code is divided into a number of modules, each with a different responsibility.

```
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
```

7 Examples

In this section we present a few examples to show the capabilities of the language. The ideas have been based on [3], and thus will not be described in great details. More exemplary programs in Freak language can be found here.

7.1 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))`.

7.2 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.

7.3 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 [14], and handlers are homomorphisms preserving the model structure [13]. Nevertheless, the results are very exciting for programming use cases. The current Freak implementation does not support I/O.

8 Future work

The Freak language is experimental and a lot of possible enhancements could be adapted. Here we provide a list of the proposed directions in which further work could be done.

8.1 Alternative evaluation

The Links language also provides small-step operational semantics and an abstract machine [7]. Implementing another way of evaluation could serve as a way to empirically assert correctness, as opposed to formally.

8.2 Type inference

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.

8.3 Make the language more usable

While the language is Turing-complete, for convenient usage it requires more basic constructs and syntactic sugar for common patterns that would ease the programming.

8.4 Multiple instances of algebraic effect

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 [3], or instance variables, as in Helium [4].

8.5 Selective CPS

Other languages, like Koka [11], 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.

8.6 Exceptions as separate constructs

Exceptions are a trivial example of algebraic effect where the resumption is discarded, and as described in §4.5 [9], they can be modeled as a separate construct to improve performance.

8.7 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. [8] could be another way of enhancing Freak.

References

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

[2] 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.

[3] 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.

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

[5] 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.

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

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

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

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

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

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

[12] 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.

[13] 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.

[14] 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.