Compiling Elaborations of Higher-Order Effects First-Stage Review

Terts Diepraam

April 20, 2023

1 Introduction

This document explains the current progress on my thesis project. It starts by establishing the research gap. Then gives the goals for the project, the current progress and the plans to the remainder of the project.

2 Previous Research

In standard λ-calculus, there is no model for effects of the computation, only of the result. Speaking broadly, effects concern the aspects of the program besides the pure computation. [5]

In their seminal paper, Moggi [5] unified monads with computational effects, or notions of computation as they call it. Functional programming languages often rely on monads to perform effectful computations. Haskell, for example, uses an `IO` monad for printing output and reading input. The computation is then kept pure and impure operations (like reading and printing) are delegated to the system. However, a limitation of treating effects as monads is that monads do not compose well.

Plotkin and Power [6] then showed that effects can be represented as equational theories. The effects that can be represented in this framework are called algebraic effects. The `State` and the `Maybe` monads can for example be expressed in this framework.

Plotkin and Pretnar [7] then introduced effect handlers, allowing the programmer to destruct effects by placing handlers around effectful expressions. This provides a way to treat exception handling using effects. However, the scope in which an effect is handled can only be changed by adding handlers. Effect operations cannot define their own scope. To support this, a system for higher-order effects is required, which are effects that take effectful operations as parameters.

A solution to this was the framework of scoped effects [10]. However, scoped effects require a significant increase in complexity and cannot express effects that are neither algebraic nor scoped, such as lambda abstractions [9]. Latent

effects [9] were subsequently introduced as an alternative that encapsulates a larger set of effects.

As an alternative approach to latent effects, Bach Poulsen and van der Rest [1] introduced hefty algebras. With hefty algebras, higher-order effects are treated separately from algebraic effects. Higher-order effects are not handled, but elaborated into algebraic effects, which can then be handled. The advantage is that the treatment of algebraic effects remains intact and that the process of elaboration is relatively simple.

In parallel with the work to define theoretical frameworks for effects, several libraries and languages have been designed that include effects as first-class concepts, allowing the programmer to define their own effects and handlers. For example, there are some libraries available for Haskell, like `fused-effects`[1] , `polysemy`[2] , `freer-simple`[3] and `eff`[4] , each encoding effects in a slightly different way.

Notable examples of languages with support for algebraic effects are Eff [2], Koka [3] and Frank [4]. OCaml also gained support for effects [8]. By building algebraic effects into the language, instead of delegating to a library, is advantageous because the language can provide more convenient syntax. In these languages, some concepts that were traditionally only available with language support, can be expressed by the programmer. This includes exception handling and asynchronous programming.

These languages either only support algebraic effects or scoped effects. This means that their support for higher-order effects is limited. The exception is `heft`[5] , which is produced in conjunction with the work in [1].

3 Goals of the Project

The main research question for this project is: **"Can we transform a program with elaborations and higher-order effects to a program with algebraic effects?"** If such a transformation exists, we aim to answer further questions, such as:

- Can this transformation be incremental?
- Can we transform the program without type information?
- Does this transformation apply to existing languages with algebraic effects?
- Can we apply the transformation to libraries for algebraic effects in languages that do not have first-class support for algebraic effects?

[1] `https://github.com/fused-effects/fused-effects`
[2] `https://github.com/polysemy-research/polysemy`
[3] `https://github.com/lexi-lambda/freer-simple`
[4] `https://github.com/hasura/eff`
[5] `https://github.com/heft-lang/heft`

If we cannot find such a transformation, we can ask what limitations we can impose such that we can find a suitable transformation. For example, it might be possible that elaborations must be known statically to be able to compile them.

In addition, we explore whether we can infer which elaborations to apply based on the context, removing the need to explicitly specify the elaborations. Furthermore, we explore whether it is possible (or even desirable) to do similar inference for handlers of algebraic effects.

To this end, we create a new language called `elaine`[6], building on the work in [1], which

- supports higher-order effects using hefty algebras, with separate elaborations and handlers,
- implicitly resolves elaborations, and
- can be compiled to a representation with only algebraic effects.

To answer the research question, we need to show that the compilation of higher-order effects we define is equivalent to the operational semantics for elaborations.

By showing that elaborations can be removed at compile-time, we are able to connect the theory of hefty algebras more tightly with the literature on algebraic effects. For example, we show that it is possible to compile programs in our language using the techniques from [3] and we provide an easy mechanism to add support for higher-order effects to languages with support for algebraic effects.

Finally, we aim to provide an implementation of `elaine`[7] . The implementation allows for experimentation with the techniques from the thesis and it gives us the opportunity to write and test more complex programs.

4 Current State of the Project

We have specified the elaine language by defining a syntax, typing judgments, implicit resolution of elaborations and operational semantics.

The implicit elaborations are resolved by looking up the elaborations in the current context. If there are multiple elaborations for a higher-order effect in the context, we have a type error. We have not explored the same mechanism for handlers yet. The current hypothesis is inference for handlers is not desirable, because handlers often require parameters, though it could be a nice feature to have for the sake of symmetry between handlers and elaborations.

First, there are a couple of failed attempts at defining the transformation. While these are not useful themselves, they yielded some interesting edge cases of programs for which the transformation is difficult.

[6] A name which was chosen because it shares a prefix with "elaboration", which is an integral part of the language.

[7] `https://github.com/tertsdiepraam/thesis/tree/main/elaine`

In the process, we found a type system that (with some overapproximation) gives us information about which **handle** and **elab** constructs apply to which operations in the syntax tree. While this is not used in the current definition of the transformation, it might give us information that might be used for more optimized compilation.

The first difficult example is a program in which an effectful lambda is applied, which shows that we cannot simply replace occurrences of $a!$ within any given **elab**, because the operation might be in other parts of the syntax tree.

```
let f = λx . elab[e] { x () }
f(λ x . a!())
```

The second difficult case is a program in which multiple **elab**s might apply, depending on some runtime value. In the example below, if the runtime condition evaluates to **true** then $e1$ is applied, but $e2$ is applied if not.

```
let f = λx . {
    let _ = a!()
    ()
}
elab[e1] {
    let g = elab[e2] {
        if <runtime condition> then
            f
        else {
            f()
            λx . ()
        }
    }
    g()
}
```

However, we can still define a surprisingly simple transformation that works in all of these cases. First, we make all elaborations explicit, that is, we transform each "**elab** $\{e\}$" to "**elab**$[E_1]$ $\{$**elab**$[E_2]$ $\{$... **elab**$[E_n]$ $\{e\}$ ...$\}\}$". Now that we have an elab for each effect. Without loss of generality, consider only one effect $A!$ with operations $op_1!, \ldots, op_m!$. In the whole program there is a finite list of elaborations $E_1^A, \ldots, E_n^A$ for $A!$. Let $E_i^A[op_j!(x_1, \ldots, x_k)]$ denote the elaboration of $op_j!(x_1, \ldots, x_k)$ as defined by E_i. The transformation is then defined as a fold over the syntax tree which is the identity except in the following cases:

$$\textbf{elab}[E_i^A]\ \{e\} \quad \Longrightarrow \quad \textbf{handle}\ H_A(i)\ \{e\}$$

$$op_j!(x_1, ..., x_k) \quad \Longrightarrow \quad \begin{array}{l} \{ \\ \quad \textbf{let}\ i = askElab_A() \\ \quad \textbf{if}\ eq(i,\ 1)\ \textbf{then}\ E_1^A[op_j!(x_1, \ldots, x_k)] \\ \quad \textbf{else if}\ eq(i,\ 2)\ \textbf{then}\ E_2^A[op_j!(x_1, \ldots, x_k)] \\ \quad ... \\ \quad \textbf{else if}\ eq(i,\ N{-}1)\ \textbf{then}\ E_{N-1}^A[op_j!(x_1, \ldots, x_k)] \\ \quad \textbf{else}\ E_N^A[op_j!(x_1, \ldots, x_k)] \\ \} \end{array}$$

The function H_A returns a handler for the effect with the operation $askElab_A$ defined as

```
let H_A = λi . handler {
    return(x) { x }
    askElab_A() { resume(i) }
}
```

The effect in this transformation is meant to evoke the `Reader` effect, where the $askElab_A$ operation "asks" which elaboration is active in a given scope. This transformation works because of the similarities between the semantics of handlers and elaborations.

The transformation has several nice properties. Apart from finding all relevant elaborations, it is entirely local, making it easy to reason about and implement. It is also consistent across the entire program: every operation in the program and every elaboration is expanded in the same way. As long as no elaborations are added or removed from the program, it is also incremental, meaning that previously transformed parts of the program can be cached. With some further research, we might be able to make it fully incremental, though this is non-trivial. Using the type inference, we might also be able to simplify the transformation in some cases. For example, in the cases where only one elaboration might apply to a given operation, we can remove the handler and the if-expression and replace the operation directly with the elaboration.

We have a parser and interpreter implementing most of what is described above, though resolution of implicit elaborations is still missing. The compilation of higher-order effects also has not been implemented. There is a test suite with implementations of common (higher-order) effects, though this list has to be expanded.

5 Plan for the Remainder of the Project

A few small changes should be still made to the specification of the language. Most importantly, elaborations should become first-class values, just like handlers. This means that there will be two **elab** constructs: an explicit **elab** and an implicit **elab**. The explicit elab (tentatively written **elab**$[ej]$) has a sub-

expression that evaluates to the elaboration. The implicit elab infers the correct elaborations from the values in scope. Another difference between the two is that the implicit elab can elaborate multiple effects, where the explicit elab can elaborate just one. This is not a fundamental limitation, but is just for practical reasons since the language lacks support for tuples and hence a set of multiple elaborations is difficult to represent.

We can then extend the compilation to programs in which elaborations are first-class values. This is probably quite simple, as it just adds a case to the transformation for an elaboration, while keeping the same general idea.

Once that is done, the interpreter needs to be updated to follow that new specification. Then we can implement the compilation of higher-order effects and test that. The test suite and an informal proof will comprise a convincing argument about the correctness of the transformation.

The test suite also needs to be expanded significantly. We need both more examples of practical effects and examples of edge cases like the ones described in the previous section.

If there is time left to extend the language, we can attempt to write a type checker. However, since the type information has little to do with the compilation or the inference of implicit elaborations, a type checker has a low priority. We can also provide a more extensive set of primitives (e.g. floating point numbers, lists & tuples) and define system handlers of effects which are implemented by the runtime. For example, we can define a handler for the `Writer` effect which writes directly to `stdout`. While these extensions are not necessary or theoretically very interesting, they do allow us to write more interesting example programs.

References

[1] C. Bach Poulsen and C. van der Rest. "Hefty Algebras: Modular Elaboration of Higher-Order Algebraic Effects". In: *Proceedings of the ACM on Programming Languages* 7 (POPL Jan. 9, 2023), pp. 1801–1831. ISSN: 2475-1421. DOI: 10.1145/3571255. URL: https://dl.acm.org/doi/10.1145/3571255 (visited on 01/26/2023).

[2] A. Bauer and M. Pretnar. "Programming with algebraic effects and handlers". In: *Journal of Logical and Algebraic Methods in Programming* 84.1 (Jan. 2015), pp. 108–123. ISSN: 23522208. DOI: 10.1016/j.jlamp.2014.02.001. URL: https://linkinghub.elsevier.com/retrieve/pii/S2352220814000194 (visited on 04/03/2023).

[3] D. Leijen. "Type directed compilation of row-typed algebraic effects". In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. POPL '17: The 44th Annual ACM SIGPLAN Symposium on Principles of Programming Languages. Paris France: ACM, Jan. 2017, pp. 486–499. ISBN: 978-1-4503-4660-3. DOI: 10.1145/3009837.

3009872. URL: https://dl.acm.org/doi/10.1145/3009837.3009872 (visited on 04/08/2023).

[4] S. Lindley, C. McBride, and C. McLaughlin. *Do be do be do*. Oct. 3, 2017. arXiv: 1611.09259[cs]. URL: http://arxiv.org/abs/1611.09259 (visited on 04/08/2023).

[5] E. Moggi. "Computational lambda-calculus and monads". In: *[1989] Proceedings. Fourth Annual Symposium on Logic in Computer Science*. [1989] Proceedings. Fourth Annual Symposium on Logic in Computer Science. Pacific Grove, CA, USA: IEEE Comput. Soc. Press, 1989, pp. 14–23. ISBN: 978-0-8186-1954-0. DOI: 10.1109/LICS.1989.39155. URL: http://ieeexplore.ieee.org/document/39155/ (visited on 04/08/2023).

[6] G. Plotkin and J. Power. "Adequacy for Algebraic Effects". In: *Foundations of Software Science and Computation Structures*. Ed. by F. Honsell and M. Miculan. Red. by G. Goos, J. Hartmanis, and J. van Leeuwen. Vol. 2030. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 1–24. ISBN: 978-3-540-41864-1 978-3-540-45315-4. DOI: 10.1007/3-540-45315-6_1. URL: http://link.springer.com/10.1007/3-540-45315-6_1 (visited on 04/08/2023).

[7] G. Plotkin and M. Pretnar. "Handlers of Algebraic Effects". In: *Programming Languages and Systems*. Ed. by G. Castagna. Vol. 5502. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 80–94. ISBN: 978-3-642-00589-3 978-3-642-00590-9. DOI: 10.1007/978-3-642-00590-9_7. URL: http://link.springer.com/10.1007/978-3-642-00590-9_7 (visited on 04/08/2023).

[8] K. C. Sivaramakrishnan et al. "Retrofitting Effect Handlers onto OCaml". In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. June 19, 2021, pp. 206–221. DOI: 10.1145/3453483.3454039. arXiv: 2104.00250[cs]. URL: http://arxiv.org/abs/2104.00250 (visited on 04/08/2023).

[9] B. van den Berg et al. "Latent Effects for Reusable Language Components". In: *Programming Languages and Systems*. Ed. by H. Oh. Vol. 13008. Series Title: Lecture Notes in Computer Science. Cham: Springer International Publishing, 2021, pp. 182–201. ISBN: 978-3-030-89050-6 978-3-030-89051-3. DOI: 10.1007/978-3-030-89051-3_11. URL: https://link.springer.com/10.1007/978-3-030-89051-3_11 (visited on 01/12/2023).

[10] N. Wu, T. Schrijvers, and R. Hinze. *Effect Handlers in Scope*. June 10, 2014.