Géza Csenger
Homerton College
gc562

Part II Project Proposal

Compiling Algebraic Effect Handlers

May 6, 2020

Project Originator: *Géza Csenger*

Resources Required: No extra resources are required.

Project Supervisor: *Professor Alan Mycroft*

Director of Studies: *Dr John Fawcett*

Overseers: *Professor Frank Stajano* and *Professor Simone Teufel*

```
io = effect {
    print : string -> ()
}

reversing_handler = handler {
    print (str; k) -> k (); stdout_print(str)
}

with reversing_handler handle {
    perform print("A");
    perform print("B");
    perform print("C")
}
```

Figure 1: Example code with effect signature, an effect handler and a *with ... handle ...* statement.

Introduction and Description of the Work

Handlers of algebraic effects were first described by Plotkin and Pretnar [1]. They generalise the exception-handling construct of Benton and Kennedy [2]. Today one can find many implementations of effect handlers, typically as libraries in various functional languages, but there are also new programming languages with first-class algebraic effects and handlers. One such ML-like programming language is Eff [3]. This project will be concerned with writing a compiler for core Eff [4].

Effect handlers

Effect handlers are a generalisation of exception handlers. In an effect handler the continuation of the computation performing the effect is exposed as a resumable computation (see `k` in Figure 1). Handlers allow us to stop programs before an effect is performed and to give meaning to an already declared effect.

The code sample in Figure 1 demonstrates how one can program with effect handlers. Note how we first need to declare the effect by specifying its signature[1]. A `handler` is a value in Eff and is essentially a map from effect signatures to interpretations of those effects. The example in Figure 1 will execute the `print` statements in the reverse order in the context of `reversing_handler`.

An effect handler can be used with the `with ... handle { ... }` construct. Whenever the enclosed piece of code attempts to perform an effect, a corresponding rule is looked up from the most recent[2] effect handler and execution proceeds according to that rule. One might think about `perform` clauses as *effect constructors* and effect handlers as *effect destructors*.

Making continuations first-class citizens of a language gives a lot of flexibility to the programmer but complicates the implementation of the compiler. A continuation in an effect handler might stay unused (e.g., in case of an exception), used once (one-shot continuation) or invoked more than once (multi-shot continuation).

Different uses of continuations require different compilation techniques if optimal performance is to be achieved (see the next section for examples).

Existing solutions

Multicore OCaml implements continuations using *fibers* which are essentially heap-allocated dynamically resized stacks [5]. To avoid copy overheads Multicore OCaml only supports one-shot continuations.

Capturing a delimited execution context is not a problem when we have full control over the runtime system (see OCaml) but this is not always the case. The Koka programming language [6] uses a type-and-effect driven CPS compilation scheme to implement handlers on runtime systems where one doesn't have this kind of control (such as the JVM, .NET or JavaScript) [7].

The Links [8] interpreter uses a CEK-style abstract machine [9].

[1] This is very similar to interfaces we know from object oriented languages.
[2] *with ... handle ...* constructs can be nested.

Starting point

Effect handlers

Algebraic effects and handlers are not taught as part of the Tripos. Over the summer I devoted time to study them and to get myself familiar with the literature. However, further research will be needed to properly understand the tradeoffs between the ways effect handlers can be compiled. This time is incorporated in the project timeline.

OCaml

I plan to implement my project in OCaml as it is a popular language choice among compiler implementers. In the Part IB Compiler Construction course we used OCaml to build a toy compiler for a small language, therefore I have some experience with programming in OCaml. During the summer I also read parts of the *Real World OCaml* book and experimented with the OCaml ecosystem: I learned how to use the Dune build system, the Ocaml Package Manager and tried to write simple parsers with `ocamllex` and `ocamlyacc`. I implemented simple unit tests for these using OUnit and Alcotest and decided to use the latter framework for my project.

Another advantage of using OCaml is that `ocamlopt` is available on the MCS machines. This means that I'll still be able to compile and finish my project in case my own machines fail.

Substance and Structure of the Project

The goal of my project will be to implement a compiler for core Eff [4].

Core project

The project will consist of four parts which will be implemented in the same order as presented here:

1. Lexer and parser
2. High level interpreter on parse trees
3. Translator from parse tree to bytecode
4. Bytecode interpreter

The correctness of each component depends on the correctness of the previously implemented components. Hence thorough testing is crucial in every part of the project. I intend to achieve this by writing unit tests for each component and use already tested components to generate test cases for the ones under development.

Implementing the first two components should be relatively straightforward. I expect to spend more time with the last two parts because the low level implementation of a programming language is much more error prone.

Success Criterion

My success criterion is to demonstrate that my interpreter and compiler is capable of evaluating core Eff code correctly and with reasonable efficiency.

Quantitative evaluation will happen against Eff v5.0 [10] which is the most recent implementation of the Eff language. One would also expect the bytecode interpreter to perform much better than the high level interpreter interpreting parse trees.

The correctness of the implementation can be evaluated in a similar way: with a well-chosen set of code examples one should be able to test whether my implementation gives the same results as the reference implementation. The core deliverable feature set is the minimal set of language features that makes up core Eff: effect signatures, `with ... handle { ... }`, `perform` and usual OCaml-like features such as functions, function applications, `let`, `let rec` and `match` [4].

Possible extensions

The following extensions are all concerned with optimising the compiler in some form or another. Hence, evaluating the extensions could happen in a quantitative way too: one could compare the compiler's performance[3] with and without the optimisation.

[3] Compiler performance is not well-defined. Depending on the context this can mean execution time, memory usage or even the length of the generated code.

Implement tail resumption optimisations

Resumptions in a tail-call position inside effect handlers are called tail resumptions. An analogous optimisation to tail-call elimination is possible with effect handlers as well.

Optimisations based on the use of continuations

Based on how a continuation is used in an effect handler one can take different decisions at compilation time. For instance, Multicore OCaml handles exceptions separate from effects for efficiency reasons even though an exception is just the special case of an effect where the continuation is discarded.

OCaml-like fibers

It would be interesting to extend the bytecode interpreter with OCaml-like fibers and see what performance benefits arise from this.

Work plan

My work plan consists of 10 2-week work packages. The first 5 packages are focussed on implementing the core project, while the last 5 packages are concerned with carrying out extensions and writing up the dissertation.

There are *slack times* and *catch-up times* in between the work packages. Slack times are time periods where it's predictable that the project might not progress much, while catch-up times are meant to account for delays.

25th October – 7th November — 1st work package

Set up project (OCaml, build system, testing, backup system) and decide on a software development methodology to be used. Implement and test the first part of the project.

Milestone 1: *Project is set up.*
Milestone 2: *Lexer and parser for core Eff is implemented and tested.*

8th November – 21st November — 2nd work package

Prepare unit tests for the high level interpreter. Create implementation passing the tests. The correctness of this implementation is tested and evaluated against the reference implementation.

Milestone 3: *Unit tests for the interpreter are delivered.*
Milestone 4: *Implementation of the second part of the project is delivered.*

22nd November – 28th November — Slack time

Slack time to account for potential delay and for the increased workload at the end of term due to Units of Assessments.

29th November – 12th December — 3rd work package

Design a suitable intermediate representation for the translator and the bytecode interpreter. This involves further research into how existing solutions work. The results of these findings will be sent to my supervisor. If time permits work will be started on the second half of the project.

Milestone 5: *Ideas about the design of the intermediate representation and detailed plan of implementation for the rest of the project is sent to supervisor.*

13th December – 26th December — 4th work package

Implement the translator from parse trees to bytecode.

Milestone 6: *Translator and a set of corresponding unit tests is delivered.*

27th December – 2nd January — Slack time

Slack time due to Christmas and New Year's Day.

3rd January – 16th January — 5th work package

Implement the bytecode interpreter. After the bytecode interpreter is implemented, work on end-to-end tests and make sure that all parts of the compiler work together well. Evaluate the performance of the compiler against the reference implementation and against the simple interpreter.

Milestone 7: *Bytecode interpreter, unit tests and end-to-end tests are delivered.*
Milestone 8: *Evaluation of core deliverable is complete and success criterion is met.*

17th January – 23rd January — Catch-up time

Catch-up time accounting for potential unforeseeable delays. Work should be started on the progress report so that it is easy to submit it in the next work package.

Milestone 9: *A draft of the progress report is written.*

24th January – 6th February — 6th work package

Research how to proceed with extensions. Make an implementation and an evaluation plan and send these to my supervisor.

Milestone 10: *Document with the implementation plan sent to the supervisor.*
Milestone 11: *Progress report is submitted.*

7th February – 20th February — 7th work package

Work on extensions and on initial drafts of the dissertation's Introduction and Preparation chapters.

Milestone 12: *Drafts of Introduction and Preparation chapters are sent to supervisor.*

21st February – 27th February — Catch-up time

Catch-up time accounting for potential unforeseeable delays.

28th February – 12th March — 8th work package

Work on extensions and on the initial drafts of the dissertation's Implementation and Evaluation chapters. Feedback will be received on the Introduction and Preparation chapters during this work package.

Milestone 13: *Drafts of Implementation and Evaluation chapters are sent to supervisor.*
Milestone 14: *Supervisor's feedback on previous drafts is addressed.*

13th March – 26th March — 9th work package

Evaluate any extensions completed so far. Write an initial draft of the dissertation's Conclusion chapter. With this the first full draft of the dissertation should be complete.

Milestone 14: *Evaluation of extensions is complete.*
Milestone 15: *Full draft dissertation is sent to supervisor and DoS.*

27th March – 2nd April — Catch-up time

Catch-up time accounting for potential unforeseeable delays. This gap also gives a chance for the supervisor and DoS to read through the full draft dissertation from the previous work package.

3rd April – 16th April — 10th work package

Carry out final changes to the dissertation.

Milestone 16: *Final draft of dissertation is complete.*

Resource Declaration

Personal laptop (Lenovo Thinkpad X1 Carbon, 4th gen.)

Specifications: Intel Core i7-6600U CPU @ 2.60Ghz x 4, 8GB RAM, Ubuntu 18.04 LTS

I will use my personal laptop to write and compile code, to execute unit tests and to write my dissertation. I accept full responsibility for this machine. I have made contingency plans to protect the project against hardware or software failures.

Both the source code of my project and my dissertation will be version controlled via git and will be periodically uploaded to private GitHub repositories. In case my machine fails, I will still be able to access my project online and continue making progress on the MCS machines in the Computer Laboratory.

I will also use GNOME's Déjà Dup backup tool to create weekly backups of my `/home` folder, upload them to Google Drive and save them to an external hard disk.

References

[1] G. Plotkin and M. Pretnar, "Handlers of algebraic effects," in *Programming Languages and Systems* (G. Castagna, ed.), (Berlin, Heidelberg), pp. 80–94, Springer Berlin Heidelberg, 2009.

[2] N. Benton and A. Kennedy, "Exceptional syntax," *Journal of Functional Programming*, vol. 11, pp. 395–410, 07 2001.

[3] "Eff programming language." `http://www.eff-lang.org/`. Accessed: 2019-10-16.

[4] A. Bauer and M. Pretnar, "An effect system for algebraic effects and handlers," *Logical Methods in Computer Science*, vol. 10, no. 4, 2014.

[5] S. Dolan, L. White, K. Sivaramakrishnan, J. Yallop, and A. Madhavapeddy, "Effective concurrency through algebraic effects," in *OCaml Workshop*, p. 13, 2015.

[6] "Koka programming language." `https://www.microsoft.com/en-us/research/project/koka/`. Accessed: 2019-10-17.

[7] D. Leijen, "Algebraic effects for functional programming," tech. rep., Technical Report. 15 pages. https://www. microsoft. com/en-us/research . . . , 2016.

[8] "Links programming language." `https://links-lang.org/`. Accessed: 2019-10-17.

[9] D. Hillerström and S. Lindley, "Liberating effects with rows and handlers," in *Proceedings of the 1st International Workshop on Type-Driven Development*, TyDe 2016, (New York, NY, USA), pp. 15–27, ACM, 2016.

[10] "Eff programming language implementations." `https://github.com/matijapretnar/eff/releases`. Accessed: 2019-10-24.