Géza Csenger

Compiling algebraic effect handlers

Computer Science Tripos – Part II

Homerton College

2020

Declaration of Originality

I, Géza Csenger of Homerton College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

I, Géza Csenger of Homerton College, am content for my dissertation to be made available to the students and staff of the University.

Signed

Date

Acknowledgements

Hereby, I would like to say thanks to:

- My project supervisor, **Prof. Alan Mycroft**, for his support and guidance
- My Director of Studies, **Dr John Fawcett**, for a *lot* of things
- Special thanks goes to Lili Janzer for proofreading this work as an *actual* Part IB student
- Friends and family without whom completing this work would have been impossible

Proforma

Candidate number	2375C
Project title	Compiling algebraic effect handlers
Examination	Computer Science Tripos – Part II, 2020
Word Count	11033[1]
Lines of Code	6158[2]
Page Count	40
Project Originator	The author
Supervisor	Professor Alan Mycroft

Original aims of the Project The project had four core aims: to write an interpreter for Eff, to design a bytecode for Eff, to write a compiler from Eff to this bytecode and to implement a virtual machine interpreting the bytecode. Additional goals were to implement various optimisations for Eff, such as tail resumption optimisation or minimising the copy overhead from the use of continuations.

Work completed A syntax-level Eff interpreter was implemented. A bytecode for Eff (SHADEcode) was designed. A compiler from Eff to SHADEcode was implemented as well as a virtual machine (SHADE) interpreting SHADEcode. As far as I know, SHADE and SHADEcode are the first virtual machine and bytecode designed *specifically* for Eff. In the Evaluation chapter I compare the performance of my Eff interpreter with the current official Eff interpreter and the performance of the SHADE virtual machine with the performance of Multicore OCaml (another compiled language with similar features to Eff).

Special difficulties None.

[1] Obtained using `https://app.uio.no/ifi/texcount/`

[2] Obtained using Visual Studio Code's VS Code Counter extension

Contents

List of Figures

List of Tables

Listings

1 | Introduction

This dissertation is concerned with building a compiler for the programming language *Eff*. Eff is a research functional programming language based on *algebraic effect handlers*. I describe how a language like Eff can be interpreted and how it can be compiled to bytecode.

The main contribution of this dissertation is the bytecode designed for Eff and the corresponding abstract machine capable of interpreting this code. I examine how my implementation of this abstract machine compares with existing solutions.

1.1 Motivation

The motivations for this project are:

- **Novel programming language feature.** As algebraic effect handlers are an esoteric research programming language feature not many implementations exist of the languages which support them. Writing a compiler dealing with such a new language feature seemed to be a good challenge.
- **Compiling Eff.** Nearly all current programming languages supporting algebraic effect handlers are interpreted using a high-level CEK machine or embedded into other programming languages as libraries.

  While some solutions do compile algebraic effect handlers to bytecode (e.g., the multicore branch of OCaml), the way this is done remains largely undocumented.
- **Performance.** Whereas the theory behind algebraic effect handlers is well understood, only a few attempts were made at evaluating the performance of different solutions.
- **Systems programming.** As algebraic effect handlers allow for pausing and restarting computations, they also provide a way to express effectful computations in a direct style. This can make functional systems programming less obscure.
- **Personal interest.** I am hoping that diving deeper in the design of programming languages and understanding the requirements runtime systems need to possess to interpret algebraic effect handlers will add value to my carreer on the longer term too.

1.2 Work carried out

I have implemented a lexer and a parser for Eff as well as a high-level CEK-like interpreter working with a tree-like representation of an Eff program. I designed a byte-code for Eff inspired by the OCaml byte-code and a compiler to this byte-code. I designed an abstract machine as well, which ended up being similar to an SECD machine. I implemented this machine and compared its performance to my CEK implementation and to other solutions.

I had to make some simplifications during this process. For instance, I dropped support for pattern matching as well as support for some syntactic sugar and type inference, as these were not the focus of my project.

1.3 History and related work

Roughly speaking, *computational effects* are effects like I/O, state, non-determinism and various forms of jumps [16]. The way that these effects interact with each other and how we can safely deal with them has always formed a great research interest, as the improper use of side-effects is often responsible for software bugs.

Algebraic effects and their handlers have a precise theoretical foundation and can be used to express effects like the above conveniently in a pure functional setting. Their study began in the context of the computational lambda calculus.

1.3.1 Theory: from maths to the programming language Eff

In 1989, Moggi [13, 14] laid down the theoretical foundations needed for a unified category-theoretic semantics for computational effects in the computational lambda-calculus.

In 2003, Plotkin and Power [17] used the idea that many of the computational effects can be naturally described by algebraic theories and extended the computational lambda calculus with *effect constructors*. An *algebraic effect* is a computational effect that can be described with an algebraic theory.

It turns out that effects like I/O, state and non-determinism are all algebraic. However, perhaps not surprisingly, not all effects are. Notably, the *handle* or *catch* operations—used to handle exceptions in most conventional programming languages—are a good example, these cannot be expressed with algebraic theories. In fact, these effects are exactly the duals of algebraic effects. This means that if we think about performing an effect as *producing* an effect, then the handle or catch operations can be seen as the *consumers* of such effects. The literature commonly talks about *effect constructors* and *effect destructors* in this respect.

In 2009, Plotkin and Pretnar [18] generalised the notion of exception handlers and introduced the idea of handlers for algebraic effects. With this, they established a duality between algebraic effects and their handlers. They also list a few surprising examples of algebraic effect handlers, such as stream redirection, timeout and rollback.

In 2015, Bauer and Pretnar [6] created the programming language Eff, one of the first programming languages supporting algebraic effects and handlers at the language level.

1.3.2 Practice: effect handlers in the real world

The power of algebraic effect handlers is gaining recognition in the wider programming language community. Notably, a branch of OCaml (Multicore OCaml) is being built around continuations and effect handlers.

Dolan et al. [8] explored the application space of effect handlers and demonstrated how they can ease functional concurrent system programming by implementing an asynchronous I/O library which can be used in direct style.

Effect handlers can also ease the design of runtime systems. By exposing continuations one can implement concurrency models in the user level as libraries. This gives us the opportunity to design more efficient and simpler runtime systems. Programmers benefit from this by being able to swap between concurrency models, which would be impossible if we baked a concurrency model into the runtime system.

2 | Preparation

This chapter contains an introduction to the programming language Eff. It also discusses continuations and their relationships with two abstract machines (CEK and SECD machines) which is *necessary* to understand in order to appreciate what follows in this dissertation.

It took me more than a month to understand the concepts described here and the relationships between them. Therefore this chapter is witten in a tutorial style for the benefit of the next person attempting a similar dissertation.

2.1 A quick Eff tutorial

Eff is a programming language based on algebraic effect handlers. Eff resembles OCaml in that if we removed OCaml's [3] side-effecting operations and re-created them by *simulating* them with a more general notion of exception (specifically, resumable exceptions) then we would get Eff as the result. In Eff, *effect* is used to mean resumable exception. This tutorial will show how printing can be understood as an effect and how it can be *simulated* in a pure functional way using effect handlers.

To implement effects and effect handlers Eff uses the following keywords which are not part of OCaml at the time of writing[1]: `effect`, `perform`, `handler`, `continue`, `with-handle` and `finally`. What follows is a brief introduction of these features by using the unmissable Hello World program.

2.1.1 Effects

Effects lie in the heart of Eff (hence the name). A programmer can define effects using the `effect` keyword by specifying the effect's name (which must be capitalised just like OCaml variant tags) and the type of the effect. An effect definition can be seen on line 1 of Listing 2.1. Effects always have a type of $A \twoheadrightarrow B$ for some types A and B.

In the introduction of this tutorial I said that effects are resumable exceptions. The careful reader might notice that the type $A \twoheadrightarrow B$ is similar to the function type $A \to B$ but it is quite different from usual types of exceptions. For instance, in SML [12] one would expect to see declarations representing errors like `exception Error of string` (an error which can carry an error message with itself could be declared like this) but would not expect to talk about the *return type* of an exception, as these can never return in SML. However, in Eff, due to the resumable nature of effects this makes sense.

2.1.2 Performing effects

Effects behave a bit like functions as we could see from their types above. Once an effect of type $A \twoheadrightarrow B$ is declared we can invoke it using the `perform` keyword and by providing an argument of type A. The type of the resulting expression is B.

[1] Although a branch of OCaml, namely Multicore OCaml is just getting merged in the main OCaml branch so we might see some of these keywords in OCaml in the future.

Listing 2.1: Printing is a `string -->> unit` effect

```
1 effect Print : string -->> unit;;
2 ...
3 let () = perform (Print "Hello World");;
4 ...
```

The `perform` keyword is similar to `raise` in OCaml or to `throw` in Java in that when we perform an effect the control will be given to an effect handler (this would be an exception handler in OCaml or Java) or the program will terminate with an exception if no handler can handle the effect.

Effects are not values in the Eff language. That is, `Print "Hello World"` would be meaningless in itself, just like the exception `Failure "hd"` would be meaningless in OCaml if we did not raise it. The situation is similar here but effects are *performed* and not raised.

In Java, if we throw an exception without introducing a try-catch block handling it, the exception will rise to the toplevel and will cause our program to crash. This is exactly the result Listing 2.1 would produce as we did not surround line 3 with the try-catch block equivalent of Eff, which is the *with-handle block*. However, to do this, we need to be able to declare *effect handlers* first.

2.1.3 Effect handlers

Effect handlers give meaning to effects: this is what programmers can use to specify what they mean by an effect, such as `Print` in Listing 2.1. Using handlers we can make the code evaluate properly. Right now it always crashes on line 3 and does not continue executing the rest of the program. This is hardly what we desire from `Print`.

Before we do this, we must think about how we would *simulate printing* in a pure functional language. One way of doing this would be to implement all functions of type $A \to B$ to return something of type $B \times$ `string` instead (i.e., the new function would return a pair; its original return value *together* with the message it would print). Such a transformation would be a tedious manual task to implement for *all* our functions, not to mention how we would need to unbox the actual result of a function from a tuple every time we wished to access it. Fortunately, handlers happen to alleviate this problem.

Handlers are first-class citizens (values) in Eff and they show some similarities with SML's `case` statement. A handler can be declared with the `handler` keyword and by specifying so-called *cases*:

Listing 2.2: A handler definition consisting of 2 cases: a value case and an effect case

```
1 let prepending_print_handler = handler
2 | val res -> (res, "")
3 | effect (Print msg) k ->
4     let (res, out) = continue k () in
5     (res, msg ^ out)
```

To complete our Hello World example we must be able to *apply* this handler. This is done using a *with-handle block* as illustrated in Listing 2.3.

Listing 2.3: Hello World in Eff

```
1 with prepending_print_handler handle
2     perform (Print "Hello ");
3     perform (Print "World!");
4     ()
```

The code above evaluates to the pair `((), "Hello World!")`. The handler on Listing 2.2 implements printing. It uses two types of cases as described below.

A *value case* has the form `val` $x \to c$. This is a case that describes how to handle the *return value* of a computation. It takes the return value of a computation enclosed by a with-handle block, binds it to the identifier x and performs the computation c. In Listing 2.2 we are simply saying that if the computation handled by this handler has finished computing without invoking any effects, then we are returning a pair containing the result of the computation and the empty string (this reflects that nothing was printed).

An *effect case* is a generalisation of SML's `handle` or Java's `catch`. It has the form `effect op` $e\ k \to c$, where the `effect` keyword plays the rôle of Java's catch, `op` is an effect name, e is an expression, k is an identifier to be used as the name for a continuation and c is a computation. The continuation k can then be *resumed* with the `continue` keyword and with an argument to the continuation. Note that when an effect of type $A \twoheadrightarrow B$ is handled, we get *read access* to its argument e of type A and we can resume its continuation k if we provide k a value of type B (this is why we resume the continuation by giving it a `()` value in Listing 2.2—`Print` is of type `string` $\twoheadrightarrow$ `unit`).

As handlers are values, they have their own type of the form $A \Rightarrow B$. Handler types are similar to the $\to$ function types but they work on computations rather than on values. We can see why this is so if we think about what `prepending_print_handler` is doing in our example. Used with a with-handle block it transforms any computation of type A to one of the type $A \times$ `string`. One might think of handlers as transformations on computations. The handler on Listing 2.2 is of type $\alpha \Rightarrow \alpha \times$ `string`.

The finally case

There is an extra case we did not mention, the *finally case*. A finally case is invoked after the computation in a with-handle block has finished evaluating and after all other handler cases have finished evaluating.

Finally cases are just syntactic sugar for let wrappers around with-handle blocks. That is, **finally** *res* $\to$ *fin* does the same as **let** *res* $\leftarrow$ **with** *h* **handle** *c* **in** *fin*. Finally rules were introduced to avoid having to use such inconvenient let wrappers [6]. To see the use case of this feature, compare the following two code snippets in Listing 2.4 and Listing 2.5, where the result of a physics simulation is returned in Kelvin but we wish to convert this result to Fahrenheit every time we want to display it on some user interface.

Instead of using a let-wrapper every time to convert to Fahrenheit we could use the same handler h but extend it with a finally rule that does this conversion and hence avoids code duplication.

A handler can have any number of effect cases (even zero) but at most one finally case and at most one value case. Value cases and finally cases are optional. When they are avoided they are assumed to be identities (i.e., `val x -> x` or `finally x -> x` respectively).

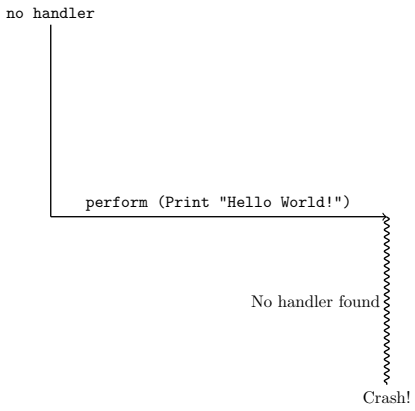
Listing 2.4: Using a handler without finally

```
let temp_in_fahrenheit =
  (* t is in Kelvins *)
  let t = with h handle ...
  in t * 1.8 - 459.67;;
```

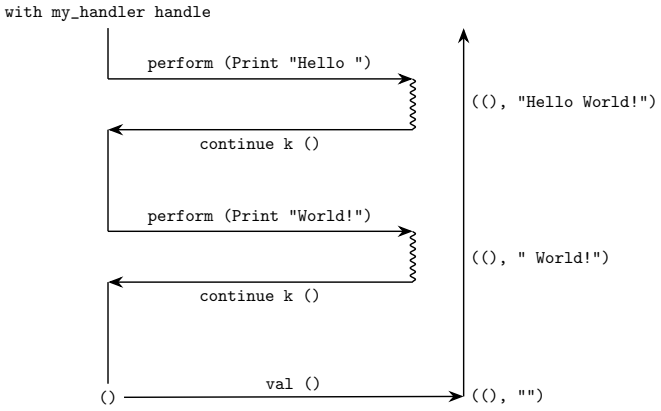
Listing 2.5: Handler with finally

```
let h' = handler
(* ... same as h ... *)
| finally t -> t * 1.8 - 459.67
;;
```

Illustration. Imagining how the execution happens in the Hello World example is not trivial. The aim of the following illustration is to help with this.



(a) Crash without a handler.



(b) Effects and resumed continuations.

Figure 2.1: The tutorial illustrated

2.2 Continuations and control operators

Although continuations came up briefly in the previous section I did not explain what they are in detail. This section is devoted to this because continuations are playing a crucial rôle in Eff and in what follows in this dissertation.

2.2.1 CPS in Scheme

Continuations can represent an arbitrary program point in the execution of a program and thus one can say that continuations are always there, whenever a program executes. But where? A programming style called *continuation passing style* (CPS) is a style of programming where every function of n arguments is rewritten to an extended version taking $n + 1$ arguments, the last of which is commonly named k and represents a continuation.

Consider the Scheme code snippet in Listing 2.6 which uses this style to redefine the built-in binary addition and multiplication functions of Scheme as ternary functions `+k` and `*k` with an extra argument `k`. Also note how the expression `(1 + 2)*(3 + 4)` is written in this style.

Listing 2.6: $(1+2)*(3+4)$ in CPS

```
1 (define return (lambda x x))
2
3 (define (+k a b k) (k (+ a b)))
4 (define (*k a b k) (k (* a b)))
5
6 #| Compute (1 + 2) * (3 + 4) |#
7 (+k 1 2 (lambda (sum1)
8   (+k 3 4 (lambda (sum2)
9     (*k sum1 sum2 return)))))
```

We see that continuation passing style makes the order of the operations explicit as well as it makes it obvious *what* the continuation is. However, writing code in this style is a bit troublesome and hence CPS is mostly used as an intermediate representation of programs in compilers—not something humans interact with.

2.2.2 Scheme's call/cc

Scheme is interesting from the point of view of continuations because it has the `call-with-current-continuation` built-in operation which is conventionally referred to as `call/cc`[2]. *The call/cc operator takes a single argument. This argument is a* function *which itself takes a* continuation *as an argument.*

If one wished to use continuations as first-class values in a Scheme program one would not have to write everything in CPS like in Listing 2.6. A similar piece of code can be written with call/cc:

Listing 2.7: Continuations with `call/cc` are not delimited

```
1 (define call/cc call-with-current-continuation)
2
3 #| Compute (1 + 2) * (3 + 4) |#
4 (* (+ 3 4) (call/cc (lambda (k) (k (+ 1 2)))))
5
6 #| The following line evaluates to 21 too |#
7 (* (+ 3 4) (call/cc (lambda (k) (+ 100 (k (+ 1 2))))))
```

Looking at line 7 we see that the continuation here returns to the toplevel. Call/cc captures the continuation of the *whole* program and *does not* return to the program point the continuation was called from. This is different from how this is done in Eff.

2.2.3 Delimited continuations

Eff uses delimited continuations. These continuations *do not* represent the rest of the whole program. They represent the continuation *of the computation enclosed by a with-handle block.*

[2] A similarly obscure control operator is Peter Landin's J operator which predates `call/cc` by almost a decade. It was discovered shortly after Peter Landin described SECD machines for the first time.

Listing 2.8: In Eff continuations behave like functions

```
let x =
  with prepending_handler handle
    perform (Print "Hello");
    perform (Print "World");
    ()
in 42;;
```

In Listing 2.8 `x` takes the value `((), "Hello World")` rather than `42` which one would expect with Scheme's `call/cc`. The fact that continuations are delimited also makes it possible to return to the program point a continuation was called from. This also allows us to resume continuations in Eff more than once if necessary. Hence, in Eff we can think about delimited continuations simply as functions.

2.3 Eff formally

2.3.1 Syntax

In the abstract syntax of Eff there is a distinction between pure expressions and computations which are possibly effectful[3].

Effect declarations

$$\textbf{effect}\ E : A \twoheadrightarrow B$$

Expressions

$$e ::= x \mid \textbf{true} \mid \textbf{false} \mid () \mid (e_1, e_2) \mid \textbf{Left}\ e \mid \textbf{Right}\ e \mid \textbf{fun}\ x \mapsto c \mid h$$

Handlers

$$h ::= \textbf{handler}\ \textbf{val}\ x \mapsto c_v \parallel \overline{\textbf{effect}\ E_i\ x\ k \to c_{\mathtt{op}_i}} \parallel \textbf{finally}\ x \mapsto c_f$$

Computations

$$\begin{aligned} c ::= {} & \textbf{val}\ e \mid \textbf{absurd}\ e \mid \textbf{let}\ x = c_1\ \textbf{in}\ c_2 \mid \textbf{let rec}\ f\ x = c_1\ \textbf{in}\ c_2 \mid e_1\ e_2 \mid \\ & \textbf{if}\ e\ \textbf{then}\ c_1\ \textbf{else}\ c_2 \mid \textbf{match}\ e\ \textbf{with}\ \textbf{Left}\ x \mapsto c_l \parallel \textbf{Right}\ y \mapsto c_r \mid \\ & \textbf{match}\ e\ \textbf{with}\ (f, s) \mapsto c \mid \textbf{perform}\ (E\ e) \mid \textbf{with}\ e\ \textbf{handle}\ c \end{aligned}$$

Figure 2.2: Abstract syntax of Eff

Most terms should be familiar to the reader by now. However, there are some differences between the concrete and the abstract syntax of Eff. Notably, the use of the *val* keyword is omitted many times from the concrete syntax because for programmers it might be tiresome if they have to

[3]The reader familiar with monads might discover some similarities between *val* and *return* as well as similarities between monadic type constructors and computations.

keep track whether they are writing an expression or a computation at the moment. The *continue* keyword is missing from the abstract syntax too. However, our ability to resume computations did not vanish. As we saw in the previous section, delimited continuations are just simply functions reifying a continuation. Hence we can handle resumptions as function application. Anyhow, to ease the understanding of the semantic rules I use the notation $(y.c)$ for delimited continuations which resume the computation c after receiving y.

2.3.2 Semantics

This section presents the small step operational semantics of Eff. The spotlight is on possibly effectful computations and on their reduction rules given by the relation $\rightsquigarrow$. I focus on the Eff specific transition steps here (see Figure 2.3). The other transition rules are standard and can be found in Appendix A.

$$\frac{c_1 \rightsquigarrow c_1'}{\textbf{let } x = c_1 \textbf{ in } c_2 \rightsquigarrow \textbf{let } x = c_1' \textbf{ in } c_2} \text{ (Let-Step)}$$

$$\frac{}{\textbf{let } x = (\textbf{val } e) \textbf{ in } c \rightsquigarrow c[e/x]} \text{ (Let-Val)}$$

$$\frac{}{\textbf{let } x = E(e, y.c_1) \textbf{ in } c_2 \rightsquigarrow E(e, y.\textbf{let } x = c_1 \textbf{ in } c_2)} \text{ (Let-Effect)}$$

$$\frac{\kappa \text{ is current delimited continuation}}{\textbf{perform } (E\ e) \rightsquigarrow E(e, \kappa)} \text{ (Perform)}$$

$$\frac{c \rightsquigarrow c'}{\textbf{with } e \textbf{ handle } c \rightsquigarrow \textbf{with } e \textbf{ handle } c'} \text{ (Handle-Step)}$$

$$\frac{h \stackrel{\text{def}}{=} (\textbf{handler val } x \mapsto c_v \ \parallel \ \dots)}{\textbf{with } h \textbf{ handle } (\textbf{val } e) \rightsquigarrow c_v[e/x]} \text{ (Handle-Val)}$$

$$\frac{h \stackrel{\text{def}}{=} (\textbf{handler } \dots \overline{\parallel \ \textbf{effect } E_i\ x\ k \mapsto c_i} \ \parallel \ \dots) \quad \exists i.E_i = E}{\textbf{with } h \textbf{ handle } E(e, y.c) \rightsquigarrow c_i[e/x, (y.c)/k]} \text{ (Handle-Eff-Match)}$$

$$\frac{h \stackrel{\text{def}}{=} (\textbf{handler } \dots \overline{\parallel \ \textbf{effect } E_i\ x\ k \mapsto c_i} \ \parallel \ \dots) \quad \forall i.E_i \neq E}{\textbf{with } h \textbf{ handle } E(e, y.c) \rightsquigarrow E(e, y.\textbf{with } h \textbf{ handle } c)} \text{ (Handle-Eff-Rise)}$$

Figure 2.3: Eff-specific rules of the small step operational semantics

The explanation of these rules is given by Table 2.1 so that the reader can quickly jump between the comments there and the rules on Figure 2.3.

Rule	Comment
Let-Val	If a computation *returns* an expression e, it is bound to the identifier x in the computation c.
Let-Effect	A computation c can also reduce to a *performed effect* $E(e, y.c_1)$, where e is the argument of the effect constructor and $(y.c_1)$ is a delimited continuation representing the program point c's execution was stopped at. This delimited continuation can be used later to resume c.
Perform	Whenever we perform an effect, a so-called *effect packet* (similar to *exception packets* in SML, see [15]) $E(e, \kappa)$ is created. Here, e is the argument to the effect and κ is the captured delimited continuation which is determined by the with-handle blocks surrounding this computation. Note, that because it is not guaranteed that there are such with-handle blocks around a **perform** call, creating such an effect package might not always be possible.
Handle-Val	If a computation in a with-handle block with handler h reduces to **val** e, then the *value case* of h (i.e., **val** $x \mapsto c_v$) is invoked by substituting e for x in c_v.
Handle-Eff-Match	This rule tells us what happens when there is a *matching effect case* in the current handler. The argument of the effect is bound to x and the delimited continuation $(y.c)$ is bound to k.
Handle-Eff-Rise	This rule tells us what happens when the next handler *cannot* handle the effect which was performed. In this case the effect packet is modified which is quite important. The *body* of the delimited continuation is surrounded by the current non-matching with-handle block. This achieves the effect that when the continuation is resumed we also *restore all the handlers we escaped* when we were "searching" for a matching handler. This is a property of the semantics which we will have to be very careful about when implementing the CEK interpreter and the SHADE VM in the next chapter.

Table 2.1: Explanation of the Eff specific rules of the small step semantics

2.3.3 Types

The types of Eff are as follows:

$$A ::= b \mid \texttt{bool} \mid \texttt{unit} \mid \texttt{empty} \mid A * B \mid A + B \mid A \to B \mid A \Rightarrow B$$

$$E_i ::= A_i \twoheadrightarrow B_i$$

where b stands for built-in types (e.g., my implementation includes built-in types such as ints, reals and strings) which are not particularly interesting here. In the following section we assume the existence of a set Σ_E containing all *effect signatures* of the form $E_i : A_i \twoheadrightarrow B_i$.

Type checking

Again, I omit standard typing rules and focus on the Eff specific ones. For the sake of completeness Appendix A contains the other rules. There are two separate typing relations denoted $\vdash_e$ and $\vdash_c$ for expressions and computations respectively. Type environments are denoted Γ and are maps from variable names to types.

$$\frac{\Gamma \vdash_e e : A}{\Gamma \vdash_c \textbf{val}\ e : A}\ (\text{T-Val}) \qquad \frac{\Gamma \vdash_e e : A \Rightarrow B \qquad \Gamma \vdash_c c : A}{\textbf{with}\ e\ \textbf{handle}\ c : B}\ (\text{T-WHandle})$$

$$\frac{E : A \twoheadrightarrow B \in \Sigma_E \qquad \Gamma \vdash_e e : A}{\textbf{perform}\ (E\ e) : B}\ (\text{T-Perform}) \qquad \frac{\Gamma \vdash_e e : \texttt{empty}}{\Gamma \vdash_c \textbf{absurd}\ e : A}\ (\text{T-Absurd})$$

$$\frac{x : A, \Gamma \vdash_c c_v : C \qquad \forall i.\ e_i : A_i, k_i : B_i, \Gamma \vdash_c c_i : C \qquad x : C, \Gamma \vdash_c c_f : D}{\Gamma \vdash_e h : A \Rightarrow D}\ (\text{T-Handler})$$

$$\text{where } h \stackrel{\text{def}}{=} (\textbf{handler}\ \textbf{val}\ x \mapsto c_v \parallel \overline{\textbf{effect}\ E_i\ e_i\ k_i \mapsto c_i} \parallel \textbf{finally}\ x \mapsto c_f) \text{ and } \forall i.\ E_i : A_i \twoheadrightarrow B_i \in \Sigma_E$$

Figure 2.4: Type checking for Eff-specific expressions and computations

Again, explanation of these rules is given in Table 2.2 in a similar way to how it was done for the semantics.

The attentive reader might have wondered what **absurd** and the `empty` type are good for. In the tutorial we got away with a slight lie which we will make amends for here. We said that in the absence of suitable handlers an effect will behave like an exception (remember Listing 2.1). This is not quite true when typing is introduced, as a computation like

```
effect Error of string -->> unit;;
if true then (perform (Error "Lie detected")) else 1337
```

will not be typeable with our rules (there is a type mismatch between `unit` and `int`), although in OCaml or SML one can write

```
exception Error of string;;
if true then (raise (Error "Lie detected!")) else 1337
```

Rule	Comment
T-Val	This rule is standard. The expression e and the computation **val** e are given the same type.
T-Perform	This rule is similar to function application for effects.
T-Handler	The rule for typing handlers might be the scariest one. We are saying here that the type of the value- and effect cases must agree (this is type C in the rule) and that the finally case which takes something of type C and gives something of type D transform a value of type C into something of type D. The resulting handler type $A \Rightarrow D$ reflects this by saying that a computation of type A will be turned into one of type D by such a handler, when it is applied.
T-WHandle	As applying handlers happens with with-handle blocks this rule is similar to the function application typing too.

Table 2.2: Explanation of the Eff specific typing rules

without any problems. We would like to use effects as exceptions—it would be weird not to, as they are a generalisation of them. This is why we need **absurd** and the `empty` type. We can now say

Listing 2.9: Absurd and empty with exceptions

```
1 effect Error : string -->> empty;;
2 if true then absurd (raise (Error "Lie detected!")) else 1337;;
```

and this will type check if we use (T-Absurd). Note also that the only way one can introduce the `empty` type is by declaring an effect that has an `empty` output.

The distinction between pure expressions and effectful computations allows for the use of an *effect system* with these rules. For instance, if one were to extend these rules with *row-typing* one could keep track of what kind of side effecting computations occur in different parts of the program. This can aid reasoning about programs, help us to discover more optimisations by using type and effect information and to ensure that programs have certain safety properties. These are nice to have features and are outside of the scope of this dissertation.

2.4 Abstract machines by example

A purist would look at rule (Perform) in Figure 2.3 and would think that the phrasing of the hypothesis is troublesome. I actually think that this ambiguity reflects well the challenge in this project. Consider for instance implementing an interpreter on the source terms of a language X. One's first instinct is that this is a trivial task: all we need to do is to write a recursive evaluator function, something of this sort:

Listing 2.10: A naïve recursive evaluator function might look like this

```
1 let rec eval = function
2 | Plus(e1, e2) -> eval(e1) + eval(e2)
3 | ...
```

Indeed, this works for most programming languages (ones that do not have control operators like continuations) and this was my first attempt when writing a 0^{th} prototype in the very early stages of the project. I started to struggle when I reached the point I had to deal with continuations, as I needed the ability to access κ any time in the evaluation of a program. But evaluators of this kind do not lend themselves easily to such maneuvers.

Hence this section is concerned with two types of abstract machines which are capable of doing what is described above. I will illustrate how they do this on a toy language and by the end of this section we will be able to discover a correspondence between the *runtime representation* of continuations in these machines and the CPS Scheme example from Section 2.2.

These two machines are CEK and SECD machines. They both interpret source terms directly and this makes it easy to implement them (a CEK implementation of Eff is discussed in the Implementation chaper).

Consider the following Toy Language (TL), which is the untyped lambda calculus with natural numbers as values and with built-in operations like addition and multiplication:

$$t ::= x \mid n \mid \lambda x.t \mid t_0\ t_1 \mid t_0 + t_1 \mid t_0 * t_1.$$

Our dummy example from Section 2.2 in TL will be:

$$\texttt{plus} = \lambda x.\lambda y.\ x + y$$

$$\texttt{times} = \lambda x.\lambda y.\ x * y$$

$$\texttt{times}\ (\texttt{plus}\ 1\ 2)\ (\texttt{plus}\ 3\ 4).$$

What follows is a brief description of CEK and SECD machines for TL illustrated on this example.

2.4.1 CEK machines

As the name suggests, the state of CEK machines consists of three components: *control* (C), *environment* (E) and *kontinuation* (K). Control is simply the program to be evaluated in its abstract syntax tree form. Environment is a mapping from variable names to values (in our case, the natural numbers) and kontinuation is a *stack of closures* representing the rest of the program to be interpreted at any given time. Hence a configuration of the machine is characterised by a three-tuple $\langle C, E, K\rangle$.

We see at $(*)$ in Figure 2.5 that whenever we reduce a term to a value v, a continuation $(y.t)$ is popped off the K stack, v is bound to the variable y in E' (this is the environment we need to restore to be able to interpret t in the same context we stopped evaluating its parent expression) and then t is resumed. We need to use *two* types of continuations for TL as it has *binary* operators $+$ and $*$. The name `CONT1` suggests that one of the operands is reduced to a value and one remains to be reduced before the actual operation can be performed. `CONT2` is used to represent the state when both of the operands are values and the operation `op` can be performed (be that the built-in `add` or `mult`).

We see that a **perform** operation would be easier to implement in this representation than in the naïve way. Furthermore, we should notice that a CPS transformation is done on the terms "on the fly" during the evaluation and we benefit from this because this avoids us (programmers) having to make the continuations explicit in our programs by writing them in CPS.

Initialisation:

$$t \to \langle t, \{\}, nil \rangle$$

Transition rules:

$$\langle x, E, K \rangle \to \langle E(x), E, K \rangle$$

$$\langle v, E, (y.t, E') :: K \rangle \to \langle t, E'[y \mapsto v], K \rangle \quad (*)$$

$$\langle (\lambda y.t_1)\ t_2, E, K \rangle \to \langle t_2, E, (y.t_1, E) :: K \rangle$$

$$\langle t_1 + t_2, E, K \rangle \to \langle \texttt{add}(t_1, t_2), E, K \rangle$$

$$\langle t_1 * t_2, E, K \rangle \to \langle \texttt{mult}(t_1, t_2), E, K \rangle$$

$$\langle \texttt{op}(t_1, t_2), E, K \rangle \to \langle t_1, E, (y.\texttt{CONT1}(y, \texttt{op}, t_2), E) :: K \rangle \text{ for a } \textit{fresh } y$$

$$\langle \texttt{CONT1}(x, \texttt{op}, t_2), E, K \rangle \to \langle t_2, E, (y.\texttt{CONT2}(x, \texttt{op}, y), E) :: K \rangle \text{ for a } \textit{fresh } y$$

$$\langle \texttt{CONT2}(x, \texttt{op}, y), E, K \rangle \to \langle v, E, K \rangle \text{ where } v = \texttt{op}(x, y)$$

Termination:

$$\langle v, E, nil \rangle \to v$$

Figure 2.5: CEK machine transition rules for TL

2.4.2 The SECD machine

SECD machines originate from Landin's famous *The mechanical evaluation of expressions* [10]. Their state is characterised by four components, namely: a *stack* (S), an *environment* (E), a *control* (C) and a *dump* (D). The first three components are standard. Here the rôle of dump is similar to the rôle of K in the CEK machine. It is a list of three tuples, each three tuple consisting of the other three components (i.e., it has the form (S, E, C)). The dump embodies a list of saved execution contexts we can return to if we wish.

We see that although this machine works on abstract syntax trees, it is still a bit lower level than the CEK machine in that it makes a stack explicit and the way it processes (disassembles) the tree is very similar to compiling. In control, we end up with a list of smaller trees and SECD instructions. With just another step we can turn this into a stack machine and this is what I will do in the Implementation chapter.

Importance Figure 2.7 sums up why this section is important. It is an illustration of how different evaluators represent continuations. Note that the representation of a continuation in Figure 2.7b looks very similar to an instruction stream. It is as if CEK did a CPS translation on the fly while SECD did both a CPS and a "sort of compilation" while interpreting a program—this is a precious insight to possess when our task is to compile a language like Eff!

Initialisation:

$$t \to \langle nil, \{\}, [t], nil \rangle$$

Transition rules:

$$\langle S, E, x :: C, D \rangle \to \langle E(x) :: S, E, C, D \rangle$$

$$\langle v :: S, E, nil, (S', E', C') :: D \rangle \to \langle v :: S', E', C', D \rangle$$

$$\langle S, E, (t_1 \ \texttt{op} \ t_2 :: C, D \rangle \to \langle S, E, t_1 :: t_2 :: \texttt{op} :: C, D \rangle$$

$$\langle S, E, (t_1 \ t_2) :: C, D \rangle \to \langle S, E, t_2 :: t_1 :: \texttt{ap} :: C, D \rangle$$

$$\langle (y, E', t) :: v :: S, E, \texttt{ap} :: C, D \rangle \to \langle nil, E'[y \mapsto v], [t], (S, E, C) :: D \rangle$$

$$\langle v_2 :: v_1 :: S, E, \texttt{op} :: C, D \rangle \to \langle v :: S, E, C, D \rangle \text{ for } v = \texttt{op}(v_1, v_2)$$

Termination:

$$\langle v :: S, E, nil, nil \rangle \to v$$

Figure 2.6: SECD machine transition rules for TL

2.5 Software engineering

2.5.1 Licensing

It is important to mention that the project fulfills licensing criteria. The dependencies of this project are listed below with their licenses:

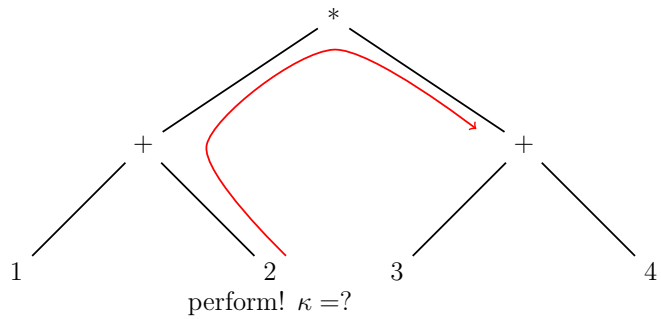
- The OCaml toplevel, version 4.06.0, GNU LGPL version 2.1
- *menhir* parser generator, version 20190924, GNU GPL version 2
- *Alcotest*, version 0.8.5, ISC license
- *OCaml Package Manager*, version 2.0.4, GNU LGPL version 2.1
- *Core* standard library by Jane Street, v0.11.3, MIT license

To the best of my knowledge these licenses allow the way I am using the software mentioned above.

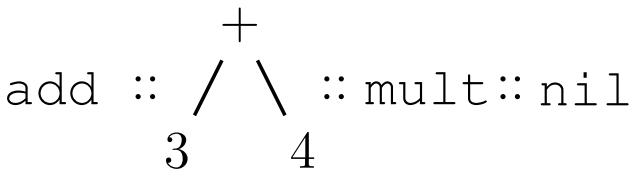
2.5.2 Starting point

The starting point of my project did not change since the writing of the project proposal (see Appendix E):

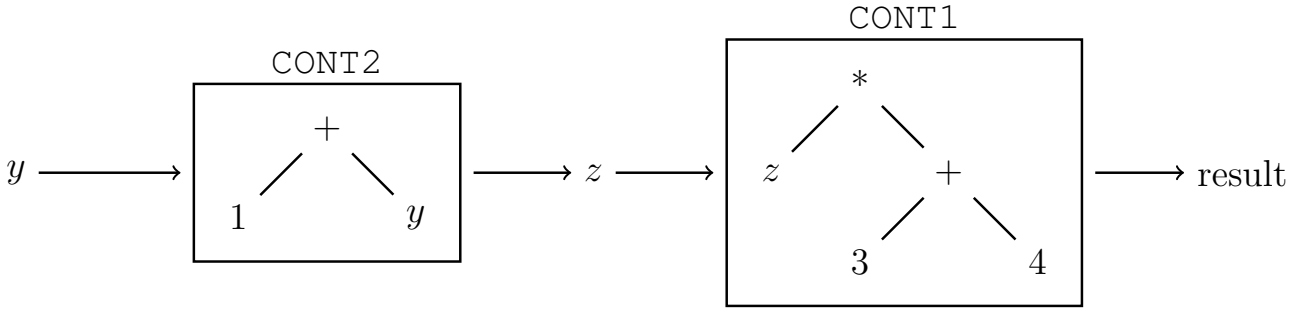
- I read briefly about Eff and algebraic effect handlers before starting my project.
- I used SML and OCaml before to implement simple interpreters for toy programming languages and to experiment with functional programming. This is my first big project where I use functional programming.
- I also had a look at the OCaml ecosystem to check if it has a stable build system, if testing frameworks are available and whether lexer and parser generators exist.



(a) *Naïve approach.* Imagine that a **perform** operation is issued at position 2 in the tree. As the naïve approach only keeps track of the continuations implicitly we cannot represent the red arrow easily.



(b) *SECD machine.* The stack of the SECD machine is a representation of the red arrow from (a). However, SECD does not need to pass values around between closures like (c) does.



(c) *CEK machine.* The K component of the CEK machine represents continuations with closures the same way our continuations were nested in each other in the Scheme example.

Figure 2.7: Where is the continuation?

2.5.3 Development methodology

I chose *test-driven development* (TDD) [7] as my project's software development methodology. The essence of TDD is that programmers work in quick iterative cycles whereby they write failing tests first and then implement the next feature that makes the failing test cases pass. Always having an *automated* test suite at hand avoids trial and error testing which would be both error prone and time consuming. The reasons for choosing this methodology was three-fold.

Firstly, the nature of my project is such that the correctness of any code written is very well-defined as I can take the semantics of Eff as a baseline and given a piece of code it is always possible to tell whether it is interpreted correctly or not. This makes it possible to write tests upfront.

Second, the structure of my project imposed an order on the parts to be delivered. This is because every module of my project (apart from the front end) depends on a previous one and hence on its correctness. As TDD is also known as "a way of managing fear during programming" it came handy that I could write code with confidence knowing that the code I have written before is tested. Other advantages were the ability to notice quickly if a code change broke some already implemented feature and the ability to refactor code with confidence (which I had to do frequently—from summer internships I knew that my coding style is such that this will be unavoidable if I want to work with a maintainable codebase).

Third, my success criterion was a correct working implementation of a compiler, so tests were crucial to demonstrate that success criteria were fulfilled.

2.6 Summary

This chapter summarised the theoretical background needed to complete the project. It also shows the preparation I undertook in the software engineering aspects.

- Eff was introduced in a brief tutorial.
- I explained how continuations represent arbitrary program points in the execution of a program and how they can be used to implement advanced control flow.
- *Core Eff* was formally introduced.
- Two abstract machines (CEK and SECD machines) were introduced. These can implement the structured *non-local flow of control* continuations make possible. The key challenge these machines solve is that they can *save and restore* execution contexts in an appropriate way whenever non-local control flow has to occur.

  A CEK machine does this by keeping a stack of continuations (K), while the SECD machine does this by using a so-called dump (D). The importance of these machines will become clearer in the Implementation section where I describe the SHADE virtual machine for Eff which has similarities with both of these abstract machines.
- I stated my starting point and discussed the development methodology used to write the code and the testing strategy to assess its correctness.

3 | Implementation

This chapter is concerned with the implementation of a CEK interpreter, the design of a bytecode for Eff (SHADEcode), a virtual machine capable of interpreting this *linearised* bytecode (SHADE VM) and a compiler to this bytecode.

The chapter is split into two sections: the first part is concerned with the *theory implementation* of the CEK and SHADE machines while the second part is more practice oriented.

3.1 A CEK machine for Eff

My implementation uses a CEK interpreter inspired by Hillerström's machine for Links [9].

3.1.1 Hlosures

I talked a lot about closures in the Preparation chapter but the concepts discussed there did not involve any handlers. In functional languages closures are used to make it possible to handle functions as values. This is non-trivial because functions can have free variables with values dependent on the context. In Eff, handlers are similar in this respect as they too can have free variables.

Listing 3.1: A handler parameterised by a free variable n

```
1 let make_handler n =
2   let h = handler
3   | val x -> x + n
4   | effect (E x) k ->
5     (* body of effect case using `n` *)
6   | finally x ->
7     (* body of finally case using `n` *)
8 ;;
```

In Listing 3.1 the `make_handler` function returns a handler *parameterised* by `n`. We see that we need to remember the value of `n` at the time of the creation of each handler at runtime. This can be done by attaching the environment E to a handler H, similarly to how one would do this with closures and functions.

In the rest of this chapter I will refer to such structures as *hlosures* (handler closures) and will denote the hlosure of handler H in the context of environment E as $\mathcal{H}(H, E)$.

3.1.2 Hlosure frames

When an effect is raised in an Eff program, control is given to a matching effect case. Now the current delimited continuation must be determined as it might be used in the body of this effect case. The continuation is delimited by the with-handle block the *matching handler* is handling. In the Preparation chapter we saw that we had access to the current continuation at all times in a CEK machine, however, now we also need to decide which closures belong to which with-handle blocks in the K stack.

At runtime we always know about the current handler, so we could simply tag all closures in the K stack with the current handler at the time of the closure's creation. However, it is a much better

idea to simply have a separate K stack *per handler*, because that will make the *unwind* operation on the K component much more efficient (we will unwind handlers and not continuations one by one).

Listing 3.2: The structure of a CEK machine for Eff

```
1  type cek_id (** Identifiers *)
2  type cek_env (** Environments *)
3  type cek_control (** Control terms *)
4
5  (** Closures and hlosures *)
6  type cek_closure = cek_id * cek_env * cek_control
7  type cek_hlosure = cek_control * cek_env
8
9  (** Hlosure frames and the new K component *)
10 type cek_hlosure_frame = cek_hlosure * cek_kont list
11 type cek_k = cek_hlosure_frame list
12
13 (** CEK machine configuration *)
14 type cek_state = {
15   c : cek_control;
16   e : cek_env;
17   k : cek_k;
18 }
```

I will call such a per handler K stack a *hlosure frame*. Hlosure frames will be the new entries in the K stack of the CEK machine. They consist of a hlosure and a *continuation stack* as can be seen in Listing 3.2 (see the types `cek_k` and `cek_hlosure_frame` to understand the structure of this new CEK machine).

3.1.3 CEK abstract machine

Now that we know how we account for handlers and delimited continuations in the CEK machine we can turn our attention to the transition rules of a CEK machine interpreting Eff (Figure 3.1).

Notation

As expressions do not admit reduction we can simply interpret them in the context of an environment γ. For an expression e this is written $[\![e]\!]\gamma$ and the notation means that if e is a variable then its value is looked up from γ. The runtime representation of functions **fun** $x \to c$ happens with closures and they are written as tuples of the form $(x.c, \gamma)$. Hlosures are written as $\mathcal{H}(h, \gamma)$ and the runtime representation of recursive functions **rec** $f\ x = c$ happens with recursive closures $\mathcal{R}(f, x.c, \gamma)$ which differ from normal closures in that they also include the name of the function represented. The continuation stack in hlosure frames is written as $\mathcal{C}$. I also write $h \downarrow \texttt{eff}$ if handler h handles effect `eff` and $h \uparrow \texttt{eff}$ otherwise.

Function and continuation application

As we discussed in the Preparation chapter delimited continuations and functions behave very similarly and there was no need to distinguish them in the abstract syntax of Eff. However, the

Initialisation:

$$c \longrightarrow \langle c, \{\}, [] \rangle$$

Termination:

$$\langle \textbf{val}\ e, \gamma, [] \rangle \longrightarrow \text{halt with } [\![e]\!]\gamma$$

Closures, hlosures and resumption from the continuation stack:

$$\langle \textbf{val}\ h, \gamma, K \rangle \longrightarrow \langle \textbf{val}\ \mathcal{H}(h, \gamma), \gamma, K \rangle$$

$$\langle \textbf{val}\ (\textbf{fun}\ x \to c), \gamma, K \rangle \longrightarrow \langle \textbf{val}\ (x.c, \gamma), \gamma, K \rangle$$

$$\langle \textbf{val}\ e, \gamma, ((x.c, \gamma') :: \mathcal{C}, \mathcal{H}) :: K \rangle \longrightarrow \langle c, \gamma'[x \mapsto [\![e]\!]\gamma], (\mathcal{C}, \mathcal{H}) :: K \rangle$$

Function and continuation application:

$$\langle (\textbf{fun}\ x \to c)\ e, \gamma, K \rangle \longrightarrow \langle c, \gamma[x \mapsto [\![e]\!]\gamma], K \rangle$$

$$\langle (\mathcal{R}(f, x.c, \gamma')\ v, \gamma, K \rangle \longrightarrow \langle c, \gamma'[x \mapsto v, f \mapsto \mathcal{R}(f, x.c, \gamma')], K \rangle$$

$$\langle \kappa\ e, \gamma, K \rangle \longrightarrow \langle [\![e]\!]\gamma, \gamma, \kappa\ @\ K \rangle$$

Perform and with-handle:

$$\langle \textbf{perform}\ E(e, k), \gamma, K \rangle \longrightarrow \langle \textbf{perform}\ E(e, k), \gamma, K, [] \rangle_{\text{unwind}}$$

$$\langle \textbf{with}\ \mathcal{H}\ \textbf{handle}\ c, \gamma, K \rangle \longrightarrow \langle c, \gamma, ([], \mathcal{H}) :: K \rangle$$

Other standard rules: (see Appendix A)

Figure 3.1: A CEK machine interpreting Eff

runtime system must be aware of what is applied to an argument! Fortunately, we can always distinguish closures, recursive closures and continuations at runtime.

Function application or applying closures is standard. When we apply a recursive closure $\mathcal{R}(f, x.c, \gamma)$ we make sure that we bind f to $\mathcal{R}(f, x.c, \gamma)$, so that f can be recursively invoked in its function body. With the hlosure frame representation of K, applying a continuation κ is just the same as prepending it to K. I would like to raise attention to how *convenient* this representation is. When an effect behaves like an exception (i.e., when κ is not resumed) all the handlers up to the handler handling the exception are all in κ and are not in K anymore. However, when we do resume κ then all the handlers are put back in place again in K, thereby restoring the original execution context the effect was raised from. This means that there is no extra effort needed to manage the handler stack separately: the handler stack is managed *automagically* by the use of continuations.

Perform and with-handle

The remaining two features of Eff can be conveniently implemented with hlosure frames too. The rule for with-handle blocks simply prepends $([], \mathcal{H})$ to K, where $[]$ is an empty continuation stack and $\mathcal{H}$ is a hlosure representing the handler from the with-handle block.

Performing effects is a little bit more difficult due to the unwinding of the K component.

When an effect is performed we must search for the closest handler capable of handling the effect. However, we must also concatenate together all the delimited continuations from the hlosure frames we jump through. Figure 3.2 depicts concisely how this can be done. The CEK state is temporarily extended with a fourth component which behaves as an accumulator for hlosure frames (and hence as an accumulator for delimited continuations).

If a handler frame contains a hlosure $\mathcal{H}$ that can handle the performed effect then κ (the list of hlosure frames carrying the delimited continuation in the fourth component) is bound to k and the identifier of the effect argument from the effect case is bound to e in $\mathcal{H}'s$ environment. Otherwise, the current hlosure frame is removed from the CEK state and is appended to κ and the unwinding continues.

An OCaml code snippet showing how to implement this same functionality can be found in Appendix B. A nice feature of functional languages is that once we get our theory right, the theory usually lends itself to an almost trivial implementation. This is the case with the CEK machine too and this is why I will not talk about the implementation in detail here—however, a typical rule implementation can be found below.

All transition rules are implemented via a `step` function which performs a single step of the transition rules shown in Figure 3.1. This can be seen in Listing 3.3 which shows how to interpret terms of the form **let** $x = c_1$ **in** c_2.

Listing 3.3: A typical implementation of a rule in the `step` function

```
1  (* cek_state -> cek_state *)
2  let step state =
3  ...
4  match state.c with
5  ...
6  | CEKletin (x, c1, c2) ->
7    let continuation = { input = x; control = c2; kont_env = state.e } in
8    {
9      c = c1;
10     e = state.e;
11     k = push_continuation continuation state.k
12   }
```

$$\langle \textbf{perform } E(e,k), \gamma, (\mathcal{C}, \mathcal{H}) :: K, \kappa \rangle_{\text{unwind}} \longrightarrow \langle \textbf{perform } E(e,k), \gamma, K, \kappa @ [(\mathcal{C}, \mathcal{H})] \rangle_{\text{unwind}}$$
$$\text{if } \mathcal{H} = (H, \_) \text{ and handler } H \text{ does not handle effect } E$$

$$\langle \textbf{perform } E(e,k), \gamma, (\mathcal{C}, \mathcal{H}) :: K, \kappa \rangle_{\text{unwind}} \longrightarrow \langle M, \gamma'[x \mapsto (\llbracket e \rrbracket \gamma), k \mapsto \kappa @ [(\mathcal{C}, \mathcal{H})]], K \rangle$$
$$\text{if } \mathcal{H} = (H, \gamma') \text{ and handler } H \text{ handles effect } E \text{ with rule } \textbf{effect } E\ x\ k \to M$$

Figure 3.2: Unwinding of the K component in the CEK machine

3.2 The SHADE virtual machine

The SHADE virtual machine is an SECD-like virtual machine in that it has a dump component, however, it interprets linearised bytecode (SHADEcode, which is described later in Table 3.1) rather than the source terms of a language. The name is an anagram from the initials of the main components of the machine: *accumulator* (A), *dump* (D), *environment* (E), *hlosure* (H) and *stack* (S). The rest of this section is concerned with a description of this machine using its transition rules. Section 3.2.2 will give an overview of how the bytecode implements the different aspects of the Eff language.

Configuration The state of the machine can be characterised with a two-tuple $\langle A, D\rangle$ where A is the accumulator and D is a dump. I call the elements of D *shadows*. Shadows are the SHADE equivalent of CEK hlosure frames. A shadow is a four-tuple $\langle pc, E, H, S\rangle$, where pc is the program-counter of the shadow, E is an environment, H is a hlosure and S is a stack [1].

Only the *top shadow* is executing of all the shadows at any point in the execution. To avoid having to refer to the top shadow through the D component I will use the notation $\langle A, d, D\rangle$ instead to denote the SHADE configuration $\langle A, d :: D\rangle$.

In the program counter field (pc) of a shadow I will write the instruction the program counter points to and its value interchangeably. On the left hand side of the transition rules it is more convenient to refer to the instruction and on the right hand side it is often useful to write $pc + 1$ to mean that the program counter is incremented or write L to mean that we perform a jump to the label L.

3.2.1 Transition rules and SHADEcode

Figure 3.3 shows only transition rules for *inter-shadow instructions*. These are instructions which implement the interactions between shadows (i.e., they manipulate the D component of the machine and implement the Eff-specific features like performing effects, applying continuations, entering and returning from with-handle blocks and handler cases).

Intra-shadow instructions are standard stack machine instructions similar to those of the Caml Virtual Machine [1]. I will not give formal transition rules for these here. Table 3.1 includes all SHADE-specific instructions and explains their purpose in prose.

The SHADE machine is initialised by loading a program compiled to bytecode and initialising D to a list containing an *empty shadow* with the default handler H_{def}. The default handler is responsible for handling effects which rise to the toplevel[2]. The machine terminates when it reaches the **halt** instruction.

Interaction of shadows I will start to introduce the SHADE bytecode and the SHADE transition rules[3]. Let us consider how with-handle blocks are compiled:

$$[\![\textbf{WithHandle } (h, e)]\!]^{\gamma}_{s} = [\![h]\!]^{\gamma}_{s}; \textbf{push}; \text{fvs-to-stack}(e, s, \gamma); \textbf{makeclosure } N, L; \textbf{castshadow}; \textbf{fin}.$$

[1] The SHADE machine had many versions throughout the year and this final version turns out to be surprisingly similar to the solution Multicore OCaml uses with its fibres—the heap-allocated fibres of OCaml show a lot of resemblance with the shadows of the SHADE machine.

[2] Runtime systems can define built-in effects this way. The Evaluation chapter will show how can one implement a web server in Eff by defining 2 built-in effects which interact with the Linux kernel.

[3] Unfortunately, there is a circular dependency between the concepts in this section. I apologise in advance for starting to talk about the compilation this suddenly, but the cycle must be broken somewhere.

Initialisation:

$$\text{program} \to \langle (), (0, \{\}, H_{def}, []), [] \rangle$$

Termination:

$$\langle v, (\mathbf{halt}, E, H, S), D \rangle \to v$$

Transition rules for Eff-specific instructions:

$$\langle \mathcal{C}(cp, \gamma), (\mathbf{castshadow}, E, H, \mathcal{H} :: S), D \rangle \to \langle A, (cp, \gamma, \mathcal{H}, []), (pc + 1, E, H, \mathcal{H} :: S) :: D \rangle$$

$$\langle A, (\mathbf{killshadow}, E, \mathcal{H}(h, \gamma), S), D \rangle \to \langle A, (L_{\text{valcase}(h)}, \gamma, \mathcal{H}(h, \gamma), S), D \rangle$$

$$\langle A, (\mathbf{fin}, E, H, \mathcal{H}(h, \gamma) :: S), D \rangle \to \langle A, (L_{\text{fincase}(h)}, \gamma, (pc, E) :: S), D \rangle$$

$$\langle A, (\mathbf{throw}\ id, E, H, S), D \rangle \to \langle A, (\mathbf{throw}\ id, E, H, S), D, [] \rangle^{id}_{\text{unwind}}$$

$$\langle A, (\mathbf{ret2}, E, H, S), D \rangle \to \langle A, D \rangle$$

$$\langle A, (\mathbf{apply}, E, H, \mathcal{C}(cp, \gamma) :: S), D \rangle \to \langle A, (cp, \gamma, H, (pc, E) :: S), D \rangle$$

$$\langle A, (\mathbf{apply}, E, H, \kappa :: S), D \rangle \to \langle A, \kappa \ @\ ((pc, E, H, S) :: D) \rangle$$

Figure 3.3: SHADE machine transition rules

The handler is compiled and then pushed to the stack. Then the closure of the with-handle block's body is constructed and a new shadow corresponding to this block is created by the **castshadow** instruction. L is a *fresh* label denoting the start of the with-handle block's body. The code for the body of the with-handle block is generated separately. The **fin** instruction invokes the finally case after control is returned to this shadow again. When this happens, the hlosure of h is still on the stack of this shadow and this is the reason why the **castshadow** instruction does not consume the hlosure from the stack. It might not be obvious why we need a separate instruction for finally cases. The reason is that control can return in many ways: it can come from a value or effect case via a **ret2** instruction. However, when many continuations are resumed in the same with-handle block (think about the Hello World example), then only the last **ret2** instruction actually leaves the with-handle block and therefore we cannot make the invocation of finally cases a feature of the **ret2** instruction. This is the responsibility of the same shadow a with-handle block resides in.

Another important detail is that three syntacticly similar contstructs are not compiled the same way: the bodies of functions, with-handle blocks and handler cases. The body of with-handle blocks always ends with a **killshadow** instruction, value cases and effect cases end with a **ret2** instruction, the bodies of functions and finally cases end with **ret**.

The reason why the transition rule of **killshadow** invokes the value case of the current handler is because the bodies of with-handle blocks end with **killshadow** instructions—**killshadow** does not actually destroy the current top shadow; it simply prepares it for the execution of the value case of a handler by jumping to the corresponding label and putting the environment of the hlosure in E. As value cases end with **ret2** instructions and they always destroy the top shadow, the job is finished by the **ret2** of a value case.

If the behaviour of an effect is exception-like then control might not reach the end of a with-handle block. This justifies why **killshadow** cannot be responsible for actually destroying shadows

$$\langle A, \underbrace{(pc, E, H, S)}_{d}, D, \kappa\rangle^{id}_{\text{unwind}} \text{ and } H \uparrow id \to \begin{cases} \text{Runtime exception} & \text{if } D = [] \\ \langle A, d', D', \kappa \mathbin{@} [d]\rangle^{id}_{\text{unwind}} & \text{if } D = d' :: D' \end{cases}$$

$$\langle A, (pc, E, \mathcal{H}(h, \gamma), S), D, \kappa\rangle^{id}_{\text{unwind}} \to \langle A, (L_{\text{effcase}(h,id)}, \gamma[x \mapsto A, k \mapsto \kappa]), D\rangle \text{ if } h \downarrow id$$

Figure 3.4: Dump unwinding in SHADE

or for invoking the finally case of a handler which further justifies the existence of the **fin** instruction.

The **fin** instruction implements the invocation of finally cases as a function call but it also pops the handler the **castshadow** instruction left on the stack. With this, Eff effect handlers are implemented.

Similarities with the CEK machine The **apply** instruction implements function and continuation application the same way it was done in the CEK machine. The **throw** instruction is used to perform an effect. When this happens the dump has to be unwound the same way the K component of the CEK machines needed to be unwound.

Calling convention Another fine but important detail is that the calling convention for functions, continuations *and* effects must match. At the time of the compilation we have no idea whether an identifier will stand for a function or a continuation. One might think that this still leaves us many possibilities, but this actually completely defines the calling convention in SHADE.

We cannot use the stack to pass function or continuation arguments around because by resuming a continuation we are also restoring shadows and therefore we lose the access to the current stack. The same goes for returning values. Hence we must use the accumulator.

Compiling to SHADEcode Compilation to SHADEcode is standard. We keep track of the size of the stack s and a compilation environment γ which tells us where the free variables of currently compiled expressions reside (stack or environment). The equation

$$[\![\textbf{LetIn } (x, e_1, e_2)]\!]^{\gamma}_{s} = [\![e_1]\!]^{\gamma}_{s}; \textbf{push}; [\![e_2]\!]^{\gamma[x \mapsto s]}_{s+1}$$

describes how one can compile let expressions. The expression e_1 is compiled first which generates the code that leaves the value of e_1 in the accumulator. Then a **push** instruction saves this on the stack and the expression e_2 is compiled with the knowledge that the size of the stack has increased to $s + 1$ and the new compilation environment knows that x resides at position s in the stack ($\gamma[x \mapsto x]$). A full description of the compilation process in this style can be found in Appendix A.

3.2.2 How does SHADEcode implement Eff?

I gave a formal but dry description of the SHADE machine and its bytecode in the previous section. The reader might rightfully demand an explanation of why SHADE makes sense for Eff. In this

Instruction	Description	Action
apply	Apply the top of the stack to the accumulator.	see Figure 3.3
ret	Return with the value in A. Restore the old program counter and environment from S.	see Figure 3.3
ret2	Return from a value or effect case with the value in A. Destroy the top shadow and increment the program counter of the new top shadow.	see Figure 3.3
throw `eid`	Throw an effect. `eid` is an integer denoting the type of the effect being thrown. The argument to the effect resides in the accumulator.	see Figure 3.3
fin	Invoke the finally case of the hlosure on the top of the stack as a function call using the accumulator as the argument. Pop the hlosure from the stack.	see Figure 3.3
castshadow	Cast a new shadow.	see Figure 3.3
killshadow	Kill a shadow.	see Figure 3.3

makeclosure L, N	Form an environment of A and the top $n-1$ elements of S and create a closure with this environment of location L.	`E = [A, S[0:N-1]];`
		`S.pop(N-1);`
		`A := closure(L, E)`

makehlosure N, L_v L_f, $(\overline{L_{e_i}, \texttt{eid}_i})$	Similar to **makeclosure** L, N but with many locations each corresponding to a handler case. Effect cases are represented with pairs $(L_i, \texttt{eid}_i)$ where L_i is a code pointer of the effect case and $\texttt{eid}_i$ is an integer identifier of the type of the effect being handled.	`E = [A, S[0:N-1]];`
		`S.pop(N-1);`
		`A := hlosure (...)`

Table 3.1: SHADEcode, the instruction set of the SHADE machine

section I will show how the bytecode can implement the various flow of control algebraic effect handlers allow.

It is clear that when no new shadow is cast (no with-handle is used) then the SHADE machine is essentially a Zinc [11] machine with all effects behaving like exceptions. The next page contains various examples and should give a good insight into how everything fits together.

Figure 3.5 shows what happens when no effect is performed in a with-handle block. The with-handle block is entered via **castshadow** (a second shadow is cast and we jump to `Lwith`). Next, after the body of the with-handle block executes fully the **killshadow** instruction gives the control to the value case. The **ret2** instruction at the end of the value case destroys the second shadow and returns to the default shadow while increasing the program counter. Now the **fin** instruction is responsible for executing the finally case and getting rid of the hlosure from the stack. The **ret** instruction at the end of the finally case simply returns as if it was a function call and execution halts as the program counter now points to **halt**. The result of the program is `86`.

Figure 3.6 depicts a case when the effect has exception-like behaviour. This case is different from the previous one in that the **killshadow** instruction is never executed. However, the **ret2** instruction at the end of the effect case of handler h will still destroy the shadow and return to

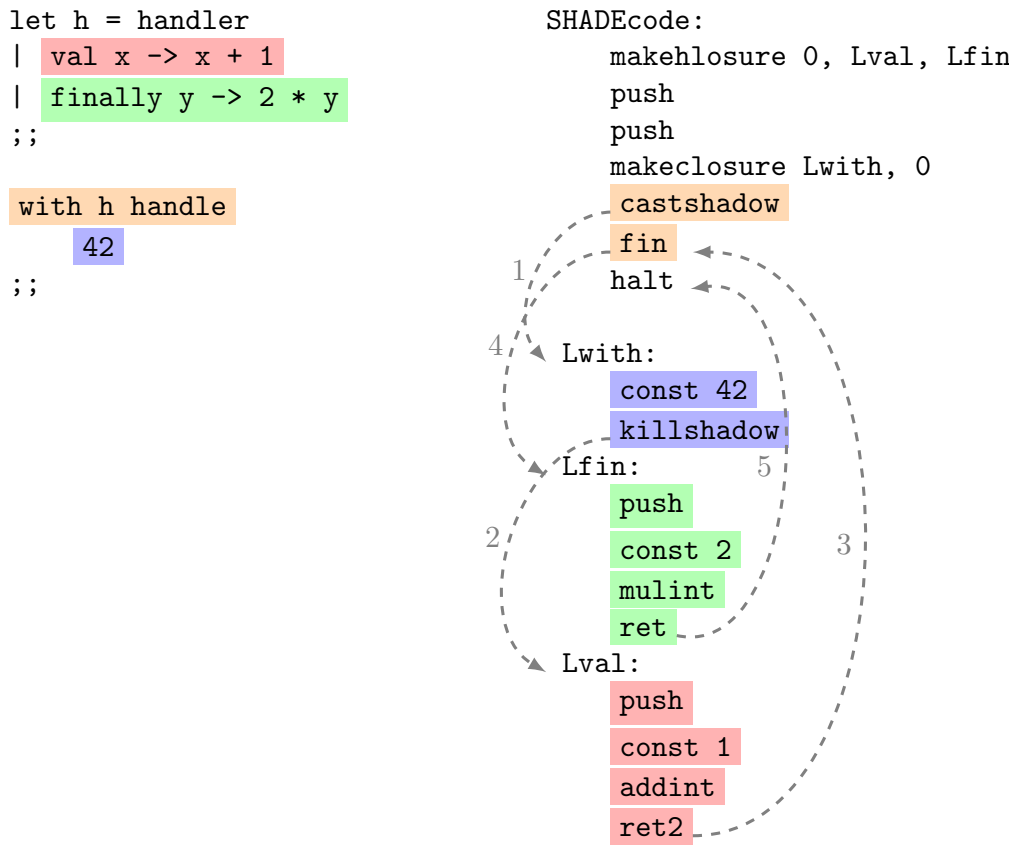


Figure 3.5: No effect is performed

the **fin** instruction which is executed before the machine halts as we would expect. The program returns with the string `"Error: got exception oops!"`.

Figure 3.7 shows an example where the continuation is resumed in the effect case (this is actually our beloved Hello World again!). This case is particularly interesting as effects are performed in the with-handle block, continuations are resumed, the **killshadow** instruction is executed and the **ret2** of the value case *returns to an effect case* which then returns to another effect case and finally the top shadow is destroyed by the **ret2** of an effect case just as we saw in the previous example.

The red dots next to the **apply** instruction are there to show that the dump is essentially a separate call stack for continuations and effect cases. Two separate **ret2** instructions return to the same **apply** instruction but first the effect case handling `Print "not to be"` is given control and only after that effect case is finished can the effect case handling `Print "Try"` resume. This program returns the tuple `("confused!", "Try not to be")`.

Figure 3.8 shows what happens when we take everything to the next level and use multi-shot continuations (i.e., we resume the same continuation more than once). This is a typical non-determinism example where an effect case resumes a continuation with both `true` and `false` to determine all the values the with-handle block can evaluate to[4]. Figure 3.8 also outlines the structure of the dump. The red arrows denote the instructions which increase the size of the dump and the black arrows are just there to show where we are in the execution. The blue **ret2** instructions are those of the value case's and they are only invoked "at depth 4" in the diagram.

[4]We are expecting the result `[5,-5,15,5]` as $(x - y)$ can be $(10 - 5)$, $(10 - 15)$, $(20 - 5)$ or $(20 - 15)$.

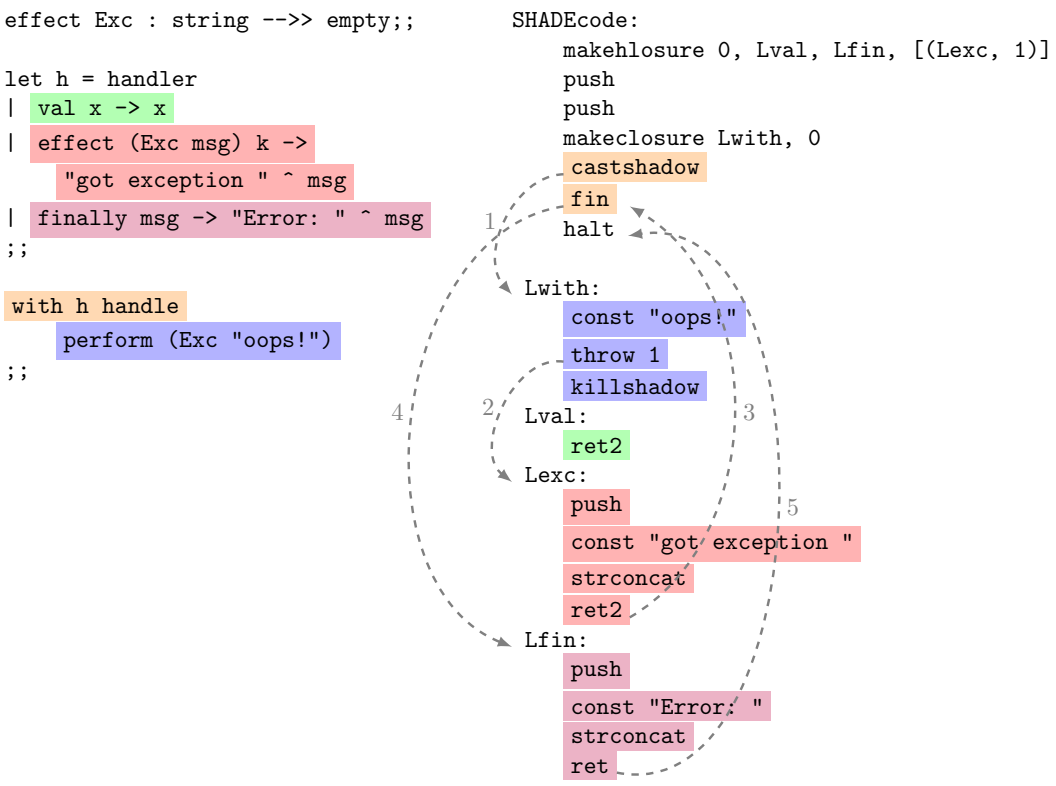


Figure 3.6: An effect is performed but its continuation is not resumed

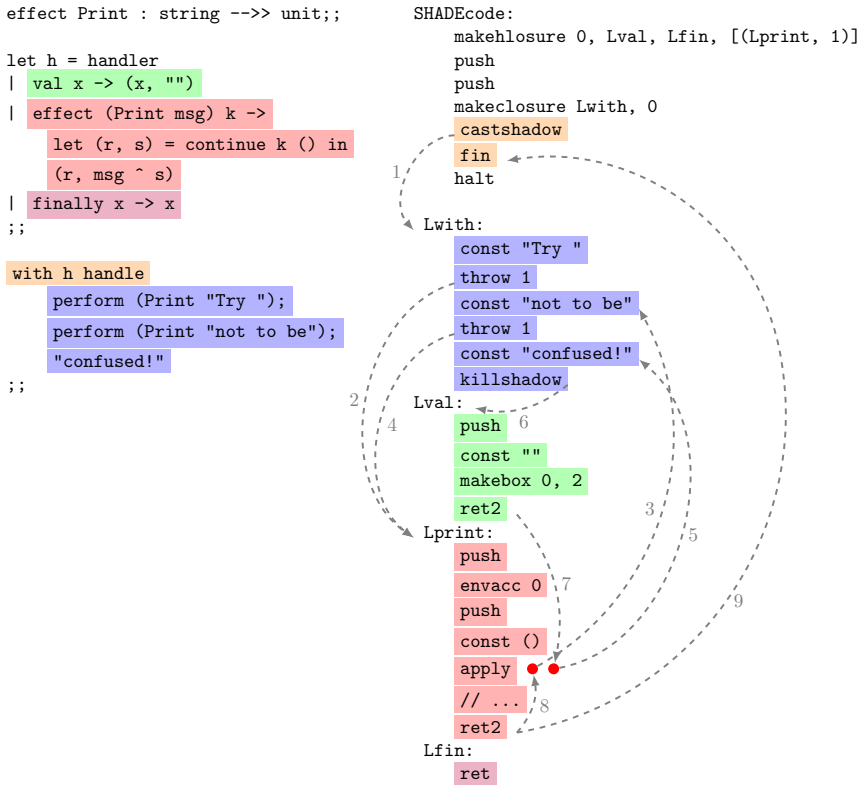


Figure 3.7: An effect is performed and its continuation is resumed

```
effect Choose : unit -->> bool;;

let h = handler
| val x -> [x]
| effect (Choose ()) k ->
    (continue k true) @ (continue k false)
| finally x -> x
;;

with h handle
    let x = if (perform (Choose ())) then 10 else 20 in
    let y = if (perform (Choose ())) then 5 else 15 in
    x - y
;;
```

```
1  SHADEcode:
2  makehlosure 0, Lval, Lfin, [(Lchoose, 1)]
3  push
4  push
5  makeclosure Lwith, 0
6  castshadow
7  fin
8  halt
9  Lval:
10   makebox 0, 1
11   ret2
```

```
12 Lchoose:
13   envacc 0
14   push
15   const true
16   apply
17   push
18   envacc 0
19   const false
20   apply
21   lstappend
22   ret2
```

```
23 Lwith:
24   const ()
25   throw 1
26   push
27   //...
28   const ()
29   throw 1
30   push
31   //...
32   subint
33   killshadow
34 Lfin:
35   ret
```

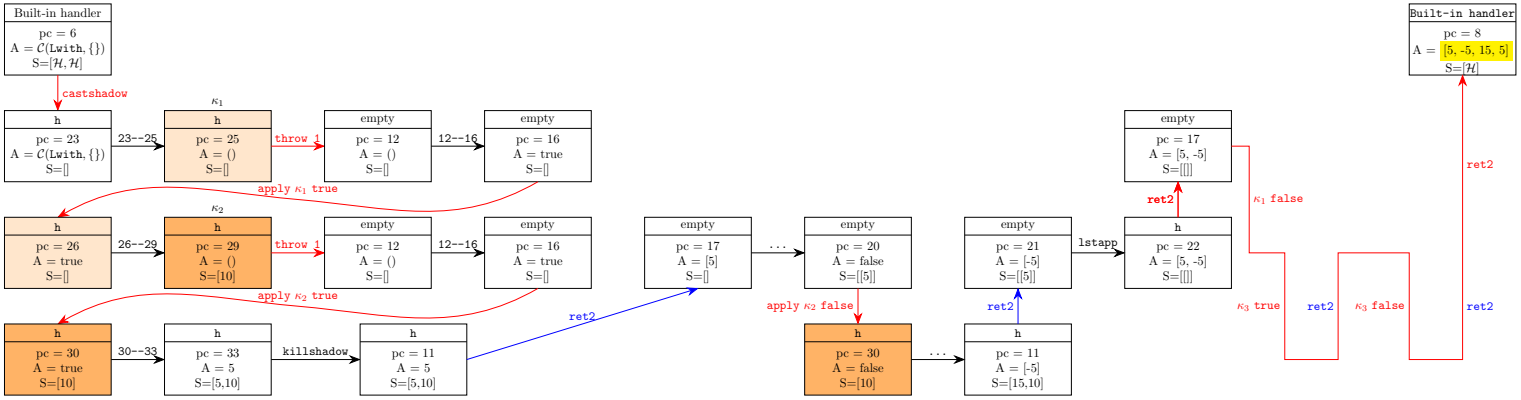


Figure 3.8: Non-determinism: multi-shot continuations can be resumed more than once in Eff

Continuations are highlighted with shades of orange. Because all continuations are invoked twice these must be cloned which can add a significant copy overhead. This is one way Eff differs from Multicore OCaml. Multicore OCaml only supports one-shot continuations and programmers must be aware whether they already used a continuation or not. In Multicore OCaml the same continuation can only be used when it was cloned before resuming it for the second time. The programmer must manually do this with the `Obj.copy_continuation` call. If the programmer forgets about this then the program can crash. As Eff is different in this respect, the SHADE machine always keeps a clean copy of every continuation which can be cloned on demand when it is resumed. This is an inefficiency but is necessary to ensure correctness.

3.3 Software engineering

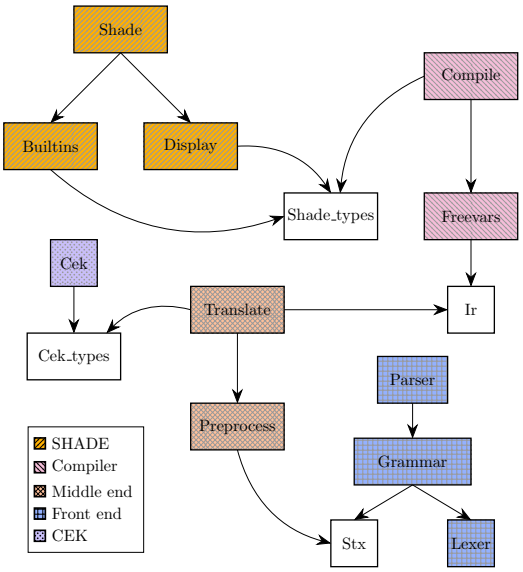
3.3.1 Requirements analysis and choice of tools

Knowing that my project would be theory based and that I would devote a lot of time to experiment with interpreters and their transition rules, it made sense to choose a functional language like OCaml to ease the implementation. The OCaml ecosystem looked mature enough to support a bigger project. I knew that I could use the `ocamllex` and `menhir` libraries to generate parsers and I knew about the existence of convenient build systems (like `dune`) and test libraries (like `alcotest`) which ease the development.

I was following OCaml forums before and I was aware of the ways I could seek quick help from the online communities in case that was needed. I knew that my evaluation will potentially involve measurements against Multicore OCaml's virtual machine and that it will be convenient to obtain and switch between different OCaml switches using `opam`.

3.3.2 Repository overview

The repository of my OCaml implementation is structured as a typical OCaml package. The `bin` directory contains the entry points of the executables generated, the `lib` directory contains the collection of modules making up the project. The `test` directory contains unit tests and regression tests.



Listing 3.4: SHADE VM's pipeline

```
1  open Core;;
2  open Lib;;
3
4  let main path_to_source =
5      path_to_source
6      |> Parser.parse_from_file
7      |> Preprocess.convert
8      |> Translate.stx_to_ir
9      |> Compile.to_byte_code
10     |> Shade.exec
11     |> Display.pretty_vm_result
12 ;;
```

Figure 3.9: Dependency graph of the modules residing in `lib` and the way they are used to build the SHADE VM pipeline. The project's main components are marked with different colours and shared modules are denoted with white.

There was a further 800 lines of OCaml code and 500 lines of Eff code written for testing as well as around a few hundred lines of extra OCaml, Eff, Python and shell scripts to automate the evaluation process and the (re-)plotting of the collected data.

Module name	Purpose	LoC
Stx	Specifies Eff syntax trees. The `ocamllex` and the Menhir parser generator uses this module to generate a parser for Eff.	132
Grammar (.mly)	Contains the grammar for the syntax of Eff. This is an `ocamlyacc` file written by me.	227
Grammar_utils	Contains the implementation of simple helper functions aiding the parsing process.	10
Lexer (.mll)	Contains a set of regular expressions used to generate a lexical analyser. This is an `ocamllex` file written by me.	117
Parser	Uses **Lexer** and **Grammar** to implement a parser.	6
Preprocess	Implements preprocessing functionalities on ASTs. At this stage all identifiers are given unique integer De Bruijn indices and syntactic sugar is removed.	343
Translate	Implements various translation functionalities. Erases the distinctions between computations and expressions and implements translations from ASTs to various intermediate representations.	328
Cek_types	Contains the specification of the CEK machine discussed in this chapter.	130
Cek	Implementation of the CEK machine.	403
Ir	Specification of an intermediate representation close to *A-normal form*. The `Compile` module generates SHADE bytecode from this representation.	77
Freevars	Implements analysis of free variables on the IR.	87
Compile	Implements the compilation from IR to SHADE bytecode.	434
Shade_types	Contains the specification of the SHADE machine.	93
Display	Implements pretty printing of VM results and contains printers used for the debugging of the SHADE VM.	82
Builtins	Contains the implementation of built-in effects of the SHADE runtime system.	78
Shade	Implements the SHADE machine.	576

Table 3.2: The modules of `lib`

3.3.3 Techniques

Testing As stated in the Introduction correctness and hence testing were crucial for the success of this project. Therefore I took testing very seriously. I used the Alcotest library to run my test suites. In total there are a 104 tests covering the project. Figure C.2 shows a screenshot of a typical output Alcotest produces when testing the SHADE virtual machine.

As I used test driven development as my development methodology I wrote most of my tests upfront. However, when a bug was discovered later I added regression tests too.

Version control I used Git together with GitHub[5] to version control my project and used many branches to organise development better. The project comprises 168 commits and 12 branches. My main branch was `master` which always contained a stable (in the sense that it passed all the tests) version. I categorised the other branches into 3 categories.

[5] `https://github.com/`

- *Feature branches* were used to separate the development of new features from the already stable version of the project. Feature branches were merged to the main branch when I judged them to be stable enough.
- *Refactoring.* I frequently went back to refactor pieces of code I was not satisfied with. From summer internships I knew that I will do this often and hence I had a separate branch for this too. This was useful because when looking at `git log` I could quickly tell apart the meaningful commits (ones which add new code) and others which just improved the quality of the code or added documentation or comments.
- *Benchmarking.* I used a separate branch for evaluation.

OPAM The project is packaged as an OPAM package. After further refactoring I might publish this project.

Coding style and documentation I tried to stick to OCaml best-practices while developing, such as always having interface `.mli` files for all my modules. In these I used `(**` style comments to document my code. This proved very useful later as the `odoc` OCaml tool can generate browsable HTML documentation from these `mli` interfaces—an idiomatic format OCaml programmers are familiar with (see Figure C.1 for a typical example). I often found myself reading this documentation when I was implementing a complex part of the software. Documentation can also speed up the learning process for other hobbyist language enthusiasts wishing to contribute to the project were I to publish this package on GitHub.

Time management Time management is another important aspect of software engineering. The original timeline proved to be over-ambitious but there was enough time set aside for potential delays and unforeseeable complications. I also allocated enough time for my other academic responsibilities (such as completing Units of Assessments and doing supervision work) and hence these did not interfere much with the workflow of my project during the year.

Backups My strategy for saving my work and creating backups happened as per it is described in the Project Proposal. I only had to use backups once when I re-installed my computer.

3.4 Summary

- The design of a syntax-level CEK interpreter was described in this chapter.
- The design of the SHADE virtual machine which interprets *linearised* bytecode was explained and SHADEcode was introduced.
- The software engineering practices used to carry out the work were discussed. It was explained why I chose the different tools I used in respect of the requirements of my project.

4 | Evaluation

This chapter is concerned with comparing the performance of my CEK interpreter and my SHADE VM with existing solutions implementing algebraic effect handlers. The comparison is split into two parts: to evaluate the performance of the CEK interpreter I compare against the Eff interpreter and against the interpreter of Multicore OCaml. The SHADE virtual machine is compared to Multicore OCaml's bytecode interpreter.

At the end of this chapter a proof of concept web server is described. This is more of a qualitative evaluation and is supposed to show what can be done with effect handlers in the real-world.

Evaluation strategy and technical details I run multiple benchmarks each of which uses effect handlers in a different way. Each benchmark is run 10 times and on the plots the arithmetic average of the execution times is reported (the standard deviation is displayed as error bars). Data was collected using GNU's `time` command. During the measurements my laptop was connected to the power supply and I tried to minimise noise in the measurements by not running any other process (apart from a terminal) and turning off unnecessary functionalities like WiFi, Ethernet, Bluetooth, etc.

All benchmarks are executed on a Lenovo Thinkpad X1 6$^{\text{th}}$ gen. with an Intel Core i7-8565U CPU, 16GB of RAM using a single thread. The operating system in use was a 64bit Ubuntu 18.04.4 LTS, Linux kernel version `5.3.0-46-generic`.

Eff was compiled from its official GitHub repository at the time where the top commit had SHA1 `796900d`[1]. The version of Multicore OCaml used in the benchmarks was obtained by OPAM, the identifier of the OPAM switch was `4.06.1+multicore`.

4.1 Exceptions

The exception benchmark raises and catches exception-like effects (the continuation is never resumed). I must add that in the OCaml benchmark I used effects simulating exceptions rather than *actual* OCaml exceptions. Actual OCaml exceptions are implemented very efficiently in OCaml (they are not resumable at all).

Figure 4.1 shows that the OCaml interpreter beats the CEK interpreter in this benchmark. However, we can see on Figure 4.2 that the performance of SHADE VM is comparable to Multicore OCaml's VM even though the execution time for SHADE includes the compilation time to SHADEcode, but the times for Multicore OCaml do not include compilation. As I did not implement serialization for my bytecode Eff programs were recompiled every time a SHADE benchmark was run. However, this extra time is less and less significant as we increase the number of exceptions.

Note that as the bytecode solutions are vastly more efficient than their interpreter counterparts they are compared with a larger number of exceptions. It takes around 63 seconds for the Eff interpreter to evaluate the benchmark with 5000 exceptions, whereas this takes only 3.5 seconds for the CEK interpreter. For SHADE VM the same takes around 200 milliseconds (with compilation included), which is a 300-fold improvement!

The y-axis is logarithmic and the plots show a linear slowdown when we increase the number of exceptions linearly. This suggests an $\mathcal{O}(N)$ time complexity for performing and catching N

[1]The official repository is `https://github.com/matijapretnar/eff/` at the time of writing.

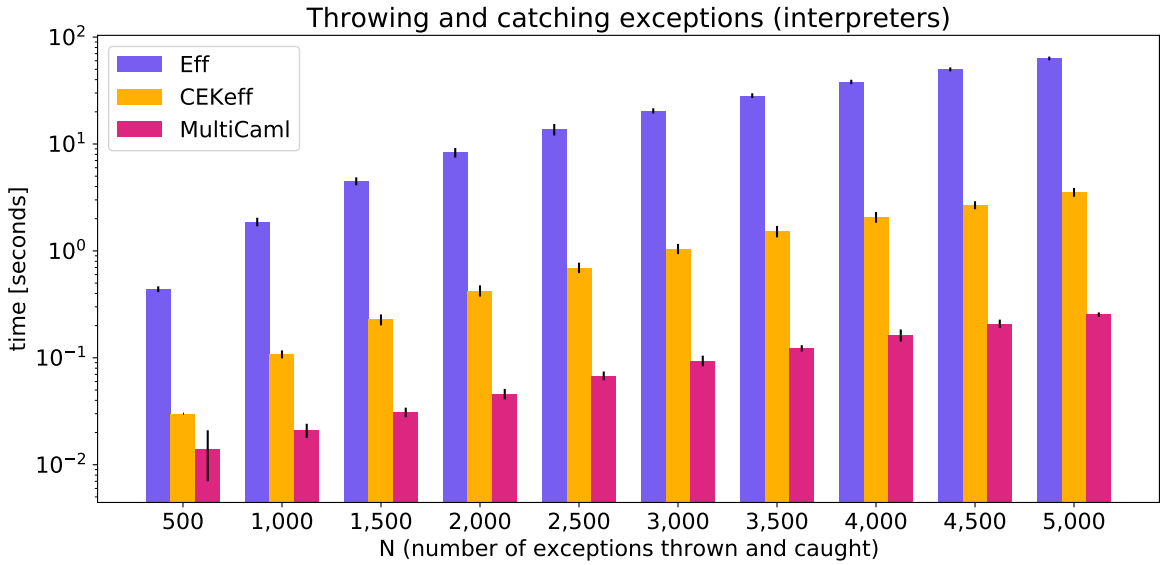


Figure 4.1: The CEK interpreter performs better than Eff but worse than Multicore OCaml

exceptions which suggests that my implementation does not contain bugs or inefficiencies that accumulate as more effects are performed in a program.

4.2 State

The state benchmark implements an integer state which gets incremented every time an `Incr` effect is performed. The handler of the `Incr` effect resumes a continuation exactly once. The reason why I chose such a computationally cheap operation as integer addition is that I wanted to measure the performance of performing effects and resuming continuations. As the number of performed effects N increases we can be confident that the cost of performing effects and resuming continuations will dominate the execution time and that we are measuring what we want.

The results are displayed on Figure 4.3 and Figure 4.4. The interpretation of these results is similar to the exception case although they reveal a *very important property* of the implementation.

In the exception case the size of the dump in SHADE VM remained constant. However, in this case the size of the dump grows linearly with N. As the dump is essentially a linked list of shadows residing in the heap it is reassuring to see that the performance does not degrade rapidly with N. In fact, we see an $\mathcal{O}(N)$ slowdown again, which is what one would expect from any efficient implementation (but we would not want a solution where the cost of performing an effect or resuming a continuation grows like $\mathcal{O}(N^2)$ for instance).

4.3 Non-determinism – Solving the N-queens problem

The N-queens[2] benchmark is a backtracking program which resumes continuations more than once. Here I do not compare against the CEK interpreter for the bigger benchmarks because due

[2]The N-queens problem is concerned with finding the positions of N queens on an $N \times N$ chessboard such that no two queens attack each other.

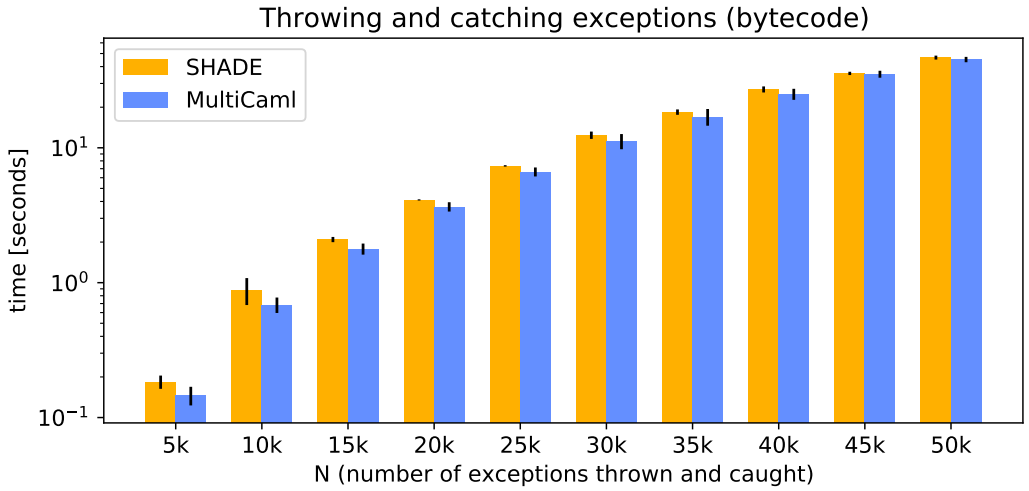


Figure 4.2: The peformance of SHADE is comparable to Multicore OCaml's

to its naïve implementation it slows down significantly after $N > 14$.

The performance of SHADE VM remains acceptable, although worse than the VM of Multicore OCaml as Figure 4.5 shows. The reason why Multicore OCaml performs so much better than SHADE VM was already mentioned briefly in the Implementation chapter. Multicore OCaml implements *one-shot continuations* and relies on the user to specify when a continuation should be cloned.

Listing 4.1: Cloning continuations must be done explicitly in Multicore OCaml

```
| effect (Decide ()) k ->
  try continue (Obj.clone_continuation k) false with
  | effect (Fail ()) _ -> continue k true
;;
```

Figure 4.6 reveals the extent to which this extra cloning affects SHADE VM's performance. I added a counter to my runtime system just for this benchmark to measure how many times continuations were cloned (the original benchmark did not have such a counter as I did not want performance monitoring to interfere with my results, i.e., the two metrics were measured in separate runs). We see that there is a good correlation between the number of extra clone calls and the memory used in the program. It is also possible to determine the average size of a continuation from the raw data. It turns out that the typical size of continuations is between 1 and 2 Kbytes (based on the bigger test cases, i.e., where N is 16, 18 or 20).

4.3.1 Vulnerability of the evaluation

A subtlety must be pointed out here. The N-queens benchmark traverses a huge search space of all N-queen configurations on an $N \times N$ chessboard. The order in which we inspect the configurations **does** matter here.

If one were to repeat this experiment one would have to ensure that their backtracking strategy is the same if they wish to compare their results to this evaluation. I used the same strategy in all my measurements and although this is a popular benchmark I do not compare my results with

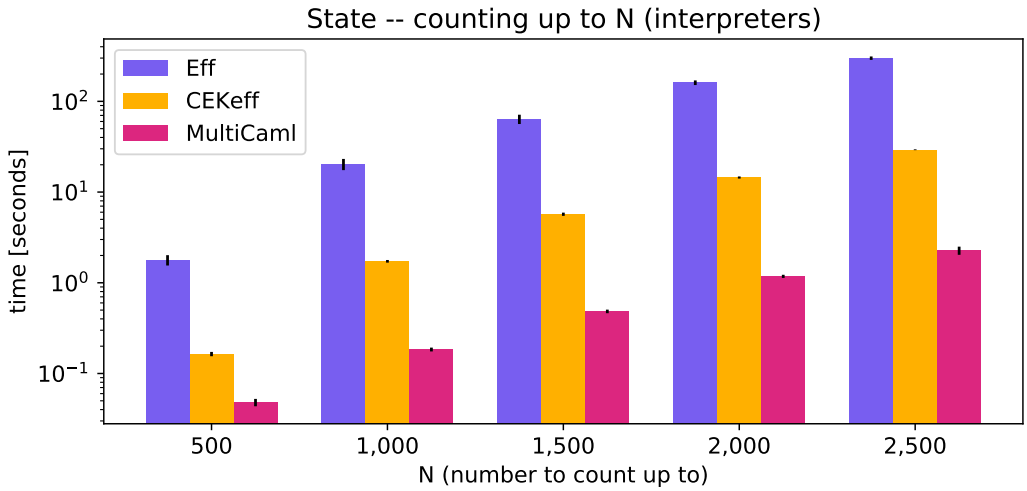


Figure 4.3: The CEK interpreter performs better than Eff but worse than the interpreter of Multicore OCaml

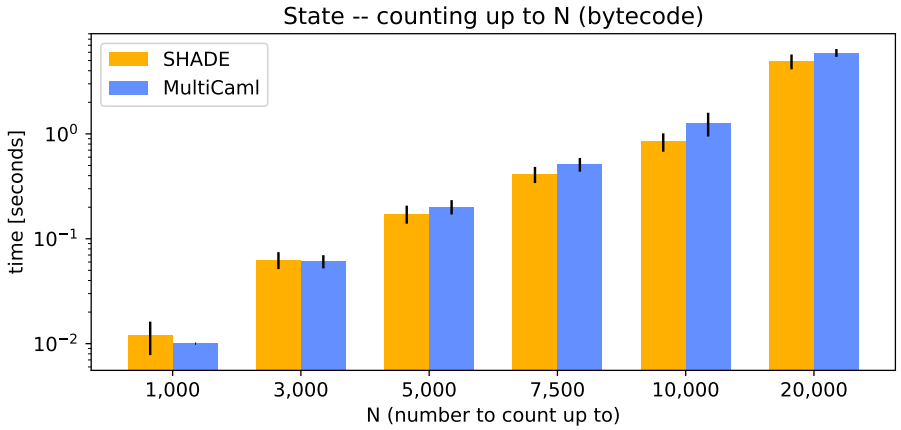


Figure 4.4: SHADE VM's performance is still head-to-head with Multicore OCaml

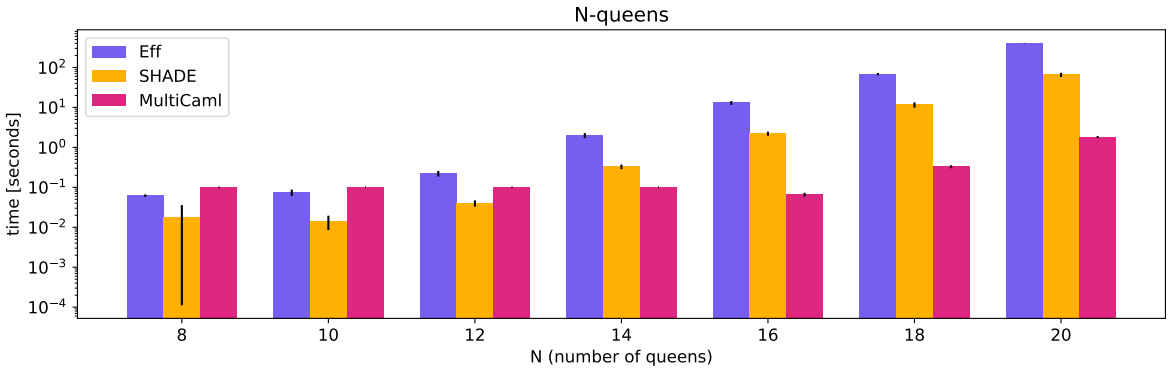


Figure 4.5: SHADE VM loses against Multicore OCaml here but is still more performant than the Eff interpreter

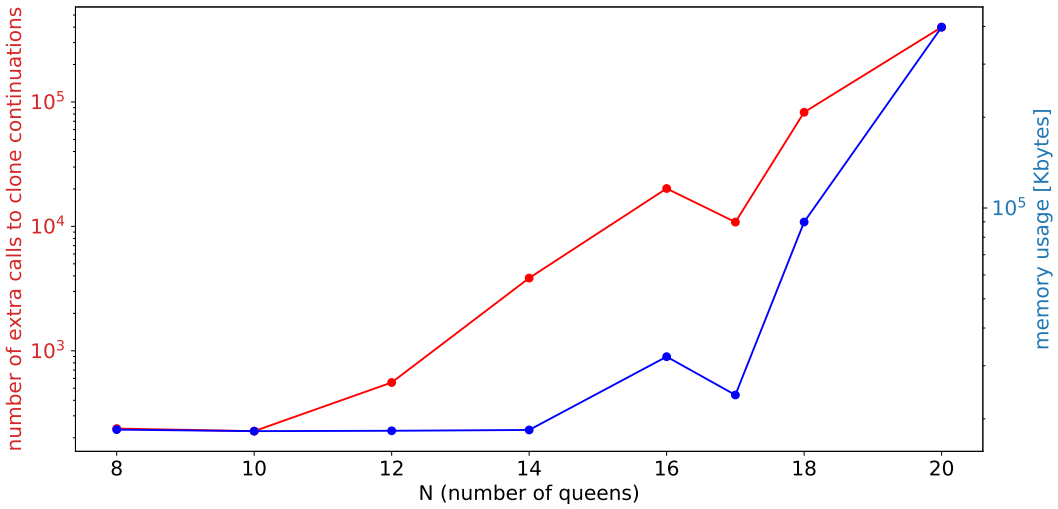


Figure 4.6: Investigating the effect of cloning on performance and memory usage

benchmarks from other papers as the papers often do not mention what strategy they use.

An even more subtle point is that N matters too. One must not fall in the pitfall of comparing the results for cases with different sizes (Figure 4.6 reveals for instance that the $N = 17$ case is actually "easier" than the $N = 16$ case; so is $N = 10$ easier than $N = 8$) as N does not represent the complexity of the problem, it is merely the identifier of a big search space. Interestingly, the $N = 8$ and $N = 10$ cases are nearly always reported in papers whereas the $N = 17$ is consistently left out as it is an obvious outlier.

4.4 Concurrency – Hello Online World!

Cooperative multitasking can be implemented with algebraic effects and their handlers too. We can realise *green threads*[3] as `unit` → `unit` functions which can be `Spawn`ed and which are capable of `Yield`ing control. The reader might correctly suspect that these operations will correspond to algebraic effects with the following types:

Listing 4.2: Effects for implementing a Hello World webserver

```
1 (* Concurrency for green threads *)
2 effect Yield : unit -->> unit;;
3 effect Spawn : (unit -> unit) -->> unit;;
4
5 (* Asynchronous effects in SHADE *)
6 effect HTTPHello : int -->> unit;;
7 effect Accept : unit -->> int;;
```

I built in two additional asynchronous effects in the runtime system of SHADE VM: `Accept` and `HTTPHello`. Both effects use non-blocking Unix system calls under the hood. `Accept` can accept an incoming network connection on port 8080 and returns the new port number that can

[3]Green threads are lightweight user space threads usually scheduled by runtime systems rather than the operating system.

be used for further communication with a client. The `HTTPHello` effect takes a port number and sends a "Hello World!" HTTP message using an already established connection.

Now we can implement two types of green threads: the function `start_server` implements the main thread of the web server: it accepts connections on the port 8080. If there is a new connection then a new green thread is spawned for that connection, otherwise control is given to other threads.

Listing 4.3: Main green thread of the web server

```
let start_server () =
  let rec run () =
    let port = perform Accept () in
    match port with
    (* Yield if no incoming connection *)
    | -1 -> perform Yield ()

    (* Spawn thread for new connection *)
    | _ -> perform Spawn (say_hello port);
    run ()
  in
    run ()
;;
```

Listing 4.4: Green thread handling a connection

```
let say_hello port =
  let rec run () =
    let error = perform (HTTPHello port) in

    match error with
    | 0 -> () (* On success *)
    | -1 -> () (* If connection closed *)

    (* Try again if not ready *)
    | _ -> perform Yield (); run ()

  in run
;;
```

The `say_hello` function takes a port number and *returns a green thread*: a `unit` → `unit` function that will send the "Hello World" message on the port given but only if the underlying socket is ready. If this is not the case it yields control and will retry later.

As effect handlers can simulate state it is possible to implement thread scheduling with handlers too. The web server thus can be started with something like `with thread_scheduler handle start_server ()`. Based on the work of Dolan et al. [8] I claimed in the Introduction that effect handlers give flexibility to programmers to implement their own concurrency models which suits their application the most while they reduce the complexity of runtime systems. This is a justification of that claim (it is actually a simplified experiment from their paper). We see that `thread_scheduler` could implement any kind of scheduling policy by handling the `Yield` and `Spawn` effects appropriately (the scheduler used in this evaluation can be found in [2]). The fact that one can implement a toy webserver using one's own runtime system written from scratch as a Part II project justifies the second claim (the cited paper is a result of the collaboration of 6 people).

I load tested this web server with the `wrk2` load testing tool [4] using 4 threads and gradually increasing the number of connections until the p90 time fell below the 5s threshold (this happened at around 100 connections as the red numbers indicate this on Figure 4.7).

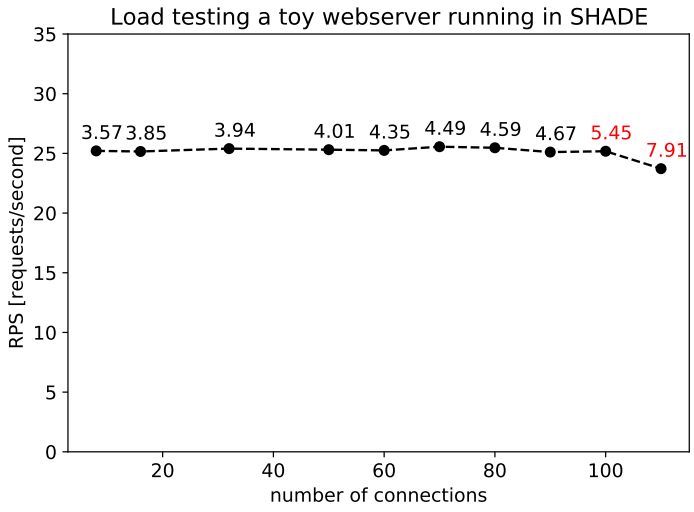


Figure 4.7: Although the performance of the webserver is not at all impressive, it is my own runtime system that implements it and it can still handle 100 connections with a p90 time of 5 seconds with a throughput of 25 requests per second.

4.5 Summary

The following sums up this chapter and reports its findings.

- The performance of the syntax-level CEK interpreter and the SHADE VM was compared to Eff and Multicore OCaml.
- My CEK interpreter was 20x quicker on the exception benchmarks and 10x quicker on the state benchmarks than the Eff interpreter. However, it was also 15x slower on both the exception and state benchmarks than the OCaml interpreter (the precise measurements results can be found in Appendix D).
- My SHADE VM was consistently quicker than all the interpreters and performed well on the exception (SHADE is 2% slower) and state benchmarks (SHADE is 17% quicker) against the VM of Multicore OCaml. However, it remained around 35x slower than the VM of Multicore OCaml on the N-queens benchmark.
- The reason for this slowdown on the N-queens benchmark was determined and it was found that the cause is inherent in Eff. Eff supports multi-shot continuations, whereas Multicore OCaml only supports one-shot continuations and therefore the copy overhead in the SHADE VM is bigger.
- As a qualitative evaluation of SHADE a toy webserver was presented and load tested.

5 | Conclusion

The project was a success. My success criterion was to implement a syntax-level interpreter for Eff, to design a low-level bytecode for Eff, to implement a compiler which can transform Eff programs into a linearised representation and finally to implement a virtual machine which is capable of interpreting the designed bytecode. Furthermore, the above components had to be reasonably efficient when compared to existing solutions.

The work described above was carried out and all components were successfully implemented. Both my CEK interpreter and my SHADE VM outperformed Eff on various benchmarks and the performance of SHADE VM was often comparable to that of Multicore OCaml.

Travel Diary This dissertation is actually a story about a journey which starts in the theoretical valleys of Mathematics, leads through the more practical (but still quite theoretical) areas of Computer Science and finally ends in the realms of practical Systems Programming.

This work was partly motivated by the desire to explore how algebraic effect handlers could be used in systems programming. The use of algebraic effect handlers is appealing as they provide a convenient abstraction for the handling of side effects functionally. However, efficient runtime systems often work with low-level bytecodes rather than with syntax-level representations of programs. Designing appropriate bytecodes and ways to realise algebraic effect handlers closer to the hardware is necessary if we are ever going to use them in the real world. The existence of Multicore OCaml and SHADE VM proves that this is possible. To show a real world systems programming application I exhibited a simple concurrent system (a toy webserver) which was executed in the runtime system I designed.

5.1 Further work

Optimisations The fact that there are algebraic theories behind algebraic effects and their handlers allows us to prove certain compiler transformations correct by equational reasoning. For instance, the following effect equation

$$\texttt{assign}(x, y); \texttt{assign}(x, z) = \texttt{assign}(x, z)$$

shows that the first `assign` can be optimised away. This gives us the opportunity to take a set of effect equations and generate new optimisations: perhaps entirely new ones no-one has thought about yet or ones that are dependent on the semantics of some effects.

Type and effect systems [5] allow for even more optimisations and the opportunity to parallelise independent parts of programs. As hardware is getting increasingly parallel such optimisations can help us to get the most out of our multicore machines.

Open source The project is packaged up as an OPAM package and I intend to make it available on Github for other programming language enthusiasts.

Part II project My implementation was concerned with only the core features of Eff. There would be countless ways to improve the compiler or the virtual machine. The rules of SHADE

VM are simple enough that a bytecode interpreter for SHADEcode should not be too hard to implement in a low-level language such as C. However, this would have to involve designing a custom garbage collector for SHADE which might prove interesting.

Masters or PhD As the size of continuations is not too big it might make sense to think about these control operators in the context of distributed systems too. For instance, imagine the following situation where the communication overhead in the network might be reduced.

Alice and Bob are communicating. Alice sends a message M_1 to Bob and Bob replies with M_2. Based on the contents of M_2 Alice performs some kind of computation c and replies with M_3. If c can be realised as a continuation resumed with M_2 as its argument and given that shadows and continuations can be serialised using a convenient bytecode representation, would it make sense to *package up a decision as a continuation* and attach it to the original message M_1? This would eliminate the need to exchange M_2 and M_3 over the network as Bob could resume the continuation using M_1 and obtain the result *locally*.

5.2 Self-reflection

If I were to start the project again I would try to be less ambitious. I vastly underestimated the intellectual challenge behind continuations, handlers and control operators.

At the beginning of the project I got a bit carried away with syntactic sugar and trying to implement a lot of features of the language properly. This proved to be counterproductive and I had to abandon these efforts. If I had to start the project again I would make the development even more iterative than it was: I would design a smaller minimal viable product first, build a very minimal but working (and tested!) pipeline and then iterate more on improvements. This was my original strategy when developing, but I could do it better the next time.

Closing thoughts Furthermore, I think there is a lack of elementary material concerning this topic on the web. I am hoping that my Preparation chapter and this dissertation as a whole will in some ways help popularise algebraic effects and their handlers and that readers of this document will find them less cryptic than I first did.

Bibliography

[1] Caml Virtual Machine — instruction set. `http://cadmium.x9c.fr/distrib/caml-instructions.pdf`. Accessed: 2020-04-25.

[2] Eff official website. `https://www.eff-lang.org/`. Accessed: 2020-01-10.

[3] OCaml official website. `https://ocaml.org/`. Accessed: 2020-03-17.

[4] wrk2, a constant throughput, correct latency recording variant of wrk. `https://github.com/defanator/wrk2`. Accessed: 2020-05-04.

[5] Andrej Bauer and Matija Pretnar. An effect system for algebraic effects and handlers. In *International Conference on Algebra and Coalgebra in Computer Science*, pages 1–16. Springer, 2013.

[6] Andrej Bauer and Matija Pretnar. Programming with algebraic effects and handlers. *Journal of logical and algebraic methods in programming*, 84(1):108–123, 2015.

[7] Kent Beck. *Test-driven development: by example*. Addison-Wesley Professional, 2003.

[8] Stephen Dolan, Spiros Eliopoulos, Daniel Hillerström, Anil Madhavapeddy, KC Sivaramakrishnan, and Leo White. Concurrent system programming with effect handlers. In *International Symposium on Trends in Functional Programming*, pages 98–117. Springer, 2017.

[9] Daniel Hillerström. *Compilation of effect handlers and their applications in concurrency*. PhD thesis, Master's thesis, School of Informatics, The University of Edinburgh, 2016.

[10] P. J. Landin. The Mechanical Evaluation of Expressions. *The Computer Journal*, 6(4):308–320, 01 1964.

[11] Xavier Leroy. The zinc experiment: an economical implementation of the ml language. 1990.

[12] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The definition of standard ML: revised*. MIT press, 1997.

[13] Eugenio Moggi. Computational lambda-calculus and monads. In *[1989] Proceedings. Fourth Annual Symposium on Logic in Computer Science*, pages 14–23. IEEE, 1989.

[14] Eugenio Moggi. *An abstract view of programming languages*. University of Edinburgh. Department of Computer Science. Laboratory for ..., 1990.

[15] Lawrence C Paulson and Larry C Paulson. *ML for the Working Programmer*. Cambridge University Press, 1996.

[16] Gordon Plotkin and John Power. Computational effects and operations: An overview. 2002.

[17] Gordon Plotkin and John Power. Algebraic operations and generic effects. *Applied categorical structures*, 11(1):69–94, 2003.

[18] Gordon Plotkin and Matija Pretnar. Handlers of algebraic effects. In *European Symposium on Programming*, pages 80–94. Springer, 2009.

Appendices

A | Theory

A.1 Semantics

$$\frac{}{(\textbf{if true then } c_1 \textbf{ else } c_2) \rightsquigarrow c_1}\ (\textsc{If-True}) \qquad \frac{}{(\textbf{if false then } c_1 \textbf{ else } c_2) \rightsquigarrow c_2}\ (\textsc{If-False})$$

$$\frac{}{(\textbf{fun } x \mapsto c)\ e \rightsquigarrow c[e/x]}\ (\textsc{Fun-App})$$

$$\frac{}{(\textbf{match } (\textbf{Left } e) \textbf{ with Left } x \mapsto c_1 \ \parallel\ \textbf{Right } x \mapsto c_2) \rightsquigarrow c_1[e/x]}\ (\textsc{Match-Left})$$

$$\frac{}{(\textbf{match } (\textbf{Right } e) \textbf{ with Left } x \mapsto c_1 \ \parallel\ \textbf{Right } x \mapsto c_2) \rightsquigarrow c_2[e/x]}\ (\textsc{Match-Right})$$

$$\frac{}{(\textbf{match } (e_1, e_2) \textbf{ with } (f, s) \mapsto c) \rightsquigarrow c[e_1/f, e_2/s]}\ (\textsc{Match-Prod})$$

$$\frac{c_1 \rightsquigarrow c_1'}{\textbf{let } x = c_1 \textbf{ in } c_2 \rightsquigarrow \textbf{let } x = c_1' \textbf{ in } c_2}\ (\textsc{Let-Step}) \qquad \frac{}{\textbf{let } x = (\textbf{val } e) \textbf{ in } c \rightsquigarrow c[e/x]}\ (\textsc{Let-Val})$$

$$\frac{}{\textbf{let } x = E(e, y.c_1) \textbf{ in } c_2 \rightsquigarrow E(e, y.\textbf{let } x = c_1 \textbf{ in } c_2)}\ (\textsc{Let-Effect})$$

$$\frac{}{(\textbf{let rec } f\ x = c_1 \textbf{ in } c_2) \rightsquigarrow c_2[(\textbf{fun } x \mapsto \textbf{let rec } f\ x = c_1 \textbf{ in } c_1)/f]}\ (\textsc{Let-Rec})$$

$$\frac{\kappa \text{ is current delimited continuation}}{\textbf{perform } (E\ e) \rightsquigarrow E(e, \kappa)}\ (\textsc{Perform})$$

$$\frac{c \rightsquigarrow c'}{\textbf{with } e \textbf{ handle } c \rightsquigarrow \textbf{with } e \textbf{ handle } c'}\ (\textsc{Handle-Step})$$

$$\frac{h \stackrel{\text{def}}{=} (\textbf{handler val } x \mapsto c_v \ \parallel\ \dots)}{\textbf{with } h \textbf{ handle } (\textbf{val } e) \rightsquigarrow c_v[e/x]}\ (\textsc{Handle-Val})$$

$$\frac{h \stackrel{\text{def}}{=} (\textbf{handler } \dots\ \overline{\parallel\ \textbf{effect } E_i\ x\ k \mapsto c_i}\ \parallel\ \dots) \qquad \exists i.E_i = E}{\textbf{with } h \textbf{ handle } E(e, y.c) \rightsquigarrow c_i[e/x, (y.c)/k]}\ (\textsc{Handle-Eff-Match})$$

$$\frac{h \stackrel{\text{def}}{=} (\textbf{handler } \dots\ \overline{\parallel\ \textbf{effect } E_i\ x\ k \mapsto c_i}\ \parallel\ \dots) \qquad \forall i.E_i \neq E}{\textbf{with } h \textbf{ handle } E(e, y.c) \rightsquigarrow E(e, y.\textbf{with } h \textbf{ handle } c)}\ (\textsc{Handle-Eff-Rise})$$

Figure A.1: Small step operational semantics of Eff

A.2 Typing

$$\frac{x : A \in \Gamma}{\Gamma \vdash_e x : A}\ (\text{T-Var}) \qquad \frac{}{\Gamma \vdash_e \textbf{true} : \texttt{bool}}\ (\text{T-True}) \qquad \frac{}{\Gamma \vdash_e \textbf{false} : \texttt{bool}}\ (\text{T-False})$$

$$\frac{}{\Gamma \vdash_e () : \texttt{unit}}\ (\text{T-Unit}) \qquad \frac{\Gamma \vdash_e e_1 : A \qquad \Gamma \vdash_e e_2 : B}{\Gamma \vdash_e (e_1, e_2) : A * B}\ (\text{T-Pair})$$

$$\frac{\Gamma \vdash_e e : A}{\Gamma \vdash_e \textbf{Left}\ e : A + B}\ (\text{T-SumLeft}) \qquad \frac{\Gamma \vdash_e e : B}{\Gamma \vdash_e \textbf{Right}\ e : A + B}\ (\text{T-SumRight})$$

$$\frac{x : A, \Gamma \vdash_c c : C}{\Gamma \vdash_e (\textbf{fun}\ x \mapsto c) : A \to C}\ (\text{T-Fun})$$

$$\frac{x : A, \Gamma \vdash_c c_v : C \qquad \forall i.\ e_i : A_i, k_i : B_i, \Gamma \vdash_c c_i : C \qquad x : C, \Gamma \vdash_c c_f : D}{\Gamma \vdash_e h : A \Rightarrow D}\ (\text{T-Handler})$$

where $h \stackrel{\text{def}}{=} (\textbf{handler val}\ x \mapsto c_v\ \overline{\parallel\ \textbf{effect}\ E_i\ e_i\ k_i \mapsto c_i}\ \parallel\ \textbf{finally}\ x \mapsto c_f)$ and $\forall i.\ E_i : A_i \twoheadrightarrow B_i \in \Sigma_E$

$$\frac{\Gamma \vdash_e e : A}{\Gamma \vdash_c \textbf{val}\ e : A}\ (\text{T-Val}) \qquad \frac{\Gamma \vdash_e e : \texttt{empty}}{\Gamma \vdash_c \textbf{absurd}\ e : A}\ (\text{T-Absurd})$$

$$\frac{\Gamma \vdash_c c_1 : A \qquad x : A, \Gamma \vdash_c c_2 : B}{\Gamma \vdash_c (\textbf{let}\ x = c_1\ \textbf{in}\ c_2) : B}\ (\text{T-Let})$$

$$\frac{x : A, f : A \to B \Gamma \vdash_c c_1 : B \qquad f : A \to B, \Gamma \vdash_c c_2 : C}{\Gamma \vdash_c (\textbf{let rec}\ f\ x = c_1\ \textbf{in}\ c_2) : C}\ (\text{T-LetRec})$$

$$\frac{\Gamma \vdash_e e_1 : A \to B \qquad \Gamma \vdash_e e_2 : A}{\Gamma \vdash_c e_1\ e_2 : B}\ (\text{T-FunApp})$$

$$\frac{\Gamma \vdash_e e : \texttt{bool} \qquad \Gamma \vdash_c c_1 : A \qquad \Gamma \vdash_c c_2 : A}{\Gamma \vdash_c \textbf{if}\ e\ \textbf{then}\ c_1\ \textbf{else}\ c_2 : A}\ (\text{T-If})$$

$$\frac{\Gamma \vdash_e e : A + B \qquad x : A, \Gamma \vdash_c c_l : C \qquad x : B, \Gamma \vdash_c c_r : C}{\textbf{match}\ e\ \textbf{with Left}\ x \mapsto c_l\ \parallel\ \textbf{Right}\ x \mapsto c_r : C}\ (\text{T-MatchSum})$$

$$\frac{\Gamma \vdash_e e : A * B \qquad f : A, s : B, \Gamma \vdash_c c_l : C}{\textbf{match}\ e\ \textbf{with}\ (f, s) \mapsto c : C}\ (\text{T-MatchProd})$$

$$\frac{E : A \twoheadrightarrow B \in \Sigma_E \qquad \Gamma \vdash_e e : A}{\textbf{perform}\ (E\ e) : B}\ (\text{T-Perform}) \qquad \frac{\Gamma \vdash_e e : A \Rightarrow B \qquad \Gamma \vdash_c c : A}{\textbf{with}\ e\ \textbf{handle}\ c : B}\ (\text{T-WithHandle})$$

Figure A.2: All typing rules for Eff

A.3 Compilation

$$[\![\textbf{Id } x]\!]^{\gamma}_{s} = \text{where\_is } (x, s, \gamma)$$

$$[\![\textbf{Const } c]\!]^{\gamma}_{s} = \textbf{const } c$$

$$[\![\textbf{Box } (t, v_1 \dots v_n)]\!]^{\gamma}_{s} = [\![v_n]\!]^{\gamma}_{s}; \textbf{push}; [\![v_{n-1}]\!]^{\gamma}_{s+1}; \textbf{push}; \dots [\![v_1]\!]^{\gamma}_{s+n-1}; \textbf{makebox } t, n$$

$$\underbrace{[\![\textbf{Fun } (x, body)]\!]^{\gamma}_{s}}_{f} = \text{fvs-to-stack}(s, \gamma, f); \textbf{makeclosure } N, L$$

$$\underbrace{[\![\textbf{Handler } (v, es, f)]\!]^{\gamma}_{s}}_{h} = \text{fvs-to-stack}(s, \gamma, h); \textbf{makehlosure } \dots$$

$$[\![\textbf{Return } v]\!]^{\gamma}_{s} = [\![v]\!]^{\gamma}_{s}$$

$$[\![\textbf{If } (v, e_1, e_2)]\!]^{\gamma}_{s} = \textbf{branchifnot } L_{\text{false}}; [\![e_1]\!]^{\gamma}_{s}; \textbf{jump } L_{\text{exit}}; \textbf{label } L_{\text{false}}; [\![e_2]\!]^{\gamma}_{s}; \textbf{label } L_{\text{exit}}$$

$$[\![\textbf{LetIn } (x, e_1, e_2)]\!]^{\gamma}_{s} = [\![e_1]\!]^{\gamma}_{s}; \textbf{push}; [\![e_2]\!]^{\gamma[x \mapsto s]}_{s+1}$$

$$[\![\textbf{LetRecIn } (f, x, body, e)]\!]^{\gamma}_{s} = \text{fvs-to-stack}(f, x, body, s, \gamma); \textbf{makeclosure } N, L; \textbf{push}; [\![e]\!]^{\gamma[f \mapsto s]}_{s+1}$$

$$[\![\textbf{Perform } (id, v)]\!] = [\![v]\!]^{\gamma}_{s}; \textbf{throw } id$$

$$[\![\textbf{WithHandle } (h, e)]\!]^{\gamma}_{s} = [\![h]\!]^{\gamma}_{s}; \textbf{push}; \text{fvs-to-stack}(e, s, \gamma); \textbf{makeclosure } N, L; \textbf{castshadow}; \textbf{fin}$$

$$[\![\textbf{BinOp } (\texttt{op}, v_1, v_2)]\!] = [\![v_1]\!]^{\gamma}_{s}; \textbf{push}; [\![v_2]\!]^{\gamma}_{s+1}; \text{instr-of}(\texttt{op})$$

$$[\![\textbf{FunApp } (v_1, v_2)]\!] = [\![v_1]\!]^{\gamma}_{s}; \textbf{push}; [\![v_2]\!]^{\gamma}_{s+1}; \textbf{apply}$$

$$[\![\textbf{GetField } (n, v)]\!]^{\gamma}_{s} = [\![v]\!]^{\gamma}_{s}; \textbf{getfield } n$$

$$[\![\textbf{ListHead } v]\!]^{\gamma}_{s} = [\![v]\!]^{\gamma}_{s}; \textbf{listhead}$$

$$[\![\textbf{ListTail } v]\!]^{\gamma}_{s} = [\![v]\!]^{\gamma}_{s}; \textbf{listtail}$$

Figure A.3: Compilation from IR to SHADE bytecode

B | Code snippets

Listing B.1: Unwinding of the K component in the CEK machine

```
(* int -> cek_value -> cek_k -> cek_k -> cek_state *)
let rec unwind eff_id eff_arg hlosure_frame_acc = function
| [] ->
    let msg = "Runtime exception: a handler handling effect with ID="
        ^ (string_of_int eff_id) ^ "could not be found" in
    raise (CEKerror msg)
| hf :: hfs ->
    let continuation = hlosure_frame_acc @ [hf] in

    (* Current hlosure frame matches effect *)
    if is_match eff_id hf then
        (* Obtain matching effect case *)
        let matching_effcase = List.find (fun effcase -> effcase.eff_id = eff_id
            ) hf.hlosure.handler.effcases in

        (* Bind argument of effect and the current continuation to the
            corresponding indentifiers in the hlosure environment *)
        let env = hf.hlosure.env
        |> (Cek_env.add matching_effcase.arg_label eff_arg)
        |> (Cek_env.add matching_effcase.kont_label (CEKcont continuation)) in

        (* Return the new CEK state *)
        { c = matching_effcase.eff_control;
          e = env;
          k = hfs; }

    (* If no match is found then continue the unwinding of hlosure frames *)
    else
        unwind eff_id eff_arg continuation hfs
    ;;
```

C | Software Engineering

Up – lib@574c0f3b95b6 » Lib » Cek

Module Lib.Cek

```
val apply : (Cek_types.cek_value * Cek_types.cek_value *
Cek_types.cek_state) -> Cek_types.cek_state
```

Function/continuation application

Entry point

These functions implement the entry point for the CEK machine.

```
val exec : Cek_types.cek_control list -> Cek_types.cek_value list
```

Given that an Eff program is a list of statements we execute these statements one by one. Note that for this to work the `run_cek` function must be side effecting because we must keep of toplevel `let` statements in a global environment

```
val run_cek : Cek_types.cek_control -> Cek_types.cek_value
```

Run a CEK program on a CEK machine. This function initialises a CEK machine that runs a statement. Every CEK machine started by `run_cek` starts with the environment the previous CEK machine stopped executing with.

```
val run : Cek_types.cek_state -> Cek_types.cek_value
```

Implements an evaluation loop.

```
val step : Cek_types.cek_state -> Cek_types.cek_state
```

Implements a single step of the Yo abstract machine.

Figure C.1: Writing documentation helps both with structuring a developer's own thoughts and helps new contributors accommodate to a new codebase

```
Testing Virtual Machine.
This run has ID `2D95521F-E043-4921-AF71-F7EA0A024884`.
[OK]                constants            0   unit.
[OK]                constants            1   bool.
[OK]                constants            2   int.
[OK]                constants            3   float.
[OK]                constants            4   string.
[OK]                expressions          0   identifier.
[OK]                expressions          1   tuple.
[OK]                expressions          2   list.
[OK]                expressions          3   empty list.
[OK]                expressions          4   variant.
[OK]                computations         0   if.
[OK]                computations         1   let.
[OK]                computations         2   let-in.
[OK]                computations         3   let-args.
[OK]                computations         4   let-rec.
[OK]                computations         5   let-pattern.
[OK]                computations         6   let-pattern-in.
[OK]                computations         7   function application.
[OK]                computations         8   continue.
[OK]                computations         9   perform.
[OK]                computations        10   with-handle.
[OK]                computations        11   valcase-fincase.
[OK]                computations        12   nondeterminism.
[OK]                computations        13   finally.
[OK]                computations        14   with-handle nesting.
[OK]                computations        15   with-handle with finally.
[OK]                computations        16   with-handle with effects.
[OK]                computations        17   queue.
[OK]                regression           0   forall + partial application.
[OK]                regression           1   filter.
[OK]                regression           2   list_length.
[OK]                regression           3   list_one_to_n.
[OK]                regression           4   fibonacci.
[OK]                regression           5   toplet_recursion.
The full test results are available in `/home/[redacted]/_build/_tests/2D955
21F-E043-4921-AF71-F7EA0A024884`.
Test Successful in 0.000s. 34 tests run.
```

Figure C.2: Tests help to guarantee some correctness properties of the software

```
HELLO on 56514
# received bytes=567
received=
GET / HTTP/1.1
Host: 127.0.0.1:8080
Connection: keep-alive
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/81.0.4044.113 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.9
Sec-Fetch-Site: none
Sec-Fetch-Mode: navigate
Sec-Fetch-User: ?1
Sec-Fetch-Dest: document
Accept-Encoding: gzip, deflate, br
Accept-Language: en-GB,en;q=0.9,hu-HU;q=0.8,hu;q=0.7,cs;q=0.6,de;q=0.5,sk;q=0.4,es;q=0.3
```

Figure C.3: Excerpt from the webserver's log

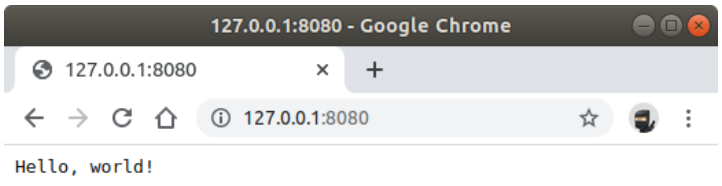


Figure C.4: The output of a Hello World webserver implemented using effect handlers in a web browser

D | Measurement results

Interpreter	500	1,000	1,500	2,000	2,500	3,000	3,500	4,000	4,500	5,000
Eff	0.439	1.871	4.491	8.322	13.705	20.437	28.355	38.094	49.978	63.145
CEK	0.03	0.108	0.228	0.425	0.698	1.048	1.526	2.076	2.683	3.546
MuliCaml	0.014	0.021	0.031	0.046	0.068	0.094	0.123	0.163	0.209	0.254

Table D.1: Exception benchmark (interpreters)

Bytecode	5k	10k	15k	20k	25k	30k	35k	40k	45k	50k
MultiCaml	0.146	0.686	1.782	3.658	6.64	11.202	16.988	25.028	35.172	45.164
SHADE	0.184	0.882	2.09	4.108	7.344	12.416	18.408	27.12	35.632	46.672

Table D.2: Exception benchmark (compilers)

Interpreter	1,000	3,000	5,000	7,500	10,000
Eff	1.783	20.325	63.755	160.697	299.495
CEK	0.164	1.726	5.702	14.474	29.278
MultiCaml	0.048	0.184	0.484	1.178	2.268

Table D.3: State benchmark (interpreters)

Bytecode	1,000	3,000	5,000	7,500	10,000	20,000
MultiCaml	0.01	0.061	0.202	0.513	1.267	5.936
SHADE	0.012	0.063	0.173	0.412	0.844	4.921

Table D.4: State benchmark (compilers)

Interpreter	8	10	12	14	16	18	20
Eff (interpreter)	0.062	0.074	0.222	1.994	13.042	67.914	406.662
MultiCaml (bytecode)	0.1	0.1	0.1	0.1	0.066	0.328	1.812
SHADE (bytecode)	0.018	0.014	0.04	0.328	2.214	11.598	65.446

Table D.5: N-queens benchmark (compilers)

Metric	8	10	12	14	16	18	20
Execution time [sec]	0.018	0.014	0.04	0.328	2.214	11.598	65.446
Unnecessary clones created	237	226	556	3,843	20,164	82,676	399,368
Memory used [Kbytes]	18,416	18,204	18,256	18,368	32,144	89,872	397,836

Table D.6: Relationship between the number of extra continuation cloning calls, memory consumption and execution time

Number of connections	Throughput [requests/sec]	p90 time [sec]
4	25.12	3.79
8	25.21	3.57
16	25.16	3.85
32	25.4	3.94
50	25.31	4,01
60	25.25	4.35
70	25.56	4.49
80	25.47	4.59
90	25.11	4.67
100	25.18	5.45
110	23.72	7.91

Table D.7: Results from load testing a toy webserver

E | Project Proposal

The project proposal starts on the next page.

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.

25^{th} October – 7^{th} November — 1^{st} 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.*

8^{th} November – 21^{st} November — 2^{nd} 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.*

22^{nd} November – 28^{th} 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.

29^{th} November – 12^{th} December — 3^{rd} 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.*

13^{th} December – 26^{th} December — 4^{th} work package

Implement the translator from parse trees to bytecode.

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

27^{th} December – 2^{nd} 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.