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.

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 [1] 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.

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 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 [5] 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.

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.

Listing 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 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.

[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.

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 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 $B \times \texttt{string}$ 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: 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 3.

Listing 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 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 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—`Print` is of type $\texttt{string} \twoheadrightarrow \texttt{unit}$).

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 $A \times \texttt{string}$. One might think of handlers as transformations on computations. The handler on Listing 2 is of type $\alpha \Rightarrow \alpha \times \texttt{string}$.

1.3.1 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 [2]. To see the use case of this feature, compare the following two code snippets in Listing 4 and Listing 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.

Listing 4: Handler without finally

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

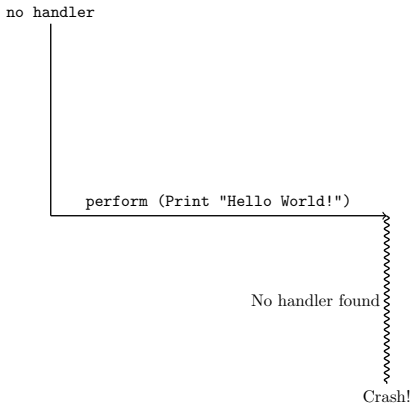
Listing 5: Handler with finally

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

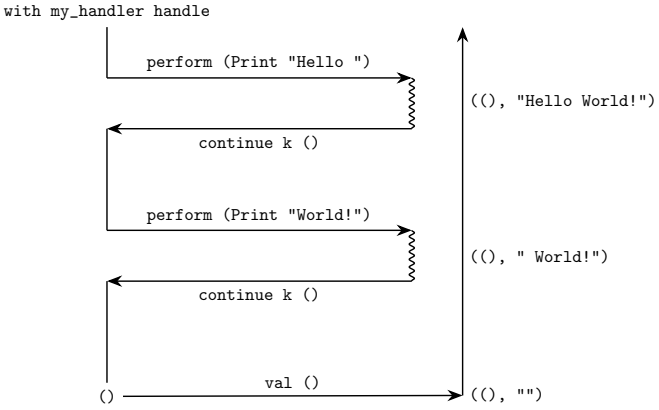
Instead of using a let-wrapper every time to convert to Fahrenheit we could use the same handler h but extend it with a finally rule that does this conversion and hence avoids code duplication.

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).

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 1: The tutorial illustrated

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.

2.1 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 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 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 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 6. A similar piece of code can be written with call/cc:

Listing 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.

2.3 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.*

Listing 8: In Eff continuations behave like functions

```
1  let x =
2    with prepending_handler handle
3      perform (Print "Hello");
4      perform (Print "World");
5      ()
6  in 42;;
```

In Listing 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] 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.

3 Eff formally

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

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

Computations

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

Figure 2: 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 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.

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 3). The other transition rules are standard and can be found in Appendix **??**.

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

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

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

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

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

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

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

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

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

Figure 3: Eff-specific rules of the small step operational semantics

The explanation of these rules is given by Table 1 so that the reader can quickly jump between the comments there and the rules on Figure 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 [6]) $E(e, \kappa)$ 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 1: Explanation of the Eff specific rules of the small step semantics

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 $E_i : A_i \twoheadrightarrow B_i$.

3.3.1 Type checking

Again, I omit standard typing rules and focus on the Eff specific ones. For the sake of completeness Appendix **??** 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.

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

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

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

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

Figure 4: Type checking for Eff-specific expressions and computations

Again, explanation of these rules is given in Table 2 in a similar way to how it was done for the semantics.

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: Explanation of the Eff specific typing rules

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 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
```

won't 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
```

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 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.

4 Abstract machines by example

A purist would look at rule (Perform) in Figure 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 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.

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 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.

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 $\langle C, E, K\rangle$.

Initialisation:

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

Transition rules:

$$\langle x, E, (y.t, E') :: K\rangle \to \langle E(x), E, K\rangle$$

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

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

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

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

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

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

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

Termination:

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

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

Figure 5: CEK machine transition rules for TL

We see at $(*)$ in Figure 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.

4.2 The SECD machine

SECD machines originate from Landin's famous *The mechanical evaluation of expressions* [4]. 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.

Initialisation:

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

Transition rules:

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

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

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

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

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

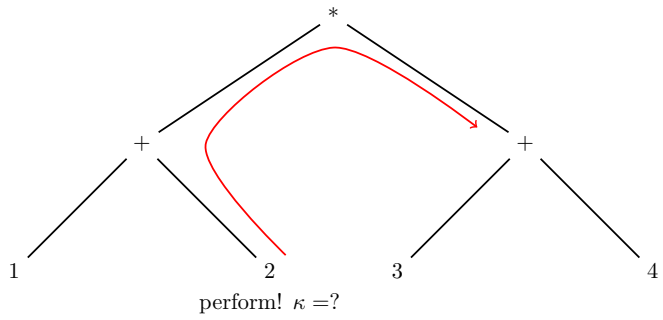
$$\langle v_2 :: v_1 :: S, E, \texttt{op} :: C, D \rangle \to \langle v :: S, E, C, D \rangle \text{ for } v = \texttt{op}(v_1, v_2)$$

Termination:

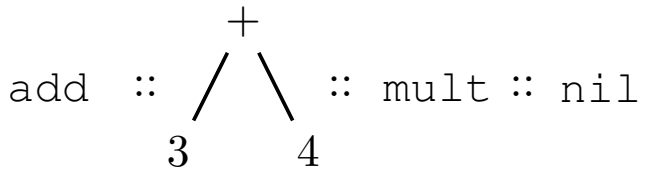
$$\langle v :: S, E, nil, nil \rangle \to v$$

Figure 6: SECD machine transition rules for TL

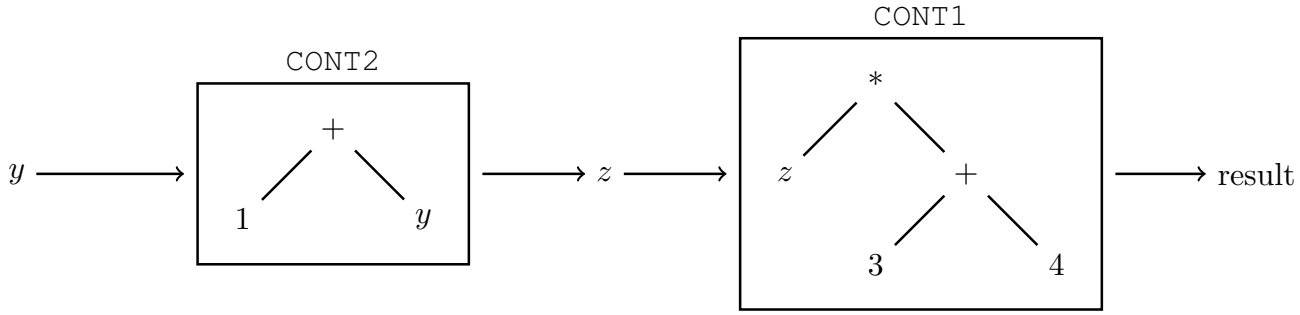
We see that although this machine works on abstract syntax trees, it is still a bit lower level than the CEK machine in that it makes a stack explicit and the way it processes (disassembles) the tree is very similar to compiling. In control, we end up with a list of smaller trees and SECD instructions.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



(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 7: Where is the continuation?

Figure 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 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!

5 Software engineering

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.

5.2 Starting point

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

- 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.

5.3 Development methodology

I chose *test-driven development* (TDD) [3] 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.

6 Summary

This section summed up the theoretical background which was needed to complete this project. It also shows the preparation I undertook in the software engineering aspects.

- Eff was introduced in a brief tutorial.
- It was explained how continuations represent arbitrary program points in the execution of a program. Therefore 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.

References

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

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

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

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

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

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