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

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.

1 A CEK machine for Eff

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

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 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 1: A handler parameterised by a free variable n

```
let make_handler n =
  let h = handler
  | val x -> x + n
  | effect (E x) k ->
    (* body of effect case using 'n' *)
  | finally x ->
    (* body of finally case using 'n' *)
;;
```

In Listing 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 $\mathcal{H}(H, E)$.

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 an 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 its 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 2: The structure of a CEK machine for Eff

```
type cek_id (** Identifiers *)
type cek_env (** Environments *)
type cek_control (** Control terms *)

(** Closures and hlosures *)
type cek_closure = cek_id * cek_env * cek_control
type cek_hlosure = cek_control * cek_env

(** Hlosure frames and the new K component *)
type cek_hlosure_frame = cek_hlosure * cek_kont list
type cek_k = cek_hlosure_frame list

(** CEK machine configuration *)
type cek_state = {
  c : cek_control;
  e : cek_env;
  k : cek_k;
}
```

I will call such a structure 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 1.

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 1).

1.3.1 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 $\mathcal{C}$.

1.3.2 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 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.

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 TODO)

Figure 1: A CEK machine interpreting Eff

1.3.3 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 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.

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

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

Figure 2: Unwinding of the K component in the CEK machine

An OCaml code snippet showing how to implement this same functionality can be found in Appendix **??**. 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 1. This can be seen in Listing 3.

Listing 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   }
```

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 byte code (SHADEcode, which is described in Table 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 2.2 will give an overview of how the byte code 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.

2.1 Transition rules and SHADEcode

Figure 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 [**?**]. I will not give formal transition rules for these here. Table 1 includes all SHADE instructions and explains their purpose in prose.

The SHADE machine is initialised by loading a program compiled to byte code 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.

[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.

Communication between shadows I will start to introduce the SHADE byte code and the SHADE transition rules. 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. Let us consider how with-handle blocks are compiled:

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

The handler is first compiled and is 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.

Initialisation:

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

Termination:

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

Transition rules for Eff-specific instructions:

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

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

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

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

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

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

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

Figure 3: SHADE machine transition rules

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 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.

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

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

Figure 4: Dump unwinding in SHADE

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.

Instruction	Description	Action
apply	Apply the top of the stack to the accumulator.	see Figure 3
ret	Return with the value in A. Restore the old program counter and environment from S.	see Figure 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
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
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
castshadow	Cast a new shadow.	see Figure 3
killshadow	Kill a shadow.	see Figure 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]];`
		`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 1: SHADEcode, the instruction set of the SHADE machine

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 **??**.

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 section I will show how the byte code 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 [**?**] machine with all effects behaving like exceptions. The next page contains various examples and should give a good insight into how everything fits together.

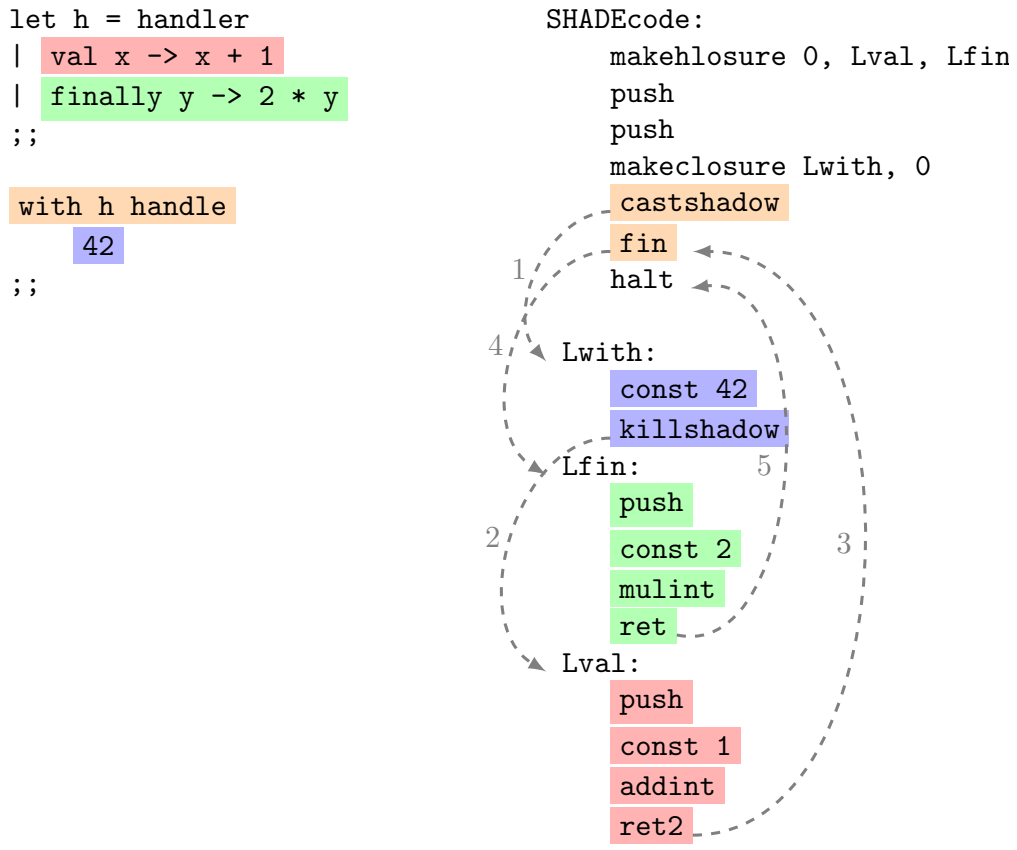


Figure 5: No effect is performed

Figure 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**.

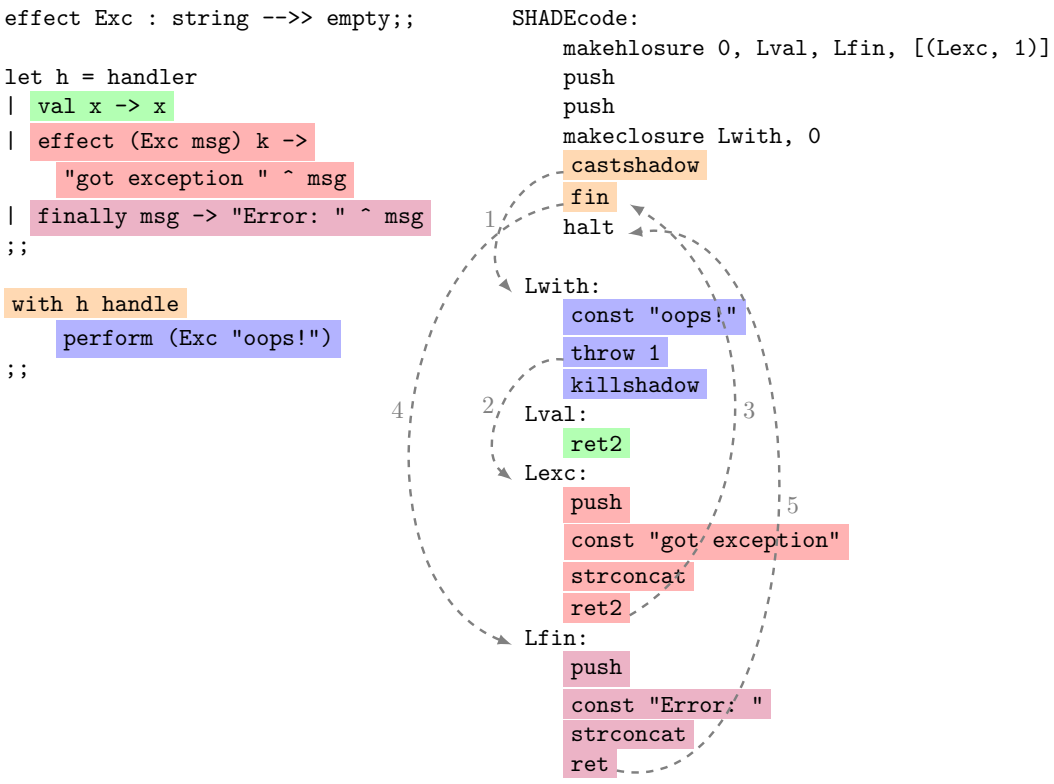


Figure 6: An effect is performed but its continuation is not resumed

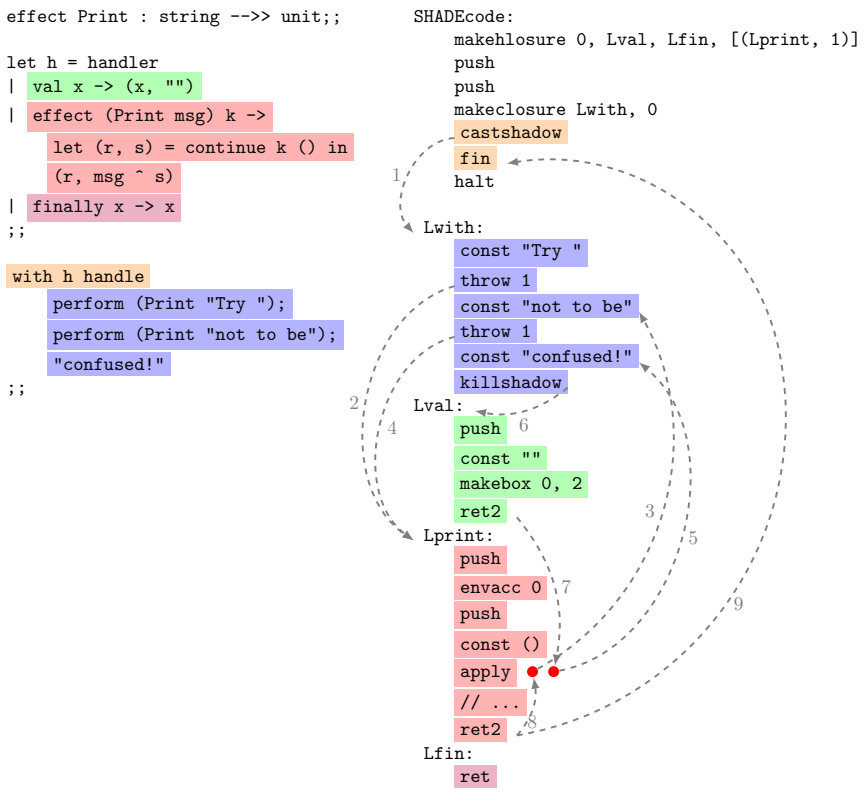


Figure 7: An effect is performed and its continuation is resumed

Figure 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 the **fin** instruction which is executed before the machine halts as we would expect. The result of the program is `86`.

Figure 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 program returns with the string `"Error: got exception oops!"`.

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 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. [3] Figure 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.

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

```
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
```

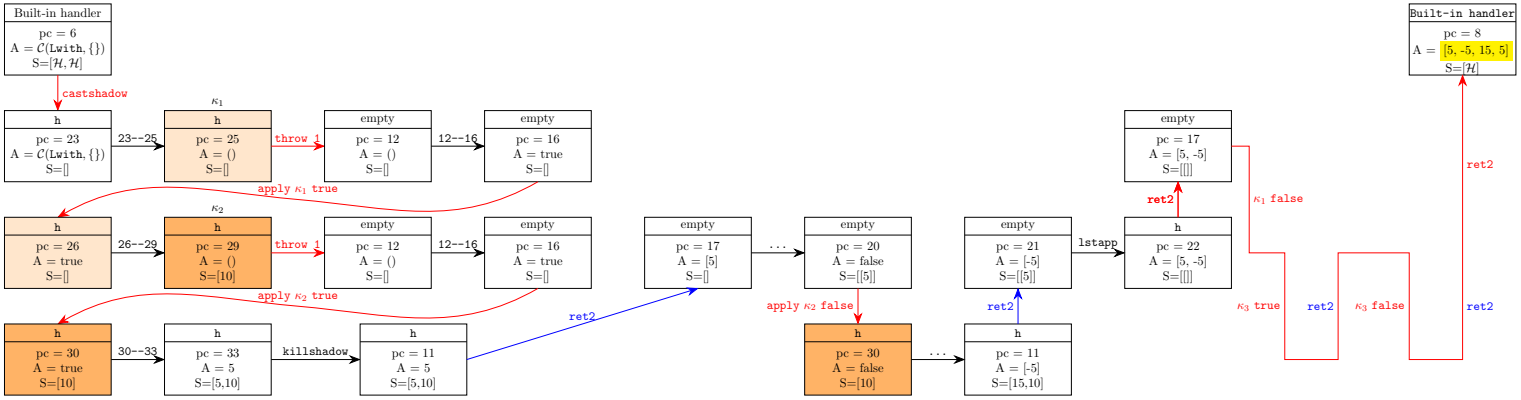


Figure 8: Non-determinism: multi-shot continuations can be resumed more than once in Eff

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 for correctness.

3 Software engineering

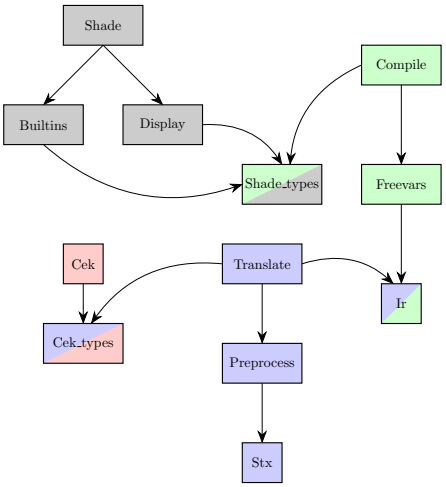
3.1 Requirements analysis and choice of tools

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`.

3.2 Repository structure

The repository of my OCaml implementation is structured as a typical OCaml package. A `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 4: SHADE VM's pipeline

```
open Core;;
open Lib;;

let main path_to_source =
    path_to_source
    |> Parser.parse_from_file
    |> Preprocess.convert
    |> Translate.stx_to_ir
    |> Compile.to_byte_code
    |> Shade.exec
    |> Display.pretty_vm_result
;;
```

Figure 9: Dependency graph of the modules residing in `lib` and the way they are used to build the SHADE VM pipeline

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
Preprocess	Implements preprocessing functionalities on ASTs. At this stage all identifiers are given unique integer De Bruijn indices and some syntactic sugars are 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 byte code from this representation.	77
Freevars	Implements analysis of free variables on the IR.	87
Compile	Implements the compilation from IR to SHADE byte code.	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 2: The modules of `lib`

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 data.

3.3 Techniques

Testing As stated in the Introduction correctness and hence testing were crucial for the success of this project. Hence 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. Appendix X contains a screenshot of a typical output Alcotest produces when testing the 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 [4] 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 other categories.

- *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/comments.
- *Benchmarking.* I used a separate branch for evaluation.

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

OPAM The project is packaged as an OCaml package. After further refactoring I intend to publish this project as an OCaml package.

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. 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.

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* byte code was explained and SHADE-code 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.