Implementation of a compiler and a virtual machine for a programming language with algebraic effects

(Implementacja kompilatora i maszyny wirtualnej
dla języka programowania z efektami algebraicznymi)

Konrad Werbliński

Praca magisterska

Promotor: dr hab. Dariusz Biernacki

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

30 maja 2022

Abstract

Algebraic effects gained a lot of traction in the functional programming community over the last decade. They provide a similar expressive power to monads, while maintaining a great composability. Efficient implementation of languages with algebraic effects is a subject of ongoing research.

In this thesis we present a virtual machine that implements a semantics for algebraic effects with instances. We also provide a compiler for the Helium language to the above-mentioned machine. Our implementation significantly improves Helium performance providing from 4 to 20 times speedup, but most importantly it provides a convenient test ground for algebraic effects compilation techniques.

W ciągu ostatniej dekady efekty algebraiczne zdobyły dużą popularność w świecie funkcyjnych języków programowania. Zapewniają one podobną moc wyrazu do monad, zachowując dobrą komponowalność. Obecnie prowadzone są prace nad ich efektywną implementacją.

W tej pracy przedstawiamy maszynę wirtualną implementującą semantykę efektów algebraicznych z instancjami oraz kompilator języka programowania Helium do kodu bajtowego wspomnianej maszyny. Nasza implementacja znacząco poprawia wydajność tego języka, oferując od 4 do 20 razy szybsze wykonywanie programów, równocześnie zapewniając praktyczne środowisko do testów technik kompilacji efektów algebraicznych.

Szczególne podziękowania dla dr Filipa Sieczkowskiego
za opiekę nad pracą

Contents

Chapter 1

Introduction

Algebraic Effects have been gaining a lot of popularity in the functional programming community over the recent years. They were initially introduced by Plotkin and Power [1] as a new formalization of effectful computations, and subsequently extended with a concept of handlers by Plotkin and Pretnar [2]. Later, Bauer and Pretnar [3] adopted algebraic effects and handlers as a practical way of programming. Thanks to their great composability, they are often viewed as an easier to use alternative for monads [4, 5]. Even tough distinct algebraic effects compose very easily, in their basic form it is not possible to combine multiple instances of the same effect (for example two different mutable memory cells). One of the first approaches to tackle this problem was implemented in the Eff language in the form of runtime created instances [3]. However Eff's instances were a purely dynamic construct and were not tracked by the type system. Subsequent approaches used row polymorphic types and recognized correct instances by their positions in the effect row [6, 7, 8]. These solutions, while well-behaved on the theoretical side, suffered in practicality requiring usage of a *lift* operation (also called *mask* or *inject*). Recent approach by Biernacki et al. [9] uses named instances bound by handlers and instance functions. This solution was incorporated in practice in the Helium programming language. However, up to this time Helium was interpreted in OCaml and lacked an efficient implementation. In this thesis we introduce the Helium Virtual Machine (HVM) and a new backend for the existing Helium infrastructure in a form of a compiler to HVM.

Virtual machines are a popular choice for high level programming language implementations. They can serve many different purposes and optimize different aspects of the language or the implementation process itself. They can be a simpler to implement alternative to a native code compiler like ZINC machine created for the Caml language [10]. They can provide an efficient runtime for the browser based applications like WebAssembly does [11]. Virtual machine may be used to implement scalable and reliable concurrent runtime environments like BEAM machine for the Erlang language. Finally, virtual machine's byte code can be used as a portable

intermediate representation for programs that are later compiled by a just-in-time compiler (e.g. JVM, .NET).

The Helium Virtual Machine is a stack-based machine implemented in C++. It provides support for algebraic effects with named instances. We made an unpopular design decision to use the automatic reference counting for memory management. We think that this approach significantly simplifies the implementation (which we believe is an important advantage for an experimental VM that is focused on algebraic effects not memory management) and yet it provides a quite satisfactory efficiency. Our compiler produces a highly optimized byte code. We put a lot of effort to clean up the abundance of a purely administrative code (e.g. type functions, module system, excessive let-bindings), but most importantly we implement a static optimization of curried functions that preserves the left to right order of evaluating arguments.

Our machine significantly improves performance of the Helium language (from 4 to 20 times speedup), but first of all it provides a reliable and convenient test ground for compilation and optimization techniques for algebraic effects (and instances). We tried to maintain a balance between high and low level concepts in our machine. It runs flat, instruction based programs, but consists of many handy abstractions like automatic memory management, closures and builtin support for algebraic effects with instances. This enables testing compilation techniques in the realistic setting without hurdles of dealing with low level details and quirks of real CPU architectures.

1.1 Related work

Aside from Helium, there exists a handful of experimental languages with algebraic effects. Algebraic effects as a practical programming construct were first featured in the Eff language [3]. Frank explores an interesting design space by eliminating the handler construct and instead enhances functional abstraction with an effect handling capability [8]. The Effekt language provides a different, lightweight approach to effect polymorphism, by treating effects as capabilities [12]. It also features a compiler to Java Script. Koka features a highly efficient compiler to C which uses the optimized reference counting for memory management [13, 14, 15, 16].

1.2 Structure of this thesis

In the further parts of this thesis we describe our work on the Helium Virtual Machine and the compiler:

- In Chapter 2 we give a brief practical introduction to algebraic effects and instances.

- In Chapter 3 we present a calculus and a formal semantics on which Helium is based on.
- In Chapter 4 we present extensive description of the virtual machine featuring description of the instruction set and the implementation details.
- In Chapter 5 we describe our compiler paying particular attention to optimizations.
- In Chapter 6 we evaluate performance of our machine based on a number of benchmarks.
- In Chapter 7 we summarize our work and discuss possible improvements and future work.

The source code of the Helium Virtual Machine can be found at:
`https://bitbucket.org/pl-uwr/helium-virtual-machine/src/main`

The HVM compiler is integrated into the official Helium implementation:
`https://bitbucket.org/pl-uwr/helium/src/master`

Chapter 2

Algebraic Effects

2.1 Introduction to Algebraic Effects

We provide a short practical introduction to algebraic effects using the Helium language. The easiest way to get a grasp of algebraic effects is to view them as generalized resumable exceptions. Thus, let's start our introduction with the example that shows how to express the exception handling.

2.1.1 Exceptions

First, we need to declare an effect. To declare the effect we provide a list of all operations that belong to that effect along with their types. Such effect definition is called `signature`. In the listing below we define the effect `Exception` with one operation `throw`. The operation takes a `String` with the error message as an argument and its return type is polymorphic, to allow us to raise it from any expression.

```
signature Exception =
| throw : {A : Type}, String => A
```

To perform the operation we simply call it as if it was an ordinary function.

```
let safeDiv x y =
  if y = 0 then
    throw "Division by zero error"
  else
    x / y
```

Similarly as we need to wrap exception throwing code with a `try ... with` block, we need to surround the effectful code with a `handle ... with` expression, which provides semantics for effectful operations. In the example below, if `safeDiv` uses the operation `throw`, we will print the error message. The `return` clause defines

what to do with a result of a handled expression when it is completely evaluated. In this case we will just print it.

```
let _ =
  let x = readInt () in
  let y = readInt () in
  handle
    safeDiv x y
  with
  | return r => printInt r
  | throw msg => printStr msg
  end
```

2.1.2 Reader

To demonstrate the resumable nature of algebraic effects we will take a look at the `Reader` effect. The `Reader` effect allows us to hide constant dependencies (for example a configuration) which otherwise had to be passed as arguments. It has only one operation `ask` which returns the value of a carried constant.

```
signature Reader (A : Type) =
| ask : Unit => A
```

As a simple demonstration of the `Reader` effect we can use it to carry around the g (standard gravity) constant while performing physical calculations.

```
(* computing weight: F =  m * g *)
let weight m =
  m * ask ()

let _ =
  handle
    printInt (weight 42)
  with
  | return x => x
  | ask () => resume 10
  end
```

2.1.3 Handlers as values

Handlers are first class values, thus we can avoid repeatedly writing the same handlers, by predefining common handlers or write a function that returns a handler. We can also omit the `return` clause if it just trivially returns a result.

```
let hReader v = handler
| ask () => resume v
end

let hEarth = hReader 10
let hSun = hReader 274

let _ =
  handle
    printInt (weight 42)
  with hSun
```

2.1.4 Nondeterminism

What may come as a surprise is the fact that we can resume more than once. In the example below, for each usage of the `flip` operation we perform two resumptions (one that returns `True` to the caller and one that returns `False`). Combining this with a `filter` function, we can print all sublists of a given list with a few simple lines of code.

```
signature Choice (S : Type) =
| flip : Unit => Bool

let _ =
  handle
    printIntList (filter (fn _ => flip ()) (range 15))
  with
  | flip () => resume True; resume False
  end
```

2.1.5 State

Expressing state is a bit tricky. However, implementation of the `State` effect is very similar to the state monad. Even though the definition of the state effect may be difficult to understand, a typical programmer does not need to know such details. As we showed earlier, effects are first class values, thus the definition of a handler for state can be predefined in the standard library.

```
signature State (S : Type) =
| put : S => Unit
| get : Unit => S
```

The handler for the `State` effect uses closures and their arguments to carry the memory cell. The first usage of either a `get` or `put` operation returns a function as a

result and exits the `handle` block. Then, the `finally` clause is executed and applies the initial state value to this function. The resumption in its body will evaluate to another function (which corresponds to the next `get` or `put` call or exiting the `handle` block through the `return` clause) and the current state value will be passed to it.

```
let hState init = handler
| return x  => fn _ => x
| put s     => fn _ => resume () s
| get ()    => fn s => resume s s
| finally f => f init
end
```

2.1.6 Combining effects

In contrast to monads, algebraic effect compose seamlessly. In the monadic approach combining two effects would require usage of the monad transformers [17], which have many drawbacks. First of all, they are hard to comprehend for the beginner functional programmers. They introduce abundance of `lift` function (this is a different `lift` than previously mentioned one in the context of other approaches to effect instances) usage and they only allow seamless composition of identical monad stacks. In the next listing we show how easy it is to combine `State` and `Exception` effects in Helium. All we need to do is to provide handlers for each effect that we use.

```
let _ =
  let x = readInt () in
  handle
    handle
      put (safeDiv x (get ()))
    with
    | return () => printInt (get ())
    | throw msg => printStr msg
    end
  with hState 5
```

2.2 Algebraic Effects with instances

What differentiates Helium from the majority of other languages with algebraic effects is the fact that we can use multiple instances of the same effect. For example, using two instances of the `State` effect we can have two separate, mutable memory cells. In the example below, the function `inc` takes an instance `i` as an argument to let the compiler know which handler will be handling the effect. As we can see, it is handled by two different handlers depending on the instance passed during a call.

```
let inc `i () =
  let n = get `i () in
  put `i (n + 1)

let _ =
  handle `a in
    handle `b in
      inc `a ();
      inc `b ();
      printInt (get `a ());
      printInt (get `b ())
    with hState 2
  with hState 1
```

It is worth noting that there is still ongoing work on improving and simplifying programmer experience with instances in Helium. Thus, the syntax presented above may change in future.

Chapter 3

Language semantics

Giving semantics for the programming language with algebraic effects with named instances is a non trivial task. The intuitive approach would be to treat handlers as binders and reduce accordingly. However, this approach leads to reductions under binders. Furthermore, bound variables can escape their scope and become globally free. Lets consider the following program: $\texttt{handle}_a\ (\texttt{fun}\ \_\ \_\ =\ \texttt{do}_a())\ \texttt{with}\ \{\dots\}$. It reduces in one step to the following term: $\texttt{fun}\ \_\ \_\ =\ \texttt{do}_a()$, in which the variable a has clearly escaped its context. Moreover, this semantics becomes unsound when the calculus is extended with universal quantifiers.

To tackle this problem, Helium semantics is based on generative semantics from Biernacki et al. [9]. Although we do not prove equivalence of programs compiled to HVM with this semantics, it was a starting point and inspiration for the machine's design. It is defined for a minimalistic calculus, which is a stripped down version of the Helium language.

3.1 Calculus syntax

Calculus syntax is formulated in the A-normal form [18], which is a typical choice for calculi with algebraic effects. To simplify the definition, each effect has only one operation called `do`, which is also not an uncommon formulation.

3.1.1 Values

Term level values are rather standard. Since Helium is a call-by-value language, variables are also classified as values. The only surprising thing is that there are three different abstractions. The existence of the type function may seem odd, since the calculus is untyped. However, the calculus is just an intermediate representation for the typed language. Thus, type abstractions ($\Lambda\, e$) are necessary for maintaining correct semantics. The instance functions ($\lambda\, a.\, e$) are used to allow existence

of effectful procedures, because at their definitions we cannot determine to which handlers they will be bound (they can also be bound to more than one handler, since they can be used in multiple places in our program).

$v ::=$

x	variable
$\mid$ `()`	unit
$\mid$ `fun` $f\, x.\, e$	function
$\mid \Lambda\, e$	type function
$\mid \lambda\, a.\, e$	instance function

3.1.2 Expressions

Expressions are also not surprising. Since we have three different abstractions, we also have three different applications. Each `do` expression is indexed with the instance variable. Instance variables are bound either by a `handle` expression or an instance function.

$e ::=$

v	value
$\mid$ `let` $x\ =\ e$ `in` e	let
$\mid v\, v$	application
$\mid v\, *$	type application
$\mid v\, a$	instance application
$\mid$ `do`$_a\, v$	effect call
$\mid$ `handle`$_a\ e$ `with` `{`h; r`}`	handle

3.1.3 Handler

Effect handler binds two values: the operation argument and the resumption.

$h ::= x,\, k.\, e$

3.1.4 Return clause

Unsurprisingly, the return clause binds one variable to a result of a corresponding `handle` expression.

$r ::= x.\, e$

3.2 Generative semantics

The generative semantics generates runtime instances that replace the binders instead of reducing under them. Runtime instances are implemented as freshly gen-

erated global labels, which ensures that complete programs always reduce closed expressions. For example, the program $\texttt{handle}_a\ (\texttt{fun}\_\_ = \texttt{do}_a())\ \texttt{with}\{\dots\}$ now evaluates in two steps. In the first step to: $\texttt{handle}_{l_1}\ (\texttt{fun}\_\_ = \texttt{do}_{l_1}())\ \texttt{with}\{\dots\}$. Finally, in the second step to: $\texttt{fun}\_\_ = \texttt{do}_{l_1}()$. It may seem that we still have a variable escaping its lexical scope, namely l_1. However, l_1 is not a variable, but a runtime instance in form of a label. Thus, there are no variables escaping their scope.

3.2.1 Evaluation contexts

We have three kinds of evaluation contexts: empty, let and handle. Let's take note that a `handle` expression creates a context only when the instance variable was substituted with a label. As we can observe in the reduction rules below, the handle expression with an instance variable is a redex that evaluates to the same expression but with a fresh label.

$E ::=$

$\square$	empty
$\mid E[\texttt{let}\, x\, =\, \square\, \texttt{in}\, e]$	let
$\mid E[\texttt{handle}_l\, \square\, \texttt{with}\, \{h;\, r\}]$	handle

3.2.2 Reification contexts

Reification contexts are used to express matching executed `do` expressions with the corresponding handlers and creating resumptions.

$R ::=$

$\square$	empty
$\mid \texttt{let}\, x\, =\, R\, \texttt{in}\, e$	let
$\mid \texttt{handle}_l\, R\, \texttt{with}\, \{h;\, r\}$	handle

3.2.3 Reduction rules

In Figure 3.1 we present reduction rules of the generative semantics. First four rules are standard for most call-by-value languages. Rule (5) describes exiting a `handle` expression when its subexpression in reduced to a value. This value is bound to a variable x and the return clause is executed. Rule (6) represents invoking an operation with a `do` expression. We search for the corresponding handler and create the resumption. Then, we enter the handler code with operation argument and resumption bound to variables x and k. Rule (7) depicts the generation of a fresh instance label. The final rule expresses the reduction step over an evaluation context.

$$\frac{}{(\texttt{fun}\, f\, x.\, e)\, v \mapsto e\{\texttt{fun}\, f\, x.\, e/f\}\{v/x\}}(1) \qquad \frac{}{(\Lambda\, e)\, * \mapsto e}(2)$$

$$\frac{}{(\lambda\, a.\, e)\, l \mapsto e\{l/a\}}(3) \qquad \frac{}{\texttt{let}\, x = v\, \texttt{in}\, e \mapsto e\{v/x\}}(4)$$

$$\frac{}{\texttt{handle}_l\ v\ \texttt{with}\ \{h;\ \texttt{return}\, x.\, e\} \mapsto e\{v/x\}}(5)$$

$$\frac{R = \texttt{handle}_l\ R'\ \texttt{with}\ \{x, k.\, e;\ r\} \qquad \text{free}(l, R')}{R[\texttt{do}_l\ v] \mapsto e\{v/x\}\{\lambda\, z.\, R[z]\ /\ k\}}(6)$$

$$\frac{\text{fresh}(l)}{E[\texttt{handle}_a\ e\ \texttt{with}\ \{h;\ r\}] \mapsto E[\texttt{handle}_l\ e\{l/a\}\ \texttt{with}\ \{h;\ r\}]}(7)$$

$$\frac{e_1 \mapsto e_2}{E[e_1] \mapsto E[e_2]}(8)$$

Figure 3.1: Generative semantics

Chapter 4

Virtual Machine

Helium Virtual Machine is a stack-based machine. Stack-based machines are a popular choice for a VM architecture. Most notable example from the industry would include ZINC [10], JVM, WebAssembly [11] or even Ethereum VM [19]. In fact, register-based machines also use stack for spilling and storing the return addresses of the called functions. Thus, we believe that a stack based architecture is a good choice for our needs.

Helium Virtual Machine shares many similarities with the ZINC machine created by Xavier Leroy for the CAML light language [10]. However, our implementation does not feature Push/Enter (nor Eval/Apply) mechanism, which optimizes usage of curried multi-argument functions [20]. We exclude those mechanisms, because they can change the order of effects (since Helium is an experimental language focused on effects, we want to be very strict about their order). Instead, our machine features multi-argument closures. This way the compiler can make optimized versions of the fully applied functions and statically decide when it is safe to use them.

Below we present the semantics of a slightly simplified version of the Helium Virtual Machine. The program P is represented as a list of instructions. The machine uses the accumulator a for passing results and arguments. It also features environment E and two stacks: the argument stack S_a and the return stack S_r. Finally, the machine configuration consists of the meta stack MS used for algebraic effects handling.

Let's start the presentation of the semantics with the two simplest rules. The `Const` instruction writes a constant value to the accumulator. The `Push` instruction puts the value of the accumulator at the top of the argument stack.

$$\langle \texttt{Const}\, v : P,\ a,\ E,\ S_a,\ S_r,\ MS \rangle_{eval} \Rightarrow \langle P,\ v,\ E,\ S_a,\ S_r,\ MS \rangle_{eval}$$

$$\langle \texttt{Push} : P,\ a,\ E,\ S_a,\ S_r,\ MS \rangle_{eval} \Rightarrow \langle P,\ a,\ E,\ a : S_a,\ S_r,\ MS \rangle_{eval}$$

We use De Bruijn indices to represent variables. `Let` expression creates a new variable with a value from the accumulator. The `AccessVar` instruction transitions the machine into the *var* configuration which preforms the environment lookup.

$$\langle \texttt{Let} : P,\ a,\ E,\ S_a,\ S_r,\ MS\rangle_{eval} \Rightarrow \langle P,\ a,\ a : E,\ S_a,\ S_r,\ MS\rangle_{eval}$$

$$\langle \texttt{EndLet} : P,\ a,\ v : E,\ S_a,\ S_r,\ MS\rangle_{eval} \Rightarrow \langle P,\ a,\ E,\ S_a,\ S_r,\ MS\rangle_{eval}$$

$$\langle \texttt{AccessVar}\, k : P,\ a,\ E,\ S_a,\ S_r,\ MS\rangle_{eval} \Rightarrow \langle k,\ E,\ P,\ E,\ S_a,\ S_r,\ MS\rangle_{var}$$

When a closure is created, a snapshot of the current environment is taken. The constructed closure is placed in the accumulator.

$$\langle \texttt{MakeClosure}(arity,\ P') : P,\ a,\ E,\ S_a,\ S_r,\ MS\rangle_{eval} \Rightarrow$$

$$\langle P,\ Closure(arity,\ E,\ P'),\ E,\ S_a,\ S_r,\ MS\rangle_{eval}$$

The recursive closure is just a syntactic marker that informs a `Call` instruction that it has to be handled differently than a normal closure.

$$\langle \texttt{MakeRecursiveClosure}(arity,\ P') : P,\ a,\ E,\ S_a,\ S_r,\ MS\rangle_{eval} \Rightarrow$$

$$\langle P,\ RecClosure(arity,\ E,\ P'),\ E,\ S_a,\ S_r,\ MS\rangle_{eval}$$

To avoid unnecessary copying of values, primitive operations take their second argument through the accumulator.

$$\langle \texttt{Prim} \oplus : P,\ v_2,\ E,\ v_1 : S_a,\ S_r,\ MS\rangle_{eval} \Rightarrow \langle P,\ v_1 [\![\oplus]\!] v_2,\ E,\ S_a,\ S_r,\ MS\rangle_{eval}$$

The `Call` instruction adds a return address to the return stack (comprising of the current environment and the next instructions). The environment and the instructions from the called closure are loaded. Afterwards, the machine enters the *app* configuration to load the arguments from the argument stack.

$$\langle \texttt{Call} : P,\ Closure(arity,\ E',\ P'),\ E,\ S_a,\ S_r,\ MS\rangle_{eval} \Rightarrow$$

$$\langle arity,\ P',\ E',\ S_a,\ \{P, E\} : S_r,\ MS\rangle_{app}$$

$$\langle \texttt{Call} : P,\ RecClosure(arity,\ E',\ P'),\ E,\ S_a,\ S_r,\ MS\rangle_{eval} \Rightarrow$$

$$\langle arity,\ P',\ RecClosure(arity,\ E',\ P') : E',\ S_a,\ \{P, E\} : S_r,\ MS\rangle_{app}$$

$$\langle \texttt{Return} : P,\ a,\ E,\ S_a,\ \{P', E'\} : S_r,\ MS\rangle_{eval} \Rightarrow \langle P',\ a,\ E',\ S_a,\ S_r,\ MS\rangle_{eval}$$

The `Handle` instruction generates a fresh instance label. This corresponds to the rule (7) of the semantics from Chapter 3. The new frame is pushed onto the meta

stack. It contains the instance label, a handler code, an environment and a stack with the return address added on top. The `EndHandle` instruction removes the frame from the meta stack when the evaluation of the handled expression is finished. This corresponds to the rule (5) from the generative semantics.

$$\langle \texttt{Handle}(P', H) : P,\ a,\ E,\ S_a,\ S_r,\ MS\rangle_{eval} \Rightarrow$$

$$\langle P',\ a,\ i : E,\ S_a,\ \bullet,\ (i,\ E,\ H,\ \{P, E\} : S_r) : MS\rangle_{eval} \text{ where fresh}(i)$$

$$\langle \texttt{EndHandle} : P,\ a,\ E,\ S_a,\ \bullet,\ (i,\ E',\ H,\ S_r) : MS\rangle_{eval} \Rightarrow$$

$$\langle P,\ a,\ E',\ S_a,\ S_r,\ MS\rangle_{eval}$$

The argument for the `Op` instruction is passed through the accumulator. The argument stack is required to be empty, because it is not captured by a resumption. All values that need to be preserved have to be added to the environment. The machine enters the *inst* configuration to find the corresponding handler instance.

$$\langle \texttt{Op}\, k : P,\ a,\ E,\ \bullet,\ S_r,\ MS\rangle_{eval} \Rightarrow \langle k,\ E,\ P,\ a,\ E,\ S_r,\ MS\rangle_{inst}$$

Since from the programmer's point of view a resumption is indistinguishable from a function, the `Call` instruction is also used for invoking resumptions. The only argument is passed through the argument stack. Afterwards, the machine enters the *resume* configuration to unwind the reified meta stack from the resumption.

$$\langle \texttt{Call} : P,\ \mathit{Resume}(RMS,\ E',\ P'),\ E,\ v : \bullet,\ S_r,\ MS\rangle_{eval} \Rightarrow$$

$$\langle RMS,\ MS,\ P',\ E',\ v,\ \{P, E\} : S_r\rangle_{resume}$$

The *var* configuration performs the environment lookup to find the value assigned to the variable. When the value is found, the machine enters the *eval* configuration and continues the evaluation of the program.

$$\langle 0,\ v : E,\ P,\ E',\ S_a,\ S_r,\ MS\rangle_{var} \Rightarrow \langle P,\ v,\ E',\ S_a,\ S_r,\ MS\rangle_{eval}$$

$$\langle k,\ v : E,\ P,\ E',\ S_a,\ S_r,\ MS\rangle_{var} \Rightarrow \langle k - 1,\ E,\ P,\ E',\ S_a,\ S_r,\ MS\rangle_{var}$$

The *app* configuration moves function arguments from the stack to the closure environment. To eliminate the need for an extra buffer, the argument order is reversed.

$$\langle 0,\ P,\ E,\ S_a,\ S_r,\ MS\rangle_{app} \Rightarrow \langle P,\ (),\ E,\ S_a,\ S_r,\ MS\rangle_{eval}$$

$$\langle \mathit{arity},\ P,\ E,\ v : S_a,\ S_r,\ MS\rangle_{app} \Rightarrow \langle \mathit{arity} - 1,\ P,\ v : E,\ S_a,\ S_r,\ MS\rangle_{app}$$

The *inst* configuration performs the environment lookup to find the corresponding instance label. Afterwards, the machine enters the *op* configuration.

$$\langle 0,\ i : E,\ P,\ a,\ E',\ S_r,\ MS\rangle_{inst} \Rightarrow \langle i,\ MS,\ \bullet,\ a,\ P,\ E',\ S_r\rangle_{op}$$

$$\langle k,\ v : E,\ P,\ a,\ E',\ S_r,\ MS\rangle_{inst} \Rightarrow \langle k-1,\ E,\ P,\ a,\ E',\ S_r,\ MS\rangle_{inst}$$

The *op* configuration searches through the meta stack to find the frame with the matching instance label. When the frame is found the machine begins evaluating the handler code.

$$\langle i,\ (i,\ E',\ H,\ S'_r) : MS,\ RMS,\ a,\ P,\ E,\ S_r\rangle_{op} \Rightarrow$$

$$\langle H,\ (),\ a : Resume((i,\ E',\ H,\ S_r) : RMS, E, P) : E',\ \bullet,\ S'_r,\ MS\rangle_{eval}$$

$$\langle i,\ (i',\ E',\ H,\ S'_r) : MS,\ RMS,\ a,\ P,\ E,\ S_r\rangle_{op} \Rightarrow$$

$$\langle i,\ MS,\ (i',\ E',\ H,\ S_r) : RMS,\ a,\ P,\ E,\ S'_r\rangle_{op} \text{ where } i \neq i'$$

The *resume* configuration takes the reified meta stack and unwinds it frame by frame onto the current meta stack. When we get to the end of the reified meta stack, the machine enters the *eval* configuration (with the resumption argument placed in the accumulator).

$$\langle (i,\ E',\ H,\ S'_r) : \bullet,\ MS,\ P,\ E,\ v,\ S_r\rangle_{resume} \Rightarrow$$

$$\langle P,\ v,\ E,\ \bullet,\ S'_r,\ (i,\ E',\ H,\ S_r) : MS\rangle_{eval}$$

$$\langle (i,\ E',\ H,\ S'_r) : RMS,\ MS,\ P,\ E,\ v,\ S_r\rangle_{resume} \Rightarrow$$

$$\langle RMS,\ (i,\ E',\ H,\ S_r) : MS,\ P,\ E,\ v,\ S'_r\rangle_{resume}$$

As we mentioned at the beginning of this chapter this is a slightly simplified version of the Helium Virtual Machine. The actual byte code is flat and uses instruction pointers and jumps instead of nested subprograms (we chose a structured syntax for this presentation for the sake of readability). Moreover, the real HVM uses a more complex approach to recursive closures to allow mutual recursion (more on this in section 4.1.2). The presented semantics is a selection of most important and interesting instructions. Our machine also features instructions for creation of tuples and constructors and supports shallow pattern matching. It also has instructions that provide optimized versions of certain often emerging code patterns (more on this in section 4.2). HVM also provides the extern call mechanism to allow implementation of input and output.

4.1 Implementation details

Helium Virtual Machine is implemented in C++. We chose C++ over C to take advantage of the higher level of abstraction and to speed up the development time, while maintaining efficiency comparable to C. C++ is a popular choice for implementation of virtual machines, for example Google's V8 engine for Java Script is implemented in C++.

4.1.1 Data structures

Since the environment and the return stack are captured in resumptions (environments are also captured in closures), we implement them as persistent lists. Immutable data structures enable us to share them instead of copying, which makes such captures inexpensive. Argument stack is not captured in resumptions, thus it is represented as `std::vector`, which enables us to efficiently implement the `Push` instruction. Meta stack is also represented as `std::vector`, because it is never shared.

4.1.2 Memory management

Helium Virtual Machine uses automatic reference counting (ARC) for memory management. Reference counting is by far not the most popular design choice. The vast majority of programming languages uses either a garbage collector or manual memory management. However, there are some exceptions to this rule that use automatic reference counting and offer a very competitive results. Most notable examples being the Swift programming language [21] used in the Apple ecosystem and Koka - an experimental language with algebraic effects [15, 16].

Our main motivation for using ARC was simplifying the implementation and speeding up the development process. In the last chapter of this thesis we discuss possible improvements and different solutions for memory management in HVM.

To make the reference counting work properly we need to ensure that there are no cycles in the memory graph. All values in Helium are immutable (we have the `State` effect to provide the mutable memory cell functionality), hence the only entities that can create a loop are recursive closures (however, we are able to break these cycles with weak references). Thus, we are able to make the reference counting mechanism invisible to the programmer (and also to the compiler's front-end).

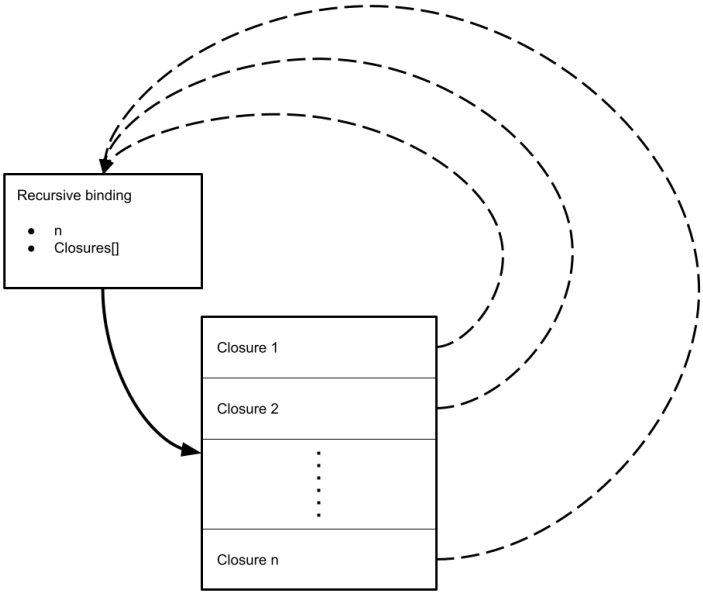
Recursive definitions

Let's consider a very important detail. Helium allows mutually recursive definitions, like in the simple example below:

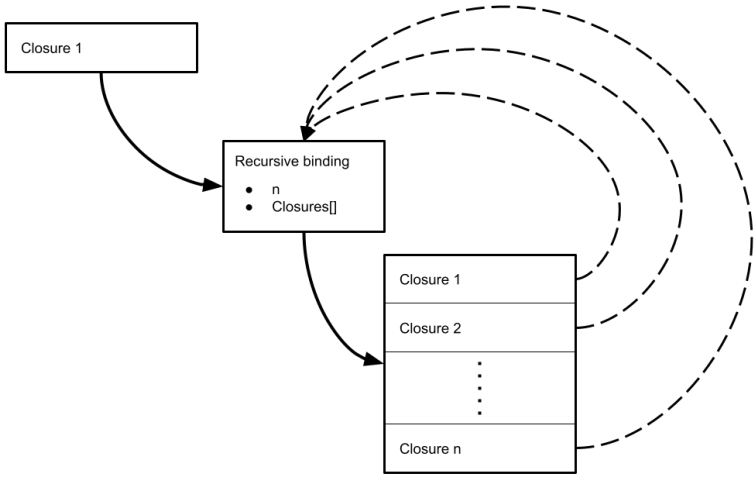
```
let rec even n =
  if n = 0 then True
           else odd (n - 1)
and odd n =
  if n = 0 then False
           else even (n - 1)
```

This means that we not only need to break a cycle in the memory graph, but also ensure the same lifetimes of the mutually recursive functions. To accomplish this,

the actual implementation of the machine, instead of the `MakeRecursiveClosure` instruction, features the `MakeRecursiveBinding` and `AccessRecursiveBinding` instructions. The recursive binding is a container in which we put all functions defined in the mutually recursive definition and which ensures their equal lifetime. Each function in the recursive binding has a weak reference to the container. Thus, no cycle in the memory graph is created.



When the `AccessRecursiveBinding` instruction extracts the closure from the container, the weak reference in that closure is changed into a strong one. This way we ensure that the recursive binding container will be allocated until it or any closure extracted from it remains in scope. For instance, if the binding presented in the above figure is stored in the accumulator, by executing the `AccessRecursiveBinding 1` instruction we will get:



When the closure from the recursive binding is called the reference to the binding is inserted to the function environment (similarly like the reference to the recursive closure is inserted to the environment in the simplified machine semantics presented in the previous section).

4.2 Improving machine's performance

The biggest drawback of building a virtual machine relying on persistent data structures is an added overhead related to memory allocations. To minimize negative performance impact of immutable data structures, we introduce a range of specialized instructions that allow us to reduce the number of allocations of stack frames and environment entries.

4.2.1 Tail call optimization

Optimizing tail calls is a standard practice while compiling functional languages. Helium Virtual Machine features a `TailCall` instruction, which saves us from the necessity of allocating a return address on the stack. A pleasant convenience of the virtual machine approach is how easy it is to perform the tail call optimization. All we need to do is to rewrite every `Call, Return` sequence that we encounter into a `TailCall` instruction.

4.2.2 Global definitions

Top level functions and definitions have a static lifetime: they remain allocated for the entire run time of a program. Quick access to top level functions is crucial. Consider a large library with dozens of functions. Our approach to represent environments as lists is aimed at small local contexts, thus working with large libraries could introduce noticeable overhead. To prevent that from happening, our virtual machine allows for allocation of global variables. They are allocated in `std::vector`, thus providing a constant time access. Global definitions also improve performance of recursive functions, since, by declaring them as global, we do not need to pass a recursive binding while performing a call. A support for global definitions is realized through the two instructions: `CreateGlobal` and `AccessGlobal`.

4.2.3 Specialized arithmetic instructions

Our machine features specialized versions of arithmetic instructions for the situations when one of the arguments is known at compile time (i.e. `AddConst`, `SubConst`, `SubFromConst`, etc.). Since the remaining argument is passed through the accumulator, such specialized instruction does not require any stack allocations.

4.2.4 Specialized call instructions

The standard `Call` instruction uses the accumulator for storing the called function. This means that all arguments must be passed through the argument stack. However, in some scenarios a function could be called without passing it through the accumulator. For example, when we call a function that is stored in the environment, we could pass the DeBruijn index of that function as an instruction parameter. Thus, instead of two instructions: `AccessVar 2; Call`, we could use one specialized instruction: `CallLocal 2`.

We also provide a similar specialized instruction for calling functions stored as global definitions and instruction for calling closures from recursive bindings stored in the local environment. For instance, the sequence of three instructions: `AccessVar 2; AccessRecursiveBinding 1; Call`, can be rewritten into just one instruction: `CallRecursiveBinding 2 1`.

Similarly to specialized versions of arithmetic instructions, this allows us to pass the last argument of a function through the accumulator and saves us a stack allocation.

Chapter 5

Compilation

The source for our compilation procedure is the intermediate language called *core*. It is an extended version of the calculus shown in Chapter 3. The compilation process uses two intermediate languages. The first one is called *untyped* and it is a simplified version of the *core* language stripped down from the last traces of the type system. It features multi-argument functions. It also moves away from the A-normal form. Moving away from the A-normal form may seem counterintuitive, but inlining expressions into arguments exactly corresponds to direct stack allocations. Below, we present syntax of the *untyped* language:

$e ::=$

x	variable
$\mid \texttt{()}$	unit
$\mid \texttt{let}\, x\, =\, e\, \texttt{in}\, e$	let
$\mid \texttt{fun}\, [x^*].\, e$	function
$\mid \underline{\texttt{fun}}\, [x^*].\, e$	reducible function
$\mid \texttt{fix}\, f\, [x^*].\, e$	recursive function
$\mid e\, [e^*]$	application
$\mid \texttt{do}_a\, e$	effect call
$\mid \texttt{handle}_a\, e\, \texttt{with}\, \{h;\, r\}$	handle

$h ::= x,\, k.\, e$

$r ::= x.\, e$

The second intermediate representation is the abstract version of the virtual machine code (similar to the byte code shown in Chapter 4). The actual machine code is completely flat and instead of nested instructions uses instruction pointers and jumps. We omit the translation from the abstract to the actual machine code, since it is trivial and not particularly interesting.

In the next two sections we present the compilation process. First, we describe the translations between the *core*, *untyped* and the abstract machine code. In the

later section, we describe optimizations performed on the *untyped* language and the abstract machine code.

5.1 Translation

5.1.1 From *core* to *untyped*

The translation process is in fact just a simple embedding of *core* into the *untyped* language. This means that the produced code is highly suboptimal. In order to produce efficient code we perform extensive optimizations described in detail in a later section. The translation preserves the A-normal form of the *core* language, thus it keeps the effects order unchanged.

$$[\![x]\!]_v = x$$

$$[\![()]\!]_v = ()$$

$$[\![\texttt{let}\ x = e_1\ \texttt{in}\ e_2]\!]_e = \texttt{let}\ x = [\![e_1]\!]_e\ \texttt{in}\ [\![e_2]\!]_e$$

$$[\![\texttt{do}_a\ v]\!]_e = \texttt{do}_a\ [\![v]\!]_v$$

$$[\![\texttt{handle}_a\ e\ \texttt{with}\ \{x,\ k.\ e_h;\ \ y.\ e_r\}]\!]_e = \texttt{handle}_a\ [\![e]\!]_e\ \texttt{with}\ \{x,\ k.\ [\![e_h]\!]_e;\ \ y.\ [\![e_r]\!]_e\}$$

The most interesting part of the translation concerns the treatment of functions. In the *untyped* language functions take a list of arguments instead of a single argument. The single arguments of the *core* functions are translated into singleton lists. The multi-argument nature of the *untyped* closures allows for intuitive embedding of the type abstractions as zero-argument functions (marked as reducible).

$$[\![\texttt{fun}\ f\ x.\ e]\!]_v = \texttt{fix}\ f\ [x].\ [\![e]\!]_e$$

$$[\![\Lambda\ e]\!]_v = \underline{\texttt{fun}}\ [\,].\ [\![e]\!]_e$$

$$[\![\lambda\, a.\ e]\!]_v = \texttt{fun}\ [a].\ [\![e]\!]_e$$

Similarly to functions, applications in the *untyped* language pass multiple arguments. This also enables natural embedding of the type applications as zero-argument calls.

$$[\![v_1\ v_2]\!]_e = [\![v_1]\!]_v\ [\,[\![v_2]\!]_v\,]$$

$$[\![v\ *]\!]_e = [\![v]\!]_v\ [\,]$$

$$[\![v\ a]\!]_e = [\![v]\!]_v\ [a]$$

5.1.2 From *untyped* to abstract byte code

The translation from *untyped* to the abstract machine code is rather unsurprising. We substitute variables with their corresponding DeBruijn indices. The argument passing during function calls is performed through the argument stack, thus we need to insert explicit stack pushes.

$$[\![()]\!] = \texttt{Const}\,0$$

$$[\![x]\!] = \texttt{AccessVar}\,\mathit{DeBruijn}(x)$$

$$[\![\texttt{let}\,x\;=\;e_1\;\texttt{in}\,e_2]\!] = [\![e_1]\!];\,\texttt{Let};\,[\![e_2]\!];\,\texttt{EndLet}$$

$$[\![\texttt{fun}\,[x_1, x_2, \ldots, x_n].\,e]\!] = \texttt{MakeClosure}(n,\,[\![e]\!])$$

$$[\![\texttt{fix}\,f\,[x_1, x_2, \ldots, x_n].\,e]\!] = \texttt{MakeRecursiveClosure}(n,\,[\![e]\!])$$

$$[\![e_f\,[e_1, e_2, \ldots, e_n]]\!] = [\![e_1]\!];\,\texttt{Push};\,[\![e_2]\!];\,\texttt{Push};\ldots;\,\texttt{Push};[\![e_n]\!];\,\texttt{Push};\,[\![e_f]\!]$$

$$[\![\texttt{do}_a\,e]\!] = [\![e]\!];\,\texttt{Op}\,a$$

$$[\![\texttt{handle}_a\;e\;\texttt{with}\;\{x,\,k.\,e_h;\;\;y.\,e_r\}]\!] =$$

$$\texttt{Handle}([\![e]\!];\,\texttt{EndHandle};\,\texttt{Let};[\![e_r]\!];\,\texttt{EndLet};\,\texttt{Return},$$

$$[\![e_h]\!];\texttt{Return})$$

5.2 Optimization

5.2.1 Optimization on the *untyped* language

The code produced by the translation from the *core* language is highly suboptimal. It remains in the A-normal form, thus it features abundance of `let` expressions which perform very costly allocations of new variables. To address this problem we perform an extensive range of optimizations described in the following subsections. Aside from lambda lifting, which is performed once at the end, all of the later discussed transformations are iterated multiple times.

Inlining

To move away from the A-normal form, we perform extensive inlining. However, one must be careful with inlining because it can lead to code size explosion or performance overheads. Thus, we enforce strict rules to define which expressions can be inlined:

- Any pure expression that is used exactly once.

- Any (possibly impure) expression that is used exactly once and inlining it will not change the effect order. For example, the expression `let x = e in do`$_a$ `x` can be safely rewritten to `do`$_a$ `e`.
- Literals.
- Any reducible function, but only at the call site. For example, let's assume that `f` is a single-argument reducible function. Then, `f` will be inlined in the expression `f [5]`, but will not be inlined in the expression `let g = f in g`.

Beta reduction

We beta-reduce every applied reducible function. The combination of inlining and beta reduction is able to remove most of the type function embeddings and transform applications of multi-argument functions.

Transformation of the multi-argument function definitions

To allow transformation of applications of multi-argument curried functions, we rewrite their definitions. For example, consider the following curried function:

```
let foo = fun [x]. fun [y]. e1
in ...
```

The above function `foo` is rewritten into a reducible wrapper over the two argument function `foo'`:

```
let foo' = fun [x, y]. e1 in
let foo = fun [x]. fun [y]. foo' [x, y]
in ...
```

Too see how the above transformation works in practice let's consider an example of the function `foo` application resembling the code obtained straight from translation of the *core* language.

```
let r = foo [4] in
r [5]
```

Thanks to the transformation, the function `foo` can now be inlined.

```
let r = (fun [x]. fun [y]. foo' [x, y]) [4] in
r [5]
```

Next, the inlined function is beta reduced.

```
let r = (fun [y]. foo' [4, y]) in
r [5]
```

Another iteration of inlining and beta reduction leaves us with the desired expression `foo' [4, 5]`.

5.2.2 Specialization

Even though the combination of inlining and beta reduction is able to transform multi-argument functions in most situations, there are some cases that would remain untouched. For example, consider the following code which uses the `foldl` function over a list. The `add` function will not be inlined, because it is not applied. The function `add` would become applied, if the `foldl` function was inlined and beta-reduced. However, according to our rules, the `foldl` will most likely not be inlined, since it is not reducible and probably used many times in the program.

```
let add =
fun [x]. fun [y]. x + y
in
foldl [add, 0, list_of_ints]
```

To prevent such situations from blocking the optimization, we identify such cases where a multi- argument reducible function is passed as a first argument (this pattern matches typical higher order functions like `fold`, `map`, `filter` etc.) and we inline both applied and passed functions at the call site. Next, we beta-reduce the applied function (but only substituting the first argument).

Finding functions eligible for reduction

In the *core* language constructors and operation calls are wrapped in lambda functions to allow their partial application. We scan for such wrappers and mark them as reducible.

Let hoisting

We hoist the nested let expressions. This simplifies expressions bound to variables which allows for a greater degree of inlining.

```
let x =
  let y = e1 in e2
in e3
```

The above expression is translated into the code below. In the rewritten code the expression bound to the variable x is simpler, thus it has a greater chance of being classified as eligible for inlining.

```
let y = e1 in
let x = e2 in
e3
```

Eta reduction

We perform the eta reduction on functions and on let-expressions. For example, the code below will be simplified to just `f`.

```
let y =
  fun [x]. f x
in y
```

Removal of unused pure expressions

We remove unused pure expressions bound by lets. We do not treat non termination as en effect. Hence, it is possible that, on a rare occasion, a pure looping expression will be deleted. However, the non terminating expression without any effects is most certainly a programmer error, thus we believe that it can be treated as an undefined behaviour and removed.

Lambda lifting

At the end of the optimization chain on the untyped language we perform the lambda lifting. We lift functions that do not have any free variables in their bodies. After translation to the abstract byte code, lifted expressions are represented as global definitions.

Correctness of transformations

To ensure correctness of the optimization process, we need to make sure that we do not change the execution order of effectful expressions. Code before transformations is in the A-normal form and keeps the same effect order as a corresponding *core* program. The inlined expressions are either pure or inlining them keeps the same effect order. However, one could worry that an unfortunate beta reduction could change the order of effects. But, thanks to our inlining rules it is impossible to inline an impure expression as an argument in the function application. Thus, beta reduction only substitutes parameters with pure expressions or variables.

5.2.3 Optimization on the abstract byte code

Optimizations on byte code are much simpler in their nature and based on the recognition of syntactic patterns.

Tail call optimization

Optimizing tail calls is trivial. All we need to do is to rewrite every `Call, Return` sequence we encounter to a `TailCall` instruction. However, this pattern is often obscured by `EndLet` instructions. To mitigate this problem we remove every `EndLet` placed directly before `Return`. The removal of `EndLet` instructions does not disrupt the correctness of a program because the `Return` instruction disposes of the whole current environment. Moreover, even if there is no tail call to be optimized, the `EndLet` removal is very useful optimization on its own.

Optimizing accumulator usage

We optimize usage of the accumulator by removing unnecessary reads and pure overwrites with the same value. For instance, let's take a look at the following listing.

```
Const 5;
Const 5;
Let;
AccessVar 0;
Push;
...
```

The second `Const` instruction writes the value 5 to the accumulator, but the accumulator already stores the value 5. The `Let` instruction does not change the content of the accumulator, hence the `AccessVar 0` instruction is redundant. Thus, the above code is transformed into the program below.

```
Const 5;
Let;
Push;
...
```

Removal of unused let expressions

Similarly to the optimization on the *untyped* language, we prune unused let expressions. However, we do not remove the bound expression itself, but only the let binding. Thus, we do not need to take the purity into account.

Chapter 6

Benchmarks

To present capability and efficacy of our virtual machine, we conducted a number of benchmarks. We divide our tests into two groups. In first one we measure efficiency of basic operations like arithmetic, function calls, recursion and data structure creation. In the second group we test the performance of algebraic effects. We present our benchmark sets in the figures at the end of this chapter. Presented test cases are written in Helium, for other languages we tried to implement them as close as possible to these programs. However, we were forced to make some subtle changes due to technical reasons. For example, for some Python programs we use loops to simulate tail call optimization (which is not present in CPython) and for some Effekt programs we use loops to avoid stack overflow. Whenever possible we use library functions to measure the most realistic performance of the language. We perform all of our tests on linux (Ubuntu 20) notebook with Intel i5-8265U CPU and 8GB of RAM.

6.1 Basic performance benchmarks

We compare interpreted Helium, Helium Virtual Machine (the same program as in the interpreted case, but compiled with our compiler), Python (3.8.10), Haskell (GHC 8.6.5, compiled with the `-O2` flag), Frank (interpreted), Effekt (compiled to JavaScript) and Koka (compiled to C with the `-O3` flag). In Figures from 6.1 to 6.5 we present programs that we use as basic performance benchmarks.

In the table below we present the results of our benchmarks. As we can see our virtual machine performance sits right in the middle between the interpreted and compiled languages, being around 10 to 20 times faster than interpreted Helium and Frank and around 10 to 20 times slower than Haskell and Koka. We can observe that HVM performs quite comparably to Python (CPython is also implemented as a byte code interpreter), which we believe is a very satisfactory result for such an experimental implementation. Moreover, HVM is notably faster than Effekt.

Language \ Benchmark	**EvenOdd**	**Fib**	**Foldr**	**Foldl**	**Tree**
Helium	4,419 s	7,982 s	8,463 s	5,759 s	5,654 s
HVM	0,187 s	0,695 s	0,824 s	0,498 s	1,022 s
Python	0,179 s [1]	0,555 s	0,103 s [2]	0,103 s [2]	4,416 s
Haskell	0,009 s	0,030 s	0,110 s	0,042 s	0,091 s
Frank	15,364 s	7,526 s	24,127 s	25,047 s	9,920 s
Effekt	2,074 s	1,237 s	2,473 s [3]	2,473 s [1]	1,618 s
Koka	0,013 s	0,037 s	0,082 s	0,057 s	0,046 s

6.2 Effects performance benchmarks

To measure the performance of algebraic effects we run three benchmarks: reader, state and combination of reader and state. We compare interpreted Helium, Helium Virtual Machine, Frank, Effekt, Koka and Haskell with Monads and Extensible Effects library. In Figures from 6.6 to 6.8 we present the set of the effects benchmarks. Koka and Effekt do not allow capture of resumptions in closures. However, they feature mutable local variables. Thus, for these languages, we implement the handler for the state effect using a local mutable variable.

In the table below we present the results of the second group of benchmarks. Similarly to the first set we can see that Helium Virtual Machine performance sits in the middle between the interpreted languages and Haskell. However, the gap to Koka (especially in the **State** benchmark) is a bit bigger. It is worth noting, that part of the Koka performance is a result of implementation of the state handler using a mutable variable. Moreover, Koka features a special syntax to mark tail recursive handlers and implement them as functions (we use it in our benchmarks because otherwise program exceeds the call stack limit). Similarly to the first group, HVM is from 2 to 3 times faster than the Effekt language.

Language \ Benchmark	**Reader**	**State**	**ReaderState**
Helium	3,584 s	5,587 s	7,177 s
HVM	0,401 s	0,940 s	1,352 s
Haskell + Monads[4]	0,035 s	0,177 s	0,212 s
Haskell + EE[5]	0,228 s	0,190 s	0,275 s
Frank	11,060 s	12,749 s	20,783 s
Effekt[1]	1,738 s	2,890 s	3,684 s
Koka	0,026 s	0,035 s	0,075 s

[1] Using a loop, because of the lack of support for tail call optimization.
[2] Using reduce.
[3] Using a loop, because of the call stack limit.
[4] Using Control.Monad.Reader and Control.Monad.State.
[5] Using Control.Eff.Reader.Lazy and Control.Eff.State.Lazy.

```
let rec even n =
  if n = 0 then True
           else odd (n - 1)
and odd n =
  if n = 0 then False
           else even (n - 1)

let _ =  even 4222223
```

Figure 6.1: **EvenOdd** - Mutual recursion of two simple functions.

```
let rec fib n =
  if n < 2 then
    n
  else
    fib (n - 1) +
    fib (n - 2)

let _ = printInt (fib 32)
```

Figure 6.2: **Fib** - calculating fibonacci numbers with the recursive definition.

```
let _ =
  printInt
    (foldr (+) 0
      (range 1000000))
```

Figure 6.3: **Foldr** - Sum of the list elements using foldr.

```
let _ =
  printInt
    (foldl (+) 0
      (range 1000000))
```

Figure 6.4: **Foldl** - Sum of the list elements using foldl.

```
data rec Tree =
| Leaf
| Node of Tree, Tree

let rec mkTree n =
  if n = 0 then Leaf
  else
    Node (mkTree (n - 1)) (mkTree (n - 1))

let rec count t =
  match t with
  | Leaf => 0
  | Node l r => 1 + count l + count r
  end

let _ = printInt (count (mkTree 20))
```

Figure 6.5: **Tree** - Creation and traversal of the full and complete binary tree.

```
let rec benchmark `a n acc =
  if n = 0 then acc
  else
    benchmark (n - 1)
              (acc + ask())

let _ =
  handle
    printInt
      (benchmark 1000000 42)
  with
  | ask () => resume 1
  end
```

Figure 6.6: **Reader** - reader effect benchmark.

```
let rec benchmark `a n =
  if n = 0 then get ()
  else
    (let _ =
      put (get () + 1) in
    benchmark (n - 1))

let _ =
  printInt (
    handle
      benchmark 1000000
    with hState 42)
```

Figure 6.7: **State** - state effect benchmark.

```
let rec benchmark `a `b n =
  if n = 0 then get `a ()
  else
    (let _ = put `a (get `a () + ask `b ()) in
    benchmark `a `b (n - 1))

let _ =
  printInt (
    handle `r in
      handle `s in
        benchmark `s `r 1000000
      with hState 42
    with
    | ask () => resume 2
    end)
```

Figure 6.8: **ReaderState** - benchmark of combining the reader and state effects.

Chapter 7

Discussion and future work

We believe that the virtual machine and the compiler are a valuable contribution into the Helium project. HVM significantly improves the performance of the language. We achieve efficiency similar to Python which we think is a very satisfactory result for an experimental VM with such a simple memory management model. Our compiler produces highly optimized output comparable in performance to a hand written byte code. We believe that it could be a good starting point for native code compiler and that lessons learned through the work on our implementation could prove very useful for the further development of the Helium language.

We think that further work on this project could produce another valuable contributions to the Helium language. Thus, in the next sections we present our ideas for improvements and future work.

7.1 Register machine

We decided to use a stack-based machine to simplify the compilation process by avoiding the register coloring. However, a register-based machine could be a great middle point on a path towards the native code compiler. Nevertheless, register-based architectures use stack for spilling and storing the return addresses, thus we think that our machine could be a good starting point for such development.

7.2 More advanced memory management

We chose the automatic reference counting for memory management to simplify and speed up the implementation process. Hence, it could be worth investigating more complex solutions to that problem.

7.2.1 Garbage collector

Garbage collectors are a standard approach to memory management in functional and object oriented programming languages. We could either implement our own GC or try to incorporate an existing collector for C/C++ like Google's Oilpan used in V8 [22].

7.2.2 Optimized reference counting

Koka uses the optimized reference counting algorithm called Perceus [15, 16]. It features a sophisticated reuse analysis and allows in-place updates at runtime, which enables implementing in-place algorithms in a purely functional way. Moreover, it offers a very competitive performance. We believe that including a similar technique in Helium Virtual Machine could prove very beneficial and possibly more efficient than a traditional garbage collector.

7.3 Loading extern calls from Dynamic-link libraries

One of the most important factors deciding about the success of a programming language are libraries and frameworks. Loading extern calls from DLLs would allow to easily create bindings for popular libraries. Algebraic effects still mostly remain a research subject (especially new concepts like instances), thus design patterns, best practices and guidelines for them are yet to be formulated. Providing a tooling for easy library creation, could help to test algebraic effects in many practical scenarios.

7.4 Optimizations of algebraic effects

At the moment we do not perform any algebraic effects specific optimizations. We think that this is an area with a lot potential for experiments and improvements. For example, we could detect the tail resumptive handlers with the instance binding known at the compile time and replace them with the ordinary functions.

Bibliography

[1] Gordon Plotkin and John Power. Algebraic operations and generic effects. *Applied Categorical Structures*, 11:69–94, 02 2003.

[2] Gordon D. Plotkin and Matija Pretnar. Handlers of algebraic effects. In Giuseppe Castagna, editor, *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, volume 5502 of *Lecture Notes in Computer Science*, pages 80–94. Springer, 2009.

[3] Andrej Bauer and Matija Pretnar. Programming with algebraic effects and handlers. *CoRR*, abs/1203.1539, 2012.

[4] Philip Wadler. Comprehending monads. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, LFP '90, page 61–78, New York, NY, USA, 1990. Association for Computing Machinery.

[5] Philip Wadler. Monads for functional programming. In Manfred Broy, editor, *Program Design Calculi*, pages 233–264, Berlin, Heidelberg, 1993. Springer Berlin Heidelberg.

[6] Daan Leijen. Koka: Programming with row polymorphic effect types. In Paul Blain Levy and Neel Krishnaswami, editors, *Proceedings 5th Workshop on Mathematically Structured Functional Programming, MSFP@ETAPS 2014, Grenoble, France, 12 April 2014*, volume 153 of *EPTCS*, pages 100–126, 2014.

[7] Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. Abstracting algebraic effects. *Proc. ACM Program. Lang.*, 3(POPL), jan 2019.

[8] Lukas Convent, Sam Lindley, Conor McBride, and Craig McLaughlin. Doo bee doo bee doo. *J. Funct. Program.*, 30:e9, 2020.

[9] Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. Binders by day, labels by night: Effect instances via lexically scoped handlers. *Proc. ACM Program. Lang.*, 4(POPL), dec 2019.

[10] Xavier Leroy. The ZINC experiment : an economical implementation of the ML language. Technical Report RT-0117, INRIA, February 1990.

[11] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the web up to speed with webassembly. *SIGPLAN Not.*, 52(6):185–200, jun 2017.

[12] Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. Effects as capabilities: Effect handlers and lightweight effect polymorphism. *Proc. ACM Program. Lang.*, 4(OOPSLA), nov 2020.

[13] Daan Leijen. Koka: Programming with row polymorphic effect types. In Paul Blain Levy and Neel Krishnaswami, editors, *Proceedings 5th Workshop on Mathematically Structured Functional Programming, MSFP@ETAPS 2014, Grenoble, France, 12 April 2014*, volume 153 of *EPTCS*, pages 100–126, 2014.

[14] Ningning Xie and Daan Leijen. Generalized evidence passing for effect handlers: efficient compilation of effect handlers to C. *Proc. ACM Program. Lang.*, 5(ICFP):1–30, 2021.

[15] Alex Reinking, Ningning Xie, Leonardo de Moura, and Daan Leijen. Perceus: Garbage free reference counting with reuse. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, page 96–111, New York, NY, USA, 2021. Association for Computing Machinery.

[16] Anton Lorenzen and Daan Leijen. Reference counting with frame limited reuse (extended version, v2). Technical Report MSR-TR-2021-30, Microsoft, November 2021. Mar 15, 2022, v2.

[17] Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, page 333–343, New York, NY, USA, 1995. Association for Computing Machinery.

[18] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. *SIGPLAN Not.*, 28(6):237–247, jun 1993.

[19] `https://ethereum.org/en/developers/docs/evm/`.

[20] Simon Marlow and Simon Peyton Jones. Making a fast curry: Push/enter vs. eval/apply for higher-order languages. In *Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming*, ICFP '04, page 4–15, New York, NY, USA, 2004. Association for Computing Machinery.

[21] `https://docs.swift.org/swift-book/LanguageGuide/AutomaticReferenceCounting.html`.

[22] `https://v8.dev/blog/high-performance-cpp-gc`.