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

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

1 Motivation

The motivations for this project are:

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

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

2 Work carried out

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

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

3 History and related work

Roughly speaking, *computational effects* are effects like I/O, state, non-determinism and various forms of jumps [**?**]. The way that these effects interact with each other and how we can safely deal

with them has always formed a great research interest, as the improper use of side-effects is often responsible for software bugs.

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

3.1 Theory: from maths to the programming language Eff

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

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

It turns out that effects like I/O, state and non-determinism are all algebraic. However, perhaps not surprisingly, not all effects are. Notably, the *handle* or *catch* operations—used to handle exceptions in most conventional programming languages—are a good example, these cannot be expressed with algebraic theories.

In fact, these effects are exactly the duals of algebraic effects. This means that if we think about performing an effect as *producing* an effect, then the handle or catch operations can be seen as the *consumers* of such effects. The literature commonly talks about *effect constructors* and *effect destructors* in this respect.

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

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

3.2 Practice: effect handlers in the real world

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

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

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