Honoursprogramme: Research track, option A (9 ECTS)

Efficient Compilation of Algebraic Effects and Handlers in Eff

Student: Axel Faes
Promotor: Prof. dr. ir. Tom Schrijvers
Daily supervisors: Prof. dr. ir. Tom Schrijvers &
Amr Hany Shehata Saleh

Master in de ingenieurswetenschappen:
computerwetenschappen
Specialisatie: Artificial Intelligence
Fase 1

September 2016 - March 2017

Contents

1 Description

1.1 Introduction

My honoursproject was part of the C1 project *Algebraic Effect Handlers: Harnessing the Fundamental Powers of Effects* [1]. During my honoursproject I had to work on the Eff programming language. More specifically, I had to create optimizations within the compiler in order to optimize away effect handlers. More specifically, my honoursproject can be broken down in the following steps:

1. Literature study of Eff and existing optimizations
2. Designing and implement new optimizations
3. evaluation: Eff versus OCaml
4. evaluation: Eff versus Other Systems

As we are working with both OCaml and Eff, whose syntax closely follows OCaml, colors are used to distinguish between `OCaml code` and `Eff code`. A paper was written towards the end of my honoursproject. The paper has been submitted and is currently in review. The abstract of the paper reads as follows: [2]

> The popularity of algebraic effect handlers as a programming language feature for user-defined computational effects is steadily growing. Yet, even though efficient runtime representations have already been studied, most handler-based programs are still much slower than hand-written code.
>
> In this paper we show that the performance gap can be drastically narrowed (in some cases even closed) by means of type-and-effect directed optimising compilation. Our approach consists of two stages. Firstly, we combine elementary source-to-source transformations with judicious function specialisation in order to aggressively reduce handler applications. Secondly, we show how to elaborate the source language into a handler-less target language in a way that incurs no overhead for pure computations.
>
> This work comes with a practical implementation: an optimizing compiler from Eff, an ML style language with algebraic effect handlers, to OCaml. Experimental evaluation with this implementation demonstrates that in a number of benchmarks, our approach eliminates much of the overhead of handlers and yields competitive performance with hand-written OCaml code.

1.2 Overview of Eff

Eff is a ML style functional programming language that uses algebraic effect handlers [3]. These handlers can be used to easily implement I/O, non-determinism or backtracking. Eff compiles down to OCaml code. The naive translation for effect handlers is very slow compared to a native implementation in OCaml or an implementation in Multicore OCaml [4]. Below is (part of) the naive translation of the N-queens program.

```
type (_, _) effect += Effect_decide : (unit, bool) effect
type (_, _) effect += Effect_fail : (unit, 'empty) effect

let decide x = call Effect_decide () value
let fail () = call Effect_fail () (fun _ -> assert false)

let choose_all = handler {
  value_clause = (fun x -> [x]);
  effect_clauses = fun (type a) (type b) (eff : (a, b) effect) -> (
    match eff with
    | Effect_decide -> fun _ k ->
```

```
        (k true @ k false)
    | Effect_fail -> fun _ _ ->
        []
    :
    a -> (b -> _) -> _
  )
}

let queens number_of_queens =
  let rec place (x, qs) =
    if x > number_of_queens then value qs else
      choose (available (number_of_queens, x, qs)) >>
      fun y -> (place (x + 1, (x, y) :: qs))
  in
  place (1, [])

let queens_all number_of_queens =
  choose_all (queens number_of_queens)
```

The native (hand-written) code looks as follows:

```
let queens_all number_of_queens =
  let rec place (x, qs) =
    if x > number_of_queens then [qs] else
      let rec choose = function
        | [] -> []
        | y :: ys ->
            place ((x + 1), ((x, y) :: qs)) @ choose ys
      in
      choose (available (number_of_queens, x, qs))
  in
  place (1, [])
```

The Eff code looks as follows:

```
effect Decide : unit -> bool
effect Fail : unit -> empty

let choose_all = handler
  | val x -> [x]
  | #Decide _ k -> k true @ k false
  | #Fail _ _ -> []

let queens number_of_queens =
  let rec place (x, qs) =
    if x > number_of_queens then qs else
      let y = choose (available (number_of_queens, x, qs)) in
      place ((x + 1), ((x, y) :: qs))
  in
  place (1, [])

let queens_all number_of_queens =
  with choose_all handle queens number_of_queens
```

The naive compilation is 100 times slower compared to the native code. Through the optimised compilation, the produced code becomes as fast the the native code. An advantage to the Eff code is that we can make the queens function perform differently depending on how the effects are implemented. This compared to the native version which is different for each implementation. It can be seen that in the naive translation, there is still a seperate datastructure for the effect handler. This means that each time an effect occurs, it has to go through the effect handler.

1.3 Literature study

The honoursproject was started in September 2016. In the beginning of the honoursproject, I had to get a good grasp of what I could contribute to the Eff language. I got familiar with algebraic effects and handlers aswell as the Eff programming language (and its compiler). My main resources were several publications on algebraic effect handlers, Eff and Eff's effects system [5] [6] [7] [8] [9] [10]. I also had to learn how to work in OCaml. This took me a couple weeks. The main issue that I encountered was completely understanding which optimization were already implemented and what they exactly did. Another issue I encountered was being able to work with the type-&-effect system of Eff.

1.4 Optimizations

Eventually, by the end of October, I had resolved these issues and was able to contribute to the project. During the following weeks I implemented several optimizations. The first optimization I implemented was an optimization that would remove a handler if the computation that the handler is handling was pure. This means that `with h handle (c)` would be optimized to `c`. A pure computation is a computation that does not and can not contain effects.

Another optimization that I did was the situation where `with h handle (c1 >> c2)` occurs and `c1` is pure for handler `h`. This can be optimized to `c1 >> (with h handle (c2))`.

It was also possible to consider to opposite situation, the scenario where `c2` is pure for handler `h`. This optimization becomes a bit more tricky. The code `with h handle (c1 >> c2)` can be rewritten as `with h' handle (c1)` with `h'` a rewritten handler. The `h'` handler contains the same effect clauses, but has a different value clause compared to the `h` handler. The value clause can be rewritten as `c2 >> h.value_clause`.

Also the situation where neither `c1` nor `c2` are pure for handler `h`. It still is interesting to split up the handler. Doing that might cause the compiler to see different optimization that could be done. This optimization is similar to the previous one, except that the value clause is rewritten differently. The value clause is rewritten as `with h handle c1`.

The final optimization that I made was the optimization of `with h handle (let rec defs = ... in c)`. This can be rewritten as `let rec defs = ... in (with h handle c)`.

The code for these optimizations can be found in Appendix B. The formal rules for the rewriting rules are listed in Figure 1. These rules form the minimal set of rewriting rules. During the time that the optimizations were made, I also worked on general bug fixing. The main work for the optimizations was done by the end of December. At this time, I started working on the benchmarks.

Simplification

$$\textsc{App-Fun}\quad \frac{}{(\texttt{fun}\ x \mapsto c)\ v \rightsquigarrow c[v/x]} \qquad \textsc{Do-Ret}\quad \frac{}{\texttt{do}\ x \leftarrow \texttt{return}\ v\ ;\ c \rightsquigarrow c[v/x]}$$

$$\textsc{Do-Op}\quad \frac{}{\texttt{do}\ x \leftarrow (\texttt{do}\ y \leftarrow \texttt{Op}\ v\ ;\ c_1)\ ;\ c_2 \quad \rightsquigarrow \quad \texttt{do}\ y \leftarrow \texttt{Op}\ v\ ;\ (\texttt{do}\ x \leftarrow c_1\ ;\ c_2)}$$

Handler Reduction

$$\textsc{With-LetRec}\quad \frac{}{\texttt{handle}\ (\texttt{let rec}\ f\ x = c_1\ \texttt{in}\ c_2)\ \texttt{with}\ v \rightsquigarrow \texttt{let rec}\ f\ x = c_1\ \texttt{in}\ (\texttt{handle}\ c_2\ \texttt{with}\ v)}$$

$$\textsc{With-Ret}\quad \frac{h = \{\texttt{return}\ x \mapsto c_r, [\texttt{Op}\ x\ k \mapsto c_{\texttt{Op}}]_{\texttt{Op}\in\mathcal{O}}\}}{\texttt{handle}\ (\texttt{return}\ v)\ \texttt{with}\ h \rightsquigarrow c_r[v/x]}$$

$$\textsc{With-Handled-Op}\quad \frac{h = \{\texttt{return}\ x \mapsto c_r, [\texttt{Op}\ x\ k \mapsto c_{\texttt{Op}}]_{\texttt{Op}\in\mathcal{O}}\}}{\texttt{handle}\ (\texttt{Op}\ v)\ \texttt{with}\ h \rightsquigarrow c_{\texttt{Op}}[v/x, (\texttt{fun}\ x \mapsto c_r)/k]}$$

$$\textsc{With-Pure}\quad \frac{h = \{\texttt{return}\ x \mapsto c_r, [\texttt{Op}\ x\ k \mapsto c_{\texttt{Op}}]_{\texttt{Op}\in\mathcal{O}}\} \qquad \Gamma \vdash c : A\ !\ \Delta \qquad \Delta \cap \mathcal{O} = \emptyset}{\texttt{handle}\ c\ \texttt{with}\ h \rightsquigarrow \texttt{do}\ x \leftarrow c\ ;\ c_r}$$

$$\textsc{With-Do}\quad \frac{\begin{array}{c} h = \{\texttt{return}\ x \mapsto c_r, [\texttt{Op}\ x\ k \mapsto c_{\texttt{Op}}]_{\texttt{Op}\in\mathcal{O}}\} \\ h' = \{\texttt{return}\ y \mapsto (\texttt{handle}\ c_2\ \texttt{with}\ h), [\texttt{Op}\ x\ k \mapsto c_{\texttt{Op}}]_{\texttt{Op}\in\mathcal{O}}\} \end{array}}{\texttt{handle}\ (\texttt{do}\ y \leftarrow c_1\ ;\ c_2)\ \texttt{with}\ h \rightsquigarrow \texttt{handle}\ c_1\ \texttt{with}\ h'}$$

Figure 1: Term Rewriting Rules [2]

1.5 Evaluation: Eff versus OCaml

For the evaluation of the optimizations, multiple benchmarks were used. Some simple loop benchmarks were used and a N-queens benchmark. There was a need for some more complex benchmarks to test the optimizations in bigger example programs. For this, an interpreter and a parser were chosen. Four different loop benchmarks were implemented. A pure loop program, to test the conversion from pure Eff code to OCaml. A loop program which has an effect `Fail` that is called when the loop is started with a negative amount of iterations. The final two loop benchmarks are similar to eachother, one increments a value, the other keeps state using a `Get` and `Put` effect.

The interpreter is a program that takes an AST as input and returns a result. The interpreter is based on an algorithm given by an existing published paper, Monad Transformers and Modular Interpreters, to make sure that a standard implementation was used [11]. The published paper showed that build a fully modular interpreter based on monad transformers. Not just an Eff version had to be made, but also an version in OCaml, to be able to compare the performance.

The parser is also based on an example from another published paper, Effect Handlers in Scope, which contained several examples concerning grammers and parsers, implemented in Haskell [12]. Similar to the interpreter, not just an Eff version had to be made. Also an OCaml version was made to be able to compare the performance.

The N-queens benchmarks is implemented in two different versions. The first version, `queens_all`, calculates all possible solutions to a given N-queens problem. The other version, `queens_one`, only

computes the first solution found to a given N-queens problem. The `queens_one` version has two different implementations. The first implementation uses an `Option` datatype, the other implementation uses a *continuation passing style (cps)*. The native version also contains a third implementation for `queens_one`, this implementation uses *Exceptions*.

1.6 Evaluation: Eff versus Other Systems

In order to provide a second evaluation, two different versions of the well-known N-qeens problem were tested with three different OCaml-based systems, native OCaml and Eff.

One of the OCaml-based systems was Multicore OCaml, which is a system that natively adds effect handlers to the OCaml language [13]. Mutlicore OCaml uses a modified version of the standard OCaml compiler. The implementation of the effect handlers is very different to the way Eff did it. Continuations are optimized in such a way that they can only be called once. If multiple calls are required, the continuation needs to be explicitly copied.

```
effect Decide : unit -> bool
effect Fail : unit -> empty

let queens_one_option number_of_queens =
  match (queens number_of_queens)
  with
  | effect (Decide _) k -> (match continue (Obj.clone_continuation k)
     ↪ true with Some x -> Some x | None -> continue (Obj.
     ↪ clone_continuation k) false)
  | effect (Fail _) k -> None
  | x -> (Some x)
```

The two other OCaml-based systems are Handlers in Action and Eff in OCaml [14] [15]. Both are OCaml-based systems that use a library called Delimcc to implement algebraic effect handlers [16]. We found that Delimcc does not work when compiling it to native code. However, it does work when compiled to bytecode. A mail was send to the researcher who developped Delimcc. His response was that the native compilation is indeed broken. The bytecode compilation should also be almost just as fast as the native compilation. As a result, the bytecode version was chosen. A (part of) the queens implementation of the Eff Directly in OCaml system is shown below:

```
type choice =
  | Fail of unit * (empty -> choice result)
  | Decide of unit  * (bool -> choice result)

let c = Delimcc.new_prompt ()

let fail () = match Delimcc.shift0 c (fun k -> Eff (Fail ((),k))) with _
   ↪ -> failwith "unreachable"
let decide p arg = Delimcc.shift0 p (fun k -> Eff (Decide (arg,k)))

let rec optionalize res = function
  | Done -> Some (get_result res)
  | Eff Fail ((),_) -> None
  | Eff Decide ((),k) -> (match optionalize res @@ k true with Some x ->
     ↪ Some x | None -> optionalize res @@ k false)
```

A (part of) the queens implementation of the Handlers in Action system is shown below:

```
let decide : (unit, bool) op = new_op ()
let fail : (unit, 'a) op = new_op ()

let optionalize m =
  handle m
```

```
([decide |-> (fun () k -> (match k true with Some x -> Some x | None ->
    ↪  k false));
fail |-> (fun () k -> None)],
 fun x -> (Some x))
```

Links is a functional programming language that also implemented algebraic effect handlers [17]. There is a published paper which describes a compilation from Links to OCaml [18]. During the duration of the honoursproject, I wasn't able to find a way to compile Links to OCaml. As a result, we tried to compare the resulting binary created by Links and created by OCaml (from the generated Eff code). However, for some still unknown reason, the execution time for 10-queens lasts extremely long. While other systems would compute 10-queens in less than a second, Links would take over 20 minutes. For this reason, Links was left out from the benchmarking even though the benchmarks were created. One interesting aspect is the implementation of algebraic effect handlers. Links uses a row based type-&-effect system [19]. This is subsequently the topic of my next honoursproject.

```
sig decide : () {Decide:Bool |_}~> Bool
fun decide() { do Decide }

fun optionalize(m)() {
  open handle(m) {
     case Return(x) -> Just(x)
     case Fail(k) -> Nothing
     case Decide(k) ->
        switch (k(true)) {
            case Just(x) -> Just(x)
            case Nothing -> k(false)
        }
  }
}
```

After the benchmarks were made, a method had to be devised to compare the different systems. As explained, Delimcc only works in bytecode. Multicore OCaml uses a modified compiler and Links uses a seperate compiler. It was chosen to compare the generated bytecode binaries from each system. Additional tests were done to make sure that the startup time from a binary would not change the results.

1.7 Conclusion

In the paper, not just the optimization module is described. Also a purity module is described. This module can perform purity checks on code, which has the effect that in the case of pure code, more efficient OCaml code can be generated. In Figure 2 the results for the loop benchmarks can be seen with optimizations and purity. In Figure 3, a comparison can be seen for the different systems with N-queens for all solutions. In Figure 4, the benchmark for the first solution is shown. It can be seen that Eff with purity and optimizations is as fast as native OCaml.

To review, I did a literature study of Eff and algebraic effect handlers, implemented several optimizations, implemented loop, interpreter, parser, part of the queens and the different OCaml-based systems benchmarks. Finally I was also able to contribute to the writing of the paper itself. I wrote parts of several sections of the rewrite rules.

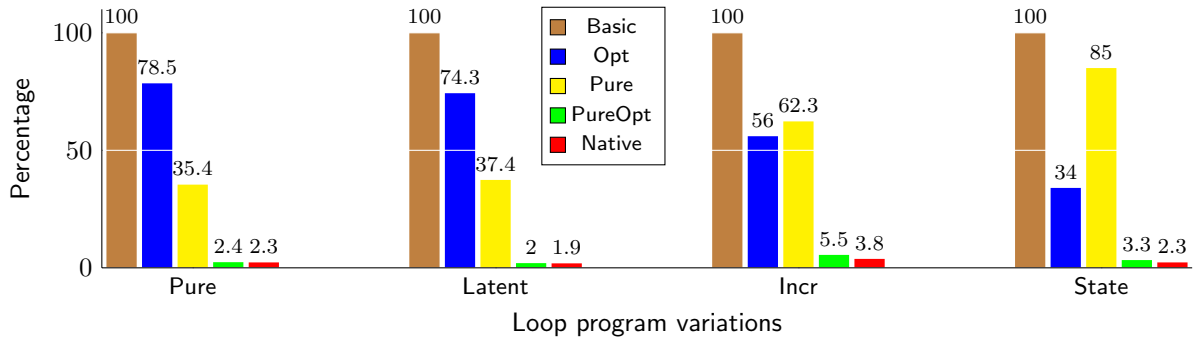

Figure 2: Relative run-times of Loops example [2]

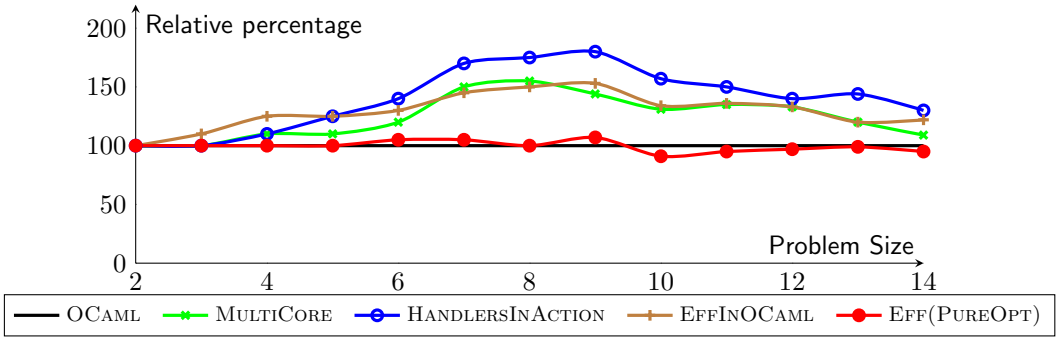

Figure 3: Results of running N-Queens for all solutions on multiple systems [2]

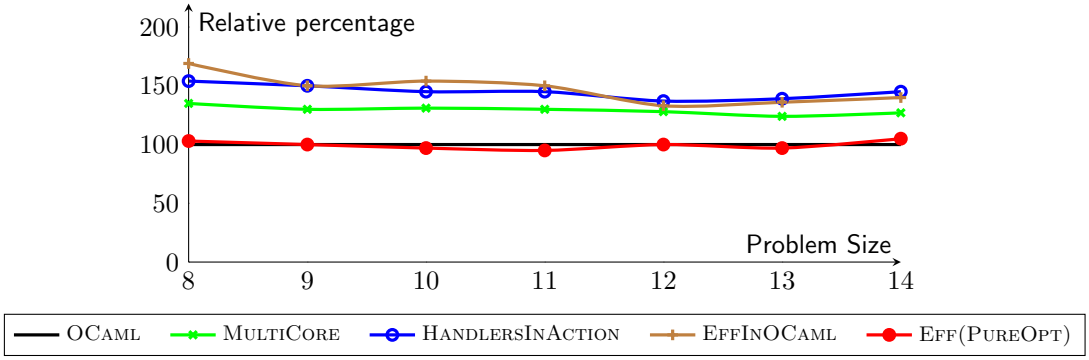

Figure 4: Results of running N-Queens for one solution on multiple systems [2]

2 Reflection

2.1 General reflection

An important, if not the most important, aspect that I learned, is that there is no shame in asking for help. This is a competency I didn't think I would learn, it is also not a competency I thought was a problem. More specifically, during the honoursproject I realised that it was better to ask a lot of questions and timely tell someone when I'm having issues then to wait.

In the proposal, I said that the honoursprogramme provided an excellent opportunity to get involved

with research. Looking back, this has become the most important reason for me. Without the honoursprogramme, I would have a different view of what research is like. I also said that I liked the challenge. In the end, the honoursproject was even more challenging than I expected it to be. Instead of doing the honoursprogramme over a period of the entire academic year, I completed it in nearly one semester. Because I really enjoyed the challenge that the honoursprogramme provided me, I decided to take the second honoursproject this year aswell. This means that I will complete the entire honoursprogramme in a single year.

2.2 Relation between honoursprogramme and degree

The honoursproject is linked to my master degree. The honoursproject helped me decide what I want to do for my masterthesis. My second honoursproject goes further in the effect handlers project. Both honoursprojects build up towards my masterthesis. This is an interesting result from doing the honoursprojects within the computer science departement. If I didn't do my honoursproject within the computer science departement, I wouldn't be able to use the honourprojects as a build up for the masterthesis. This is also something that wasn't planned in advance.

During my bachelors, I had a course on functional and logical programming. This course was useful since it provided me with knowledge about the functional programming paradigm. Though it wasn't a course, I have done several research internships. The content of these internships were not related to programming languages. However, these internships did prepare me for the honoursproject. It taught me how to work independently.

There is also a course on Formal Systems and their Applications (H04H8BE) which revolves around the typical structure and composition of a formal system such as the lambda calculus. It is taught during the second semester. I was planning to take this course at first, since the content seemed relevant to my honoursproject. However, since my honoursproject occured throughout the first semester, I wasn't able to take the course and possibly take advantage of the content.

Other than these courses, there weren't any courses which directly link to the honoursproject. However there was a synergy between the honoursproject and the courses I followed. This synergy came from the fact that my specialisation, Artificial Intelligence, is linked to the research group where I'm doing my honoursproject.

2.3 Workload of the honoursprogramme

In the proposal of the honoursproject, it was also discussed that doing the entire honoursprogramme during the masters would be difficult. To make my second year more bearable, I chose to spend 70 ECTS credits on courses during my first year instead of the usual 60 ECTS credits. This has to consequence that my first year would be even more difficult compared to the honoursproject added to the usual 60 ECTS credits.

My academic year hasn't ended yet, but I can say that it wasn't a mistake to take the 70 ECTS credits. It was a huge workload, but it also had another unintended effect. My time management skills improved significantly. The fact that I'm also taking my second honoursproject this year tells that I'm motivated and able to handle the workload.

In general, I found the workload of the honoursprogramme to be comparable to the workload of 9 ECTS credits.

2.4 Social relevance

In the initial proposal, it was said that the honoursproject is socially relevant for multiple reasons. Society continuously demands more, faster and more reliable software. If better tools for programming languages are developed, this demand can be satisfied faster. A type-&-effect system makes it easier to implement backtracking, non-determinism and more. It also makes it easier to detect bugs within a program. Optimizing a type-&-effect system makes it more viable to use such a system in commercial software.

3 Conclusions

3.1 Looking back

I was able to accomplish more than I expected in the honoursproject. I expected that I could only implement one optimization and that by then I would be done. Multiple optimizations were implemented. In this regard, I underestimated the amount of work I could do in the honoursproject.

However, It took longer to orientate myself in the landscape of algebraic effect handlers than I expected. I found that there is a big difference between getting familiar enough with algebraic effects and handlers so I could use them within programs and getting familiar enough with algebraic effects and handlers so I could do research within that field. More specifically, I could write small programs that use algebraic effects and handlers quite fast, but it took a lot longer to be able to do the optimizations.

3.2 Where the goals reached

In the motivation letter, multiple steps were given that needed to be accomplished.

1. Literature study of Eff and existing optimizations
2. Designing new optimizations
3. implementation
4. evaluation through benchmarks
5. formal proof of the optimizations

These steps, with the exclusion of step 5, were accomplished. In the group, it was decided that it was best for me to continue working on the benchmarks and writing the paper instead of writing formal proofs.

Step 1 was the first task that was accomplished. This was essential for me to be able to contribute to the research project. Afterwards step 2 and step 3 were done in parallel. Finally step 4 and working on the paper were done.

3.3 Value of the Honoursprogramme

Other goals that are important for the honoursprogramme concern my own contributions and the knowledge and experience that were gained. During the honoursproject, I was partly treated as an honoursstudent and partly as a researcher. I was partly treated as an honourstudent since I did have regular meetings with professor Schrijvers. But I was also treated as a researcher as I did have a voice during meetings for the project. This meant that I was able to make contributions and state my opinion about decisions before they are made. Ofcourse this came with the responsibility that I carried my own weight.

I would say there were a lot of values to the honoursproject. Some I have already mentioned. I

got introduced to being a researcher. The honoursproject was quite a challenge, which I definitely found a value of the honoursproject. I believe that because of these values, my thirst for more knowledge also increased. I want to delve deeper into type-&-effect systems, learn more about the theoretical foundations.

3.4 Conclusion

To conclude, something that definitely needs to be said is that I'm proud of what I've accomplished during this honoursproject. Choosing to participate in the honoursprogramme and doing my honoursproject with professor Schrijvers were the best choices I could have made. The honoursproject taught me a lot about research, programming language theory and myself.

A Competencies

Theoretical and mathematical foundations of computer science: I wanted to expand my knowledge of the theoretical and mathematical foundations of computer science. I found that I have improved this competency. More specifically, I learned about algebraic effect handlers and formal type systems. However, my thirst for more knowledge has only increased since starting the honoursproject. I want to learn more about type systems and algebraic effect handlers, especially about the theoretical foundations.

Oral communication: I learned a lot about this competency. I was able to discuss decisions to be made in the project with other researchers. In the weekly meetings, each researcher had to talk about the progress made and the issues encountered during the previous week. I feel like I was more shy towards the beginning of the honoursproject. This has gotten a lot better during the honoursproject. However, I feel like I cannot always express my thoughts in the correct scientific terms. That is definitly a working point for the future. Most communication happened in English. It was really interesting and fun to communicate professionally in English. I didn't think that improving my English communication skills was a competency I was going to learn. However, my communication skills did improve during the honoursproject. A skill that I wanted to improve was given presentations for a larger audience. However, this is something that does not really occur during the honoursproject.

Written communication: I improved this competency less than I expected. It also improved in a different way than I expected. I expected to improve my competency in writing reports. However, I didn't write many reports for this honoursproject. I did write portions of the paper, which improved my academic writing competency. This is a competency that still needs to be improved a lot. More specifically, my academic writing skills need to be improved.

Asking for help: As explained before, this is one of the most important competencies that I learned. I didn't think I would learn this competency, since I never realised that this was something that needed improvement. In a research project, it is important to understand the material, but there is also a deadline. That means that without asking for help, things are bound to go awry. This is something I learned during the honoursproject. However, I do think that this is something that can and needs improvement. Sometimes it occurs that people explain a new concept to me, but I don't fully understand it yet. In those circumstances, I would like to ask for more explanations, however I'm not always sure how to ask for help. I also think that sometimes I assume too quickly that I understand something when it would have been better to ask more questions. These are definitely issues that have a high priority for improvement.

Time management: In my reflection I explained that I improved my time management skills. I find this to be an important competency to be learned. Due to the huge workload that I took, which wasn't solely due to the honoursproject, I learned how to efficiently manage my time. I do think that there are still many improvements possible. More specifically, I'm not good at estimating how long a task will take me to execute. I often underestimate or overestimate the amount of time required for certain tasks. This happens mostly with tasks that take less than a week to complete.

Interdisciplinary interest: This competency is not directly linked to my honoursproject, but is more a consequence of the honoursprogramme. Nevertheless, I still find this to be important enough to include in this report. My honoursproject is focused on research and also happens to be within my own discipline. Due to this, the interdisciplinary character of my honoursproject is small. Due to being part of the honoursprogramme, I also attended intervision moments and honourscommunity meetings. In these meetings I met a lot of new people. Due to these events I learned how different people look towards honoursprogrammes and interdisciplinary work. This is something that broadened my view and is definitely an aspect which I find important.

Finding my limit: As explained above, I improved my time management skills. Another related competency that I improved is how to find my own limits, what I can and cannot do. This is another competency I learned due to the high workload I had. Using the honoursproject I found the bounderies of the amount of work I can do. However, I do think I'm sometimes still overzealous.

Creating a hypothesis: This is a competency I wanted to learn. I feel like this is something that improved over the course of the honoursproject. However I still find it difficult to create a hypothesis. An illustration of this is my masterthesis topic. I knew several aspects about my topic. I wanted it to be research related and it should be related to my honoursproject, type systems. However, it was quite difficult to get concrete ideas myself. Partly I believe this comes from not being knowledgeable enough about the field, which is normal since I'm a masterstudent. But partly I also believe that I also need to learn how to look for potential research ideas.

B Term rewriting: code

Below is the code for the rule rewritting as implemented in the Eff compiler.

```
| Handle ({term = Handler h}, c1)
      when (is_pure_for_handler c1 h.effect_clauses) ->
  useFuel st;
  (* Print.debug "Remove handler, since no effects in common with
     ↪ computation"; *)
  reduce_comp st (bind c1 h.value_clause)

| Handle ({term = Handler h} as handler, {term = Bind (c1, {term = (p1,
    ↪  c2)})})
      when (is_pure_for_handler c1 h.effect_clauses) ->
  useFuel st;
  (* Print.debug "Remove handler of outer Bind, since no effects in
     ↪ common with computation"; *)
  reduce_comp st (bind (reduce_comp st c1) (abstraction p1 (reduce_comp
     ↪  st (handle (refresh_expr handler) c2))))

| Handle ({term = Handler h}, {term = Bind (c1, {term = (p1, c2)})})
      when (is_pure_for_handler c2 h.effect_clauses) ->
  useFuel st;
  (* Print.debug "Move inner bind into the value case"; *)
  let new_value_clause = optimize_abs st (abstraction p1 (bind (
     ↪ reduce_comp st c2) (refresh_abs h.value_clause))) in
  let hdlr = handler {
    effect_clauses = h.effect_clauses;
    value_clause = refresh_abs new_value_clause;
  } in
  reduce_comp st (handle (refresh_expr hdlr) c1)

| Handle ({term = Handler h} as h2, {term = Bind (c1, {term = (p, c2)})
    ↪ }) ->
```

```
    useFuel st;
    (* Print.debug "Move (dirty) inner bind into the value case"; *)
    let new_value_clause = optimize_abs st (abstraction p (handle (
        ↪ refresh_expr h2) (refresh_comp (reduce_comp st c2) ))) in
    let hdlr = handler {
      effect_clauses = h.effect_clauses;
      value_clause = refresh_abs new_value_clause;
    } in
    reduce_comp st (handle (refresh_expr hdlr) (refresh_comp c1))

| Handle (h, {term = LetRec (defs, co)}) ->
let handle_h_c = reduce_comp st (handle h co) in
let res =
  let_rec' defs handle_h_c
in
reduce_comp st res
```

References

[1] Tom Schrijvers. *Algebraic Effect Handlers: Harnessing the fundamental power of effects (3E160354)*. 2016. URL: https://www.kuleuven.be/onderzoek/portaal/#/projecten/3E160354.

[2] Matija Pretnar, Amr Hany Saleh, Axel Faes, and Tom Schrijvers. "Efficient Compilation of Algebraic Effects and Handlers". In: *(in review)* (2017).

[3] Andrej Bauer and Matija Pretnar. *Eff*. 2016. URL: http://www.eff-lang.org/.

[4] KC Sivaramakrishnan. *Eff directly in OCaml*. 2016. URL: https://github.com/kayceesrk/eff_delimcc_ocaml/blob/master/queens_results.pdf.

[5] Matija Pretnar. "An Introduction to Algebraic Effects and Handlers. Invited tutorial paper". In: *Electr. Notes Theor. Comput. Sci.* 319 (2015), pp. 19–35. DOI: 10.1016/j.entcs.2015.12.003. URL: http://dx.doi.org/10.1016/j.entcs.2015.12.003.

[6] Andrej Bauer and Matija Pretnar. "An Effect System for Algebraic Effects and Handlers". In: *Logical Methods in Computer Science* 10.4 (2014). DOI: 10.2168/LMCS-10(4:9)2014. URL: http://dx.doi.org/10.2168/LMCS-10(4:9)2014.

[7] Matija Pretnar. "Inferring Algebraic Effects". In: *Logical Methods in Computer Science* 10.3 (2014). DOI: 10.2168/LMCS-10(3:21)2014. URL: http://dx.doi.org/10.2168/LMCS-10(3:21)2014.

[8] Gordon D. Plotkin and Matija Pretnar. "Handling Algebraic Effects". In: *Logical Methods in Computer Science* 9.4 (2013). DOI: 10.2168/LMCS-9(4:23)2013. URL: http://dx.doi.org/10.2168/LMCS-9(4:23)2013.

[9] Andrej Bauer and Matija Pretnar. "Programming with algebraic effects and handlers". In: *J. Log. Algebr. Meth. Program.* 84.1 (2015), pp. 108–123. DOI: 10.1016/j.jlamp.2014.02.001. URL: http://dx.doi.org/10.1016/j.jlamp.2014.02.001.

[10] Niki Vazou and Daan Leijen. "From Monads to Effects and Back". In: *Practical Aspects of Declarative Languages: 18th International Symposium, PADL 2016, St. Petersburg, FL, USA, January 18-19, 2016. Proceedings*. Ed. by Marco Gavanelli and John Reppy. Cham: Springer International Publishing, 2016, pp. 169–186. ISBN: 978-3-319-28228-2. DOI: 10.1007/978-3-319-28228-2_11. URL: http://dx.doi.org/10.1007/978-3-319-28228-2_11.

[11] Sheng Liang, Paul Hudak, and Mark Jones. "Monad Transformers and Modular Interpreters". In: *Proceedings of 22nd ACM Symposium on Principles of Programming Languages*. New York: ACM Press, 1995, pp. 333–343.

[12] Nicolas Wu, Tom Schrijvers, and Ralf Hinze. "Effect Handlers in Scope". In: *SIGPLAN Not.* 49.12 (Sept. 2014), pp. 1–12. ISSN: 0362-1340. DOI: 10.1145/2775050.2633358. URL: http://doi.acm.org/10.1145/2775050.2633358.

[13] OCamlLabs. *MultiCore OCaml*. 2016. URL: https://github.com/ocamllabs/ocaml-multicore/wiki.

[14] Ohad Kammar, Sam Lindley, and Nicolas Oury. "Handlers in Action". In: *SIGPLAN Not.* 48.9 (Sept. 2013), pp. 145–158. ISSN: 0362-1340. DOI: 10.1145/2544174.2500590. URL: http://doi.acm.org/10.1145/2544174.2500590.

[15] Oleg Kiselyov and KC Sivaramakrishnan. "Eff Directly in OCaml". In:

[16] Oleg Kiselyov. *Implementations of delimited control in OCaml, Haskell, Scheme*. 2013. URL: http://okmij.org/ftp/continuations/implementations.html.

[17] Links. *Links: Linking Theory to Practice for the Web*. 2016. URL: http://links-lang.org/.

[18] Daniel Hillerström, Sam Lindley, and K Sivaramakrishnan. "Compiling Links effect handlers to the OCaml backend". In:

[19] Daniel Hillerström and Sam Lindley. "Liberating Effects with Rows and Handlers". In: *Proceedings of the 1st International Workshop on Type-Driven Development*. TyDe 2016. Nara, Japan: ACM, 2016, pp. 15–27. ISBN: 978-1-4503-4435-7. DOI: 10.1145/2976022.2976033. URL: http://doi.acm.org/10.1145/2976022.2976033.