KU LEUVEN

FACULTEIT INGENIEURSWETENSCHAPPEN

Algebraic Subtyping for Algebraic Effects and Handlers

Axel Faes

Thesis submitted for the degree of Master of Science in Engineering: Computer Science, option Artificial Intelligence

Thesis supervisor:
Prof. dr. ir. Tom Schrijvers

Assessor:
Amr Hany Shehata Saleh, Prof. dr. ir. Bart Jacobs

Mentor:
Amr Hany Shehata Saleh

Academic year 2017 – 2018

Preface

I would like to thank everybody who kept me busy and supported me the last year, especially my promoter and my assistants. I would also like to thank the jury for reading the text.

Axel Faes

Contents

Abstract

Algebraic effects and handlers benefit from a custom type-&-effect system, a type system that also tracks which effects can happen in a program. Several such type-&-effect systems have been proposed in the literature, but all are unsatisfactory. Recently, Stephen Dolan (University of Cambridge, UK) presented a novel type system that combines subtyping and parametric polymorphism in a particulary attractive and elegant fashion. A cornerstone of his design are the algebraic properties that the subtyping relation should respect. In this work, a type-&-effect system is derived that extends Dolan's elegant type system with effect information. This type-&-effect system inherits Dolan's harmonious combination of subtyping (in our case induced by a lattice structure on the effect information) with parametric polymorphism and preserves all of its desirable properties (both low-level algebraic properties and high-level meta-theoretical properties like type soundness and the existence of principal types).

Nederlandstalige Samenvatting

Algebraische effecten en handlers ...

List of Figures

Todo list

Chapter 1

Introduction

Algebraic effects and handlers are a new formalism for formally modelling side-effects (e.g. mutable state or non-determinism) in programming languages, developed by Matija Pretnar (University of Ljubjana) and Gordon Plotkin (University of Edinburgh) [18, 19]. They can be seen as exception handlers on steroids. They look like exceptions and exception handlers. However, exception handlers have no way to access the continuation from where the exception was raised. Effect handlers do have this access.

Algebraic effects and handlers are gaining a lot of traction, not only as a formalism but also as a practical feature in actual programming languages (e.g. the Koka language developed by Microsoft Research [13]). We are collaborating with Matija Pretnar on the efficient implementation of one such language, called Eff. I have contributed to this collaboration during a project I did for the Honoursprogramme of the Faculty of Engineering Science.

Any typed programming language requires a type system in order to function. A type system governs a set of rules that assigns types to the various constructions in that typed programming language. Last year, in his December 2016 PhD thesis, Stephen Dolan (University of Cambridge, UK), has presented a novel type system that combines subtyping and parametric polymorphism in a particulary attractive and elegant fashion. A cornerstone of his design are the algebraic properties that the subtyping relation should respect. This system is called algebraic subtyping[5]

This thesis extends the algebraic subtyping approach in order to account for algebraic effects and handlers. The formal properties of this system are studied in order to find which properties are satisfied compared to other type-&-effect systems.

1.1 Motivation

Algebraic effects and handlers benefit from a custom type-&-effect system, a type system that also tracks which effects can happen in a program. Several such type-&-effect systems have been proposed in the literature [13, 19, 1]. We attribute this to the lack of

the elegant properties of Dolan's type system. Indeed the existing type-&-effect systems are not only theoretically unsatisfactory, but they are also awkward to implement and use in practice. This can be seen in recent research we did on an optimizing compiler for Eff [24]. Within this research, the main hurdle here involved working with, instead of against, the type system of Eff based on subtyping.

Research questions

- How can Dolan's elegant type system be extended with effect information?
- Which properties are preserved and which aren't preserved?
- What advantages are there to an type-&-effect system based on Dolan's elegant type system?

1.2 Contributions

The goal of this thesis, as well as it's main contribution, is to derive a type-&-effect system that extends Dolan's elegant type system with effect information. This type-&-effect system should inherit Dolan's harmonious combination of subtyping (in our case induced by a lattice structure on the effect information) with parametric polymorphism and preserve all of its desirable properties (both low-level algebraic properties and high-level meta-theoretical properties like type soundness and the existence of principal types). The following approach is taken:

1. Study of the relevant literature and theoretical background.
2. Design of a type-&-effect system derived from Dolan's, that integrates effects.
3. Proving the desirable properties of the proposed type-&-effect system: type soundness, principal typing, ...
4. Design of a type inference algorithm that derives the principal types of programs without type annotations and proving its correctness.
5. Implementation of the algorithm and comparing it to other algorithms.

1.3 Results

A novel type-&-effect system is provided. This system extends Dolan's algebraic subtyping system with effect information. The full specification for this system is given including terms, types, typing rules and equivalence rules to subtyping. A type inference algorithm for algebraic subtyping has also been extended to account for the effect information. This involves a type inference algorithm for the effects. Finally, another delivarable is an implementation of this system within the Eff programming language. This is a

fully featured programming language that is and can be used for further evaluation and comparisons.

1.4 Structure of the thesis

Chapter 2 provides the required background in programming language theory, algebraic effects and optimizing Eff. How a type system specification needs to be read can be found in the section about programming language theory. This section explains the simply typed lambda calculus which is the simplest typed calculus. This calculus is the foundation for the calculus given in the section about algebraic effects.

The section about algebraic effects gives the calculus of Eff and an thorough explanation of algebraic effects and handlers. The final section explains the research that led to this thesis, the optimization of Eff. An explanation of the optimization techniques are given and the hurdles encountered during the research are explained.

Chapter 3, 4 and 5 define the EffCore type system. This is the main constribution of this thesis. The major novelty is the representation and construction of the effect information. **Chapter 3** gives the concrete syntax and the typing rules of the EffCore type system.

Chapter 4 introduces polar types and presents the algorithm to infer principal types and the biunification algorithm. Biunification is an analogue of unification for solving subtyping constraints. The difference is that biunification works over polar types.

Chapter 5 shows how types and effects may be represented compactly as automata. This representation is an extension from the representation from Algebraic Subtyping. The automata is used in order to simplify types in order to give smaller types.

Chapter 6 explains the empirical work that is done. This includes an explanation of the Eff programming language and gives implementation details of the EffCore type system. Finally, the evaluation of the EffCore system is given as the system is compared with subtyping.

Chapter 7 reviews related work such as explicit eff subtyping, and **Chapter 8** presents some future work and concludes the thesis. Proofs are given in **Appendix A**.

Chapter 2

Background

2.1 Simply Typed Lambda Calculus

The field of programming language theory is a branch of computer science that describes how to formally define complete programming languages and programming language features, such as algebraic effect handlers.

The work described in this thesis uses several aspects from programming language theory. An important subdiscipline that is extensively used is type theory. Type theory is used to formaly describe type systems. A type system is a set of rules that are used to define the shape of meaningful programs. The *simply typed lambda calculus* will be used to show and explain the necessary background that is required for further chapters. The *simply typed lambda calculus* is the simplest and most elementary form of all typed languages. [16]

2.1.1 Types and terms

Types and terms are necessary components of the *simply typed lambda calculus*. A simply example is the identity function, $\lambda x : bool.x$. In this example, we can see the syntax of our calculus. The different elements of the syntax are called the terms. Every term also needs to have a type.

Terms Figure 2.1 shows the five term constructors of the *simply typed lambda calculus* [15]. A variable by itself is already a term. `true` and `false` are also already terms. The abstraction of some variable x from a certain term t is called a function. Finally, an application is a term. The terms define the syntax of a programming language, but it does not place any constraints on how these terms can be composed. A wanted constraint could for example be that an application $t_1\, t_2$ should only be valid if t_1 is a function. This shows that only having terms is not enough to describe a programming language. [10]

Types Since we are describing the *simply typed lambda calculus*, we require the concept of "types". As seen in figure 2.1, there are two types, the base type and the function type. The function type, also called the arrow type, is used for functions. The type of $\lambda x : bool.x$ is $bool : bool$. The function takes an input of type $bool$, which in the function type can be seen on the left of the arrow. The function return that same input variable. The return type can be seen in the function type on the right of the arrow. In a valid and meaningful program, every term has a type. A term is called well typed or typable if there is a type for that term.

terms t	$::=$	x	variable
	$\mid$	`true`	true
	$\mid$	`false`	false
	$\mid$	$\lambda x : T.t$	function
	$\mid$	$t_1\, t_2$	application
type T	$::=$	$bool$	base type
	$\mid$	$T \to T$	function type

Figure 2.1: Types and terms of simply typed lambda calculus

2.1.2 Typing rules

Typing rules are used to bring structure in the programming language. The example, $\lambda x : bool.x$ has type $bool : bool$. In this example, we would expect that both occurences of x have type $bool$. However, the specification given in section 2.1.1 doesn't impose this structure. This is done with typing rules or types judgements. The typing rules for the *simply typed lambda calculus* are given in figure 2.2.

typing contexts $\Gamma ::= \epsilon \mid \Gamma, x : T$

$$\textsc{True}\ \frac{}{\Gamma \vdash \texttt{true} : bool} \qquad \textsc{False}\ \frac{}{\Gamma \vdash \texttt{false} : bool} \qquad \textsc{Var}\ \frac{(x : T) \in \Gamma}{\Gamma \vdash x : T}$$

$$\textsc{App}\ \frac{\Gamma \vdash t_1 : T_1 \to T_2 \qquad \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1\, t_2 : T_2} \qquad \textsc{Fun}\ \frac{\Gamma, x : T_1 \vdash t : T_2}{\Gamma \vdash \lambda x : T.t : T_1 \to T_2}$$

Figure 2.2: Typing of simply typed lambda calculus

The first rules to take note of are the True and False rules. These rules do not have a premise and are called facts. They state that the terms true and false have type *bool*. A

Fact states that, under the assumption of Γ, t has type T. The context, Γ, is a mapping of the free variables of t to their types. It is called a fact since the rule always holds.

The context, Γ, is a (possibly empty) collection of variables mapped to their types. The VAR rule states that, if the context contains a mapping for a variable, that variable is also a valid term with that type. The APP rule defines the usage of a function. When there are two terms t_1 and t_2 with types $T_1 \to T_2$ and T_1, then the application $t_1\, t_2$ will have the type T_2.

There are two ways to read inference rules. It can be read top-down or bottom-up. Reading it top-down gives the above described reasoning. Given some expressions and some constraints, another expression can be constructed with a specific type. The bottom-up approach states that, given an expression such as the function application, there is a specific way the different parts of the expression can be typed. In the APP rule, a function expression has type T_2. Therefor, both t_1 and t_2 must follow a specific set of constraints. It is known that a function needs to exist of type $T_1 \to T_2$ and an expression that matches the argument of the function, T_1 needs to exist. [16]

Finally, there is the FUN rule. This rule is also called a function abstraction or simply an abstraction. It shows how a function can be constructed. The interesting part of this rule is $\Gamma, x : T_1 \vdash t : T_2$. This states that t is only entailed by some context and a variable of type T_1.

2.1.3 Extensions

Starting with the *simply typed lambda calculus*, extensions can be added onto this calculus. In chapter 2.2, EFF will be discussed. EFF's calculus is a modification of the *simply typed lambda calculus* with algebraic effects and handlers. EFF also uses subtyping rules, this concept will also be further explained in the next chapter. After this, algebraic subtyping will be added to EFF's calculus. This is described in chapter 3.

There are many other aspects to a specification than just the ones discussed in this section. Certain aspects or properties could be proved in order to show that they do (or do not) hold in the given calculus. Type inference is another aspect which is not talked about in this chapter. Type inference revolves around the automatic detection (or inference) of the types of terms. Both proofs and a type inference algorithm are given in later chapters.

2.2 Algebraic Effects and Handlers

Algebraic effect handling is a very active area of research. The theoretical background of algebraic effects and handlers as developed by Plotkin, Pretnar and Power [17, 18]. Implementations of algebraic effect handlers are becoming available and the theory is actively being developed. One such implementation is the EFF programming language. This is the first language to have effects as first class citizens [22].

Figure 2.3: Algebraic effects and handlers example

```
1  effect DivisionByZero : unit -> empty;;
2
3  let divide a b =
4    if (b == 0) then
5      #DivisionByZero ()
6    else
7      a / b;;
8
9  let safeDivide a b =
10   handle (divide a b) with
11     | #DivisionByZero () k -> 0;;
```

The type-&-effect system that is used in Eff is based on subtyping and algebraic effect handlers [1]. The *simply typed lambda calculus* is used as a basis for Eff. Let us start with a simple example in order to show what algebraic effects and handlers are. With this example, the differences with the *simply typed lambda calculus* can also be shown.

In the example in figure 2.3, a new effect is defined *DivisionByZero*. In essence, this effect can be thought of as an exception. From the type that is written, it can also be seen that an exception has some relation with functions. In this case, the effect describes a function type from *unit* to *empty*. This type describes what kind of argument the effect requires in order to be called and what kind of type the continuation needs in order to procede. Calling the effect is done by the notation `#DivisionByZero ()`.

The effect can be called just like any function can be called, by applying an argument to it. Here, an important distinction can be made. Any term that can contain effects are called computations, while terms that cannot contain effects are called expressions. Finally, computations can be handled. This can be thought of as an exception handler with the big difference being that within an effect handler, there is access to a continuation to the place where the effect was called.

In the second example in figure 2.4, the effect *Op* has an *int* as the return value. In the handler, the continuation called *k* is called with *1* as the argument. The continuation looks like a program with a "hole". In the example, figure 2.5 represents the continuation with `[ ]` representing the hole. Thus, if the function *someFun* is called with *b* equal to *0*, the number *1* will be printed due to the continuation. The handler can also choose to ignore the continuation, as seen in figure 2.3, or it can call the continuation more than once. This shows the power of algebraic effects and handlers in a small, albeit artificial, example.

Figure 2.4: Algebraic effect continuation usage example

```
effect Op : unit -> int;;

let someFun b =
  handle (
    if (b == 0) then
      let a = #Op () in
      print a
    else
      print b
  ) with
    | #Op () k -> k 1;;
```

Figure 2.5: Algebraic effect continuation example

```
let a = [ ] in
print a
```

2.2.1 Types and terms

In order to extend the *simply typed lambda calculus* to Eff's calculus, several terms need to be added. A term is required in order to call effects and a term is required to handle effects. Some additional types are needed to represent handlers and the effects. [22]

The types receive a big extension. A sort is needed to represent effects. It is also important to distuingish types between expressions and computations. Having such a distinction makes the difference also explicit on type level. Types given to expressions are called pure types. A pure type has no representation of effects. Types given to computations are called dirty types. A dirty type is represented by combining a pure type with a representation for effects. The representation for effects are called dirts.

Terms Figure 2.6 shows the two sorts of terms in Eff. As explained before, there are values, or expressions v and computations c. Computations are terms that can contain effects. Effects are denoted as operations Op which can be called. [19]

In Eff, there are also several other small additions aside from the terms required for the algebraic effects and handlers. Sequencing, a conditional and a recursive definition have also been added. This was done in order to enrich the language and further exploit the advantage of algebraic effects and handlers. [2]

value v	::=	x	variable
	|	`true`	true
	|	`false`	false
	|	$\lambda x.c$	function
	|	$\{$	handler
		`return` $x \mapsto c_r$,	return case
		$[\mathtt{Op}\ y\ k \mapsto c_{\mathtt{Op}}]_{\mathtt{Op} \in O}$	operation cases
		$\}$	
comp c	::=	$v_1\ v_2$	application
	|	`do` $x \leftarrow c_1$; c_2	sequencing
	|	`if` e `then` c_1 `else` c_2	conditional
	|	`let rec` $f\ x = c_1$ `in` c_2	rec definition
	|	`return` v	returned val
	|	$\mathtt{Op}\ v$	operation call
	|	`handle` c `with` v	handling

Figure 2.6: Terms of Eff

Types Figure 2.7 shows the types of Eff. There are two main sorts of types. There are (pure) types A, B and dirty types $\underline{C}, \underline{D}$. A dirty type is a pure type A tagged with a finite set of operations Δ, which we call dirt, that can be called. This finite set Δ is in general defined as an over-approximation of the operations that are actually called. The type $\underline{C} \Rightarrow \underline{D}$ is used for handlers because a handler takes an input computation with type $\underline{C}$, handles the effects in this computation and outputs a computation with type $\underline{D}$ as the result [19]. Other than the handler type and the distinction between pure and dirty types, there is nothing new compared to the types from the *simply typed lambda calculus*.

(pure) type A, B	::=	`bool`	bool type
	|	$A \to \underline{C}$	function type
	|	$\underline{C} \Rightarrow \underline{D}$	handler type
dirty type $\underline{C}, \underline{D}$	::=	$A\ !\ \Delta$	
dirt Δ	::=	$\{\mathtt{Op}_1, \ldots, \mathtt{Op}_n\}$	

Figure 2.7: Types of Eff

2.2.2 Subtyping

Eff uses a subtyping based system. Subtyping is a form of type polymorphism. Types can be related to eachother, being either subtypes or supertypes. Intuitively one could think about Java classes and inheritance in order to understand subtyping. There are

some big differences between inheritance and subtyping, but from the principle of gaining an understanding of what subtyping entails, the relation between the two can be made.

Let us take the subtyping judgement $\texttt{bool} \leqslant \texttt{bool}$. This judgement is about reflexivity. It states that `bool` is a subtype of itself. The subtyping judgement for the arrow type (functions) states that, if we have $A' \leqslant A$ and $\underline{C} \leqslant \underline{C'}$, then we also induce the natural subtyping judgement $A \to \underline{C} \leqslant A' \to \underline{C'}$. This tells us that, if we have a function, the caller can always call that function with a type that is "more" and that function can always return "more" than what the caller expects. This can be easily visualised when the argument and return values are records. If a function requires a record with labels "x" and "y", the caller is allowed to call the function with a record containing more that just "x" and "y". A similar analogy can be made for the return values. Functions are contravariant in its argument types and covariant in its return types.

The dirty type $A \,!\, \Delta$ is assigned to a computation returning values of type A and potentially calling operations from the set Δ. This set Δ is always an over-approximation of the actually called operations, and may safely be increased, inducing a natural subtyping judgement $A \,!\, \Delta \leqslant A \,!\, \Delta'$ on dirty types where Δ' contains extra operations compared to Δ. As dirty types can occur inside pure types, we also get a derived subtyping judgement on pure types. Both judgements are defined in Figure 2.8. Observe that, as usual, subtyping is contravariant in the argument types of functions as well as handlers, and covariant in their return types. [21]

Subtyping

$$\textsc{Sub-bool}\ \frac{}{\texttt{bool} \leqslant \texttt{bool}} \qquad \textsc{Sub-}\to\ \frac{A' \leqslant A \qquad \underline{C} \leqslant \underline{C'}}{A \to \underline{C} \leqslant A' \to \underline{C'}} \qquad \textsc{Sub-}\Rightarrow\ \frac{\underline{C'} \leqslant \underline{C} \qquad \underline{D} \leqslant \underline{D'}}{\underline{C} \Rightarrow \underline{D} \leqslant \underline{C'} \Rightarrow \underline{D'}}$$

$$\textsc{Sub-!}\ \frac{A \leqslant A' \qquad \Delta \subseteq \Delta'}{A \,!\, \Delta \leqslant A' \,!\, \Delta'}$$

Figure 2.8: Subtyping for pure and dirty types of Eff

2.2.3 Typing rules

Figure 2.9 and figure 2.10 defines the typing judgements for values and computations with respect to a standard typing context Γ. This types context can contain ϵ or a variable with a type.

Values The rules for subtyping, variables, and functions are entirely standard.

A handler expression has type $A \,!\, \Delta \cup O \Rightarrow B \,!\, \Delta$ iff all branches (both the operation cases and the return case) have dirty type $B \,!\, \Delta$ and the operation cases cover the set of

operations O. Note that the intersection $\Delta \cap O$ is not necessarily empty. The handler deals with the operations O, but in the process may re-issue some of them (i.e., $\Delta \cap O$).

When typing operation cases, the given signature for the operation $(\mathtt{Op} : A_{\mathtt{Op}} \to B_{\mathtt{Op}}) \in \Sigma$ determines the type $A_{\mathtt{Op}}$ of the parameter x and the domain $B_{\mathtt{Op}}$ of the continuation k. As our handlers are deep, the codomain of k should be the same as the type $B \,!\, \Delta$ of the cases. [21]

Computations With the following exceptions, the typing judgement $\Gamma \vdash c : \underline{C}$ has a straightforward definition. The `return` construct renders a value v as a pure computation, i.e., with empty dirt. An operation invocation $\mathtt{Op}\, v$ is typed according to the operation's signature, with the operation itself as its only operation. Finally, rule With shows that a handler with type $\underline{C} \Rightarrow \underline{D}$ transforms a computation with type $\underline{C}$ into a computation with type $\underline{D}$. [21]

typing contexts $\Gamma ::= \epsilon \mid \Gamma, x : A$

Expressions

$$\textsc{SubVal}\ \frac{\Gamma \vdash v : A \qquad A \leqslant A'}{\Gamma \vdash v : A'} \qquad \textsc{Var}\ \frac{(x : A) \in \Gamma}{\Gamma \vdash x : A} \qquad \textsc{True}\ \frac{}{\Gamma \vdash \mathtt{true} : \mathit{bool}}$$

$$\textsc{False}\ \frac{}{\Gamma \vdash \mathtt{false} : \mathit{bool}} \qquad \textsc{Fun}\ \frac{\Gamma, x : A \vdash c : \underline{C}}{\Gamma \vdash \lambda x.c : A \to \underline{C}}$$

$$\textsc{Hand}\ \frac{\Gamma, x : A \vdash c_r : B \,!\, \Delta \qquad \left[(\mathtt{Op} : A_{\mathtt{Op}} \to B_{\mathtt{Op}}) \in \Sigma \qquad \Gamma, y : A_{\mathtt{Op}}, k : B_{\mathtt{Op}} \to B \,!\, \Delta \vdash c_{\mathtt{Op}} : B \,!\, \Delta\right]_{\mathtt{Op} \in O}}{\Gamma \vdash \{\mathtt{return}\ x \mapsto c_r, [\mathtt{Op}\ y\ k \mapsto c_{\mathtt{Op}}]_{\mathtt{Op} \in O}\} : \quad A \,!\, \Delta \cup O \Rightarrow B \,!\, \Delta}$$

Figure 2.9: Typing of expressions in Eff

typing contexts $\Gamma ::= \epsilon \mid \Gamma, x : A$

Computations

$$\textsc{SubComp}\ \frac{\Gamma \vdash c : \underline{C} \qquad \underline{C} \leqslant \underline{C'}}{\Gamma \vdash c : \underline{C'}} \qquad \textsc{App}\ \frac{\Gamma \vdash v_1 : A \to \underline{C} \qquad \Gamma \vdash v_2 : A}{\Gamma \vdash v_1\, v_2 : \underline{C}}$$

$$\textsc{Cond}\ \frac{\Gamma \vdash v : \mathit{bool} \qquad \Gamma \vdash c_1 : \underline{C} \qquad \Gamma \vdash c_2 : \underline{C}}{\Gamma \vdash \texttt{if}\ v\ \texttt{then}\ c_1\ \texttt{else}\ c_2 : \underline{C}}$$

$$\textsc{LetRec}\ \frac{\Gamma, f : A \to \underline{C}, x : A \vdash c_1 : \underline{C} \qquad \Gamma, f : A \to \underline{C} \vdash c_2 : \underline{D}}{\Gamma \vdash \texttt{let rec}\ f\ x = c_1\ \texttt{in}\ c_2 : \underline{D}} \qquad \textsc{Ret}\ \frac{\Gamma \vdash v : A}{\Gamma \vdash \texttt{return}\ v : A \,!\, \emptyset}$$

$$\textsc{Op}\ \frac{(\texttt{Op} : A \to B) \in \Sigma \qquad \Gamma \vdash v : A}{\Gamma \vdash \texttt{Op}\, v : B \,!\, \{\texttt{Op}\}} \qquad \textsc{Do}\ \frac{\Gamma \vdash c_1 : A \,!\, \Delta \qquad \Gamma, x : A \vdash c_2 : B \,!\, \Delta}{\Gamma \vdash \texttt{do}\ x \leftarrow c_1\ ;\ c_2 : B \,!\, \Delta}$$

$$\textsc{With}\ \frac{\Gamma \vdash v : \underline{C} \Rightarrow \underline{D} \qquad \Gamma \vdash c : \underline{C}}{\Gamma \vdash \texttt{handle}\ c\ \texttt{with}\ v : \underline{D}}$$

Figure 2.10: Typing of computations in Eff

2.2.4 Type Inference

Type inference is the process where types are automatically inferred by the compiler. Types rules are used as a blueprint for type inference. Every typing rule indicates a situation a program can be in at any point in time. Thus, for every typing rule, there has to be a type inference rule.

In the case of a subtyping based system, contraint-based type inference rules are used. The specific rules for Eff are not fully given as they are not required for the work in this thesis. The idea behind constraint-based type inference rules is that, in each rule, constraints can be made. In case of a subtyping based system, these constraints are subtyping constraints between two types.

In figure 2.11, the type inference for function specialization can be seen. We have two expressions v_1 and v_2 with types A_1 and A_2. The application produces some type $\alpha \,!\, \delta$. In order to link the types of the two expressions to the produced type, a subtyping constraint is used. The constraint $A_1 \leqslant A_2 \to (\alpha \,!\, \delta)$ indicates that A_1 has to be a subtype of a function type $A_2 \to (\alpha \,!\, \delta)$.

The reader may wonder what happens when the subtyping constraint is changed into an equality constraint $A_1 = A_2 \to (\alpha \,!\, \delta)$. If every subtyping relation is changed into an equality relation, including for all relations in the subtyping rules in figure 2.8, than we have changed the subtyping system into a Hindley-Milner system. The Hindley-Milner system is less expressive than the subtyping system. This makes sense as an equation with subtyping $\leqslant$ allows for more solutions than using equality $=$.

For every typing rule, there is a type inference rule. At the end of the of applying all the rules, Θ contains a lot of constraints. These constraints are solved as much as possible using substitution techniques. Subtyping does not allow for all constraints to be completely solved. In contrast, the Hindley-Milner system can solve all constraints with substitution techniques.

typing contexts $\Gamma \;::=\; \epsilon \mid \Gamma, x : A$

Computations

$$\textsc{App}\quad \frac{\Gamma \triangleright v_1 : A_1 \qquad \Gamma \triangleright v_2 : A_2}{\Gamma \triangleright v_1\, v_2 : \alpha \,!\, \delta} \Theta = \Theta \cup (A_1 \leqslant A_2 \to (\alpha \,!\, \delta))$$

Figure 2.11: Type inference rule for function application for Eff

2.3 Optimizations for Eff

In the previous section, we discussed the background of algebraic effects and handlers as developed by Plotkin, Pretnar and Power [17, 18]. The system has been worked out and has been implemented as the Eff programming language of which the calculus has been described in the previous section. This was the first language to have effects as first class citizens [22]. Algebraic effects and handlers aren't just a fancy new concept, it is quickly maturing. There is more and more adoption of algebraic effects as a practical language feature for user-defined side-effects. As language features are being adopted more and more, optimization becomes a bigger priority.

This section is a summerization of the work by Matija Pretnar et al. of which I was a co-author. This work focusses on the optimization of algebraic effect and handlers. First, we explain why optimizations of algebraic effects and handlers are needed. Afterwards, the actual optimization and its evaluation is given. Finally, some issues with the optimizations are given. The novelty of my thesis, while not being a solution to these issues, arises from the work done in the optimizations.

2.3.1 Motivation

Considering that multiple implementation are available, runtime performance becomes much more important. Some implementations take the form a interpreters [2, 9]. Most implementations take the form of libraries [3, 11, 12]. Most work has been towards the optimization of the runtime performance. However, in the case of Eff, there still was a performance difference of about 4400% between the algebraic effects and hand-written code in OCaml (without algebraic effects).

Another viable option was to provide an optimised compiler in order to transform the algebraic effects and handlers such that the runtime cost is avoided entirely [24]. The optimised compiler is implemented for the Eff programming language. This work is one of the main driving factors behind the start of this thesis.

The optimised compiler shows a lot of promise. By the end of this chapter, it will be clear that compile-time optimization is a very viable method of optimizing algebraic effects and handlers. However, there are some issues with this work, as wil be further discussed in Chapter 2.3.4. More specifically, there are issues with the underlying type system of EFF. There are several methods of solving these issues, one of which leads to the work proposed in this thesis.

2.3.2 Implementation

There are two main ways that the optimising compiler used in order to optimise EFF code. The first is through the use of term rewriting rules, the other way is through purity aware compilation. Purity aware compilation provides a way for pure computations to have a more efficient representation, compared to the free monad representation.

The term rewriting rules that were created during the creation of the optimised compiler is given in figure 2.12. These rules show how terms are rewritten in order to minimize the footprint of algebraic effects and handlers.

There is also function specialisation, which is a special term rewrite rule. Function specialisation is used in order to deal with seemingly non-terminating recursive functions. Any recursive function `let rec f x = cf in c` that is used (and handled) can be rewritten with the following rewrite rule: `handle f v with h` $\mapsto$ `let rec f' x = handle cf with h in f' v`. In other words, function specialisation is about bringing handlers inside the function definition.

The standard way to compile algebraic effect handlers with free monad representations introduces a substantial performance overhead. This is especially the case for pure computations. We want to be able to differentiate between pure and impure computations.

One important aspect are the subtyping judgements that are used to elaborate types into functions that coerce one type into another, as seen in Figure 2.13. The elaboration judgement $(\Gamma \vdash v : A) \rightsquigarrow E$ and $(\Gamma \vdash c : \underline{C}) \rightsquigarrow E$ means that any value v or any computation c is elaborated into an OCAML expression E. The different elaboration

Simplification

App-Fun

$$\frac{}{(\lambda x.c)\, v \rightsquigarrow c[v/x]}$$

Do-Ret

$$\frac{}{\texttt{do}\ x \leftarrow \texttt{return}\ v\ ;\ c \rightsquigarrow c[v/x]}$$

Do-Op

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

Handler Reduction

With-LetRec

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

With-Ret

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

With-Handled-Op

$$\frac{h = \{\texttt{return}\ x \mapsto c_r, [\texttt{Op}\ y\ k \mapsto c_{\texttt{Op}}]_{\texttt{Op} \in O}\}}{\texttt{handle}\ (\texttt{Op}\, v)\ \texttt{with}\ h \rightsquigarrow c_{\texttt{Op}}[v/x, (\lambda x.c_r)/k]}$$

With-Pure

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

With-Do

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

Figure 2.12: Term Rewriting Rules [24]

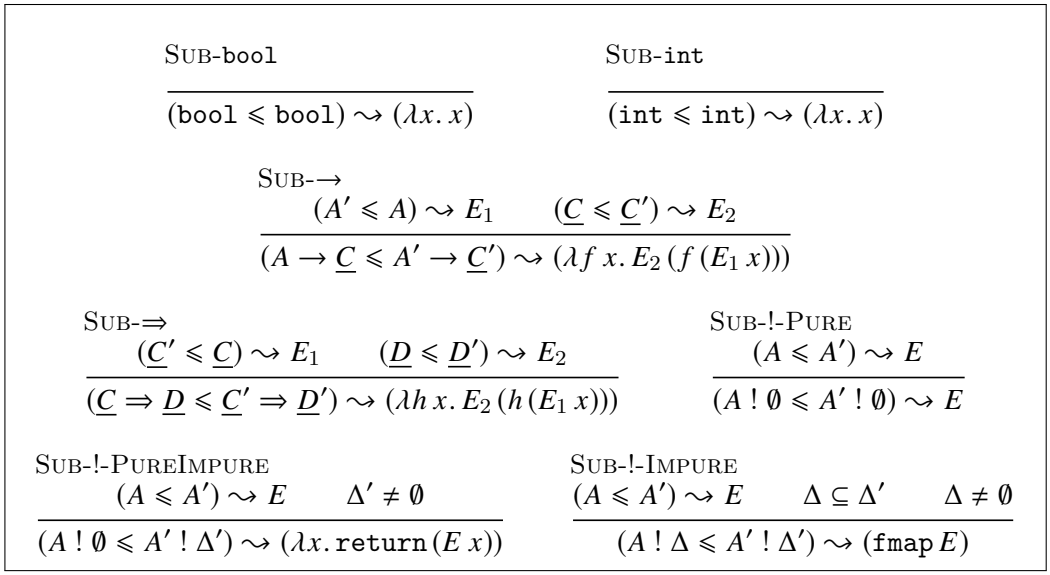

Figure 2.13: Subtyping induced coercions [24]

judgements can be seen in Figure 2.14 and Figure 2.15. Note that the rules SubVal and SubComp utilise the subtyping judgements. The rules HandPure, HandImpure, DoPure and DoImpure distinguish between pure and impure cases in order to generate the most optimal code. A pure `return` v computation is translated just like the value v.

The combination of the purity aware compilation and the term rewrite rules allows for near complete optimization. In principle, handlers are pushed as deep as possible within the program until they can either be removed due to them not being needed (e.g. handling a value, handling an effect that does not appear in the handler clause) or until they can be evaluated (handling an operation term that does appear in the handler clause). The purity aware compilation then makes sure that pure computations are efficiently translated to OCaml.

2.3.3 Evaluation

The optimising compiler needs to be evaluated empirically with several testing programs. This happens in two stages. First the different compilation schemes are compared with hand-written OCaml code. Secondly, the compiler is compared against different implementations of algebraic effects and handlers. All benchmarks were run on a MacBook Pro with an 2.5 GHz Intel Core I7 processor and 16 GB 1600 MHz DDR3 RAM running Mac OS 10.12.3. The evaluation of the system devised in this thesis, as will be discussed in Chapter 6.5, cannot be directly compared against the results from the evaluation discussed here. The evaluation of Chapter 6.5 was run on the same system running Mac OS 10.13.4 instead of 10.12.3.

Values

$$\textsc{SubVal}\ \frac{(\Gamma \vdash v : A) \rightsquigarrow E_1 \qquad (A \leqslant A') \rightsquigarrow E_2}{(\Gamma \vdash v : A') \rightsquigarrow (E_2\, E_1)} \qquad \textsc{Var}\ \frac{(x : A) \in \Gamma}{(\Gamma \vdash x : A) \rightsquigarrow x} \qquad \textsc{Const}\ \frac{(\mathtt{k} : A) \in \Sigma}{(\Gamma \vdash \mathtt{k} : A) \rightsquigarrow \mathtt{k}}$$

$$\textsc{Fun}\ \frac{(\Gamma, x : A \vdash c : \underline{C}) \rightsquigarrow E}{(\Gamma \vdash \lambda x.c : A \to \underline{C}) \rightsquigarrow (\lambda x.\, E)}$$

$$\textsc{HandPure}\ \frac{(\Gamma, x : A \vdash c_r : B\ !\ \emptyset) \rightsquigarrow E_r}{(\Gamma \vdash \{\mathtt{return}\ x \mapsto c_r\} : A\ !\ \emptyset \Rightarrow B\ !\ \emptyset) \rightsquigarrow (\lambda x.\, E_r)}$$

$$\textsc{HandImpure}\ \frac{\begin{array}{c}(\Gamma, x : A \vdash c_r : B\ !\ \Delta) \rightsquigarrow E_r \\ \left[(\mathtt{Op} : A_{\mathtt{Op}} \to B_{\mathtt{Op}}) \in \Sigma \qquad (\Gamma, x : A_{\mathtt{Op}}, k : B_{\mathtt{Op}} \to B\ !\ \Delta \vdash c_{\mathtt{Op}} : B\ !\ \Delta) \rightsquigarrow E_{\mathtt{Op}}\right]_{\mathtt{Op} \in O} \\ E_i = \begin{cases} \lambda x\, k.[\![c_{\mathtt{Op}_i}]\!] & \mathtt{Op}_i \in O \\ \lambda x\, k.\mathrm{op}_i x >\!\!>\!\!= k & \mathtt{Op}_i \in \Delta - O \\ \lambda x\, k.\mathtt{assert\ false} & \text{otherwise} \end{cases}\end{array}}{\begin{array}{c}(\Gamma \vdash \{\mathtt{return}\ x \mapsto c_r, [\mathtt{Op}\ y\ k \mapsto c_{\mathtt{Op}}]_{\mathtt{Op} \in O}\} : A\ !\ \Delta \cup O \Rightarrow B\ !\ \Delta) \\ \rightsquigarrow \mathtt{handler}\ \{\mathtt{return} = \lambda x.E_r; \mathrm{op}_1 = E_1; \ldots; \mathrm{op}_n = E_n\}\}\end{array}}$$

Figure 2.14: Type-&-effect-directed purity aware compilation for expressions [24]

In the first evaluation, Eff is compared against OCaml. Figure 2.17 shows the runtime of a simple loop example that was used for this evaluation. The loop example used is shown below in figure 2.16. Four different variations were used in the evaluation. A *Pure* version was used containing no side-effects, a *Latent* version which contains an effect that is never called during the runtime, a *Inc* version which uses a single effect to increment an implicit state and a *State* version which is what figure 2.16 shows. The *State* version uses two operations *Get* and *Put* in order to perform operations on an implicit state.

The *Native* result is the execution of handwritten OCaml code. The *Basic* version is Eff without optimizations, while *Opt*, *Pure* and *PureOpt* respectively perform optimization, purity aware compilation and both. In order to obtain the resuls, the execution run for *10,000* iterations.

Within the second evaluation, the *N-Queens* problem was used. Eff with optimization and purity aware compilation was tested against three other implementations of algebraic effects and handlers. Multicore Ocaml, which provides support for algebraic effect and handlers within OCaml, was used. Multicore Ocaml provides a very efficient way to

Computations

$$\textsc{SubComp}\quad \frac{(\Gamma \vdash c : \underline{C}) \rightsquigarrow E_1 \qquad (\underline{C} \leqslant \underline{C'}) \rightsquigarrow E_2}{(\Gamma \vdash c : \underline{C'}) \rightsquigarrow (E_2\, E_1)}$$

$$\textsc{App}\quad \frac{(\Gamma \vdash v_1 : A \to \underline{C}) \rightsquigarrow E_1 \qquad (\Gamma \vdash v_2 : A) \rightsquigarrow E_2}{(\Gamma \vdash v_1\, v_2 : \underline{C}) \rightsquigarrow (E_1\, E_2)}$$

$$\textsc{LetRec}\quad \frac{(\Gamma, f : A \to \underline{C}, x : A \vdash c_1 : \underline{C}) \rightsquigarrow E_1 \qquad (\Gamma, f : A \to \underline{C} \vdash c_2 : \underline{D}) \rightsquigarrow E_2}{(\Gamma \vdash \texttt{let rec}\ f\ x = c_1\ \texttt{in}\ c_2 : \underline{D}) \rightsquigarrow (\texttt{let rec}\ f\ x = E_1\ \texttt{in}\ E_2)}$$

$$\textsc{Ret}\quad \frac{(\Gamma \vdash v : A) \rightsquigarrow E}{(\Gamma \vdash \texttt{return}\ v : A\ !\ \emptyset) \rightsquigarrow E} \qquad \textsc{Op}\quad \frac{(\texttt{Op} : A \to B) \in \Sigma \qquad (\Gamma \vdash v : A) \rightsquigarrow E}{(\Gamma \vdash \texttt{Op}\, v : B\ !\ \{\texttt{Op}\}) \rightsquigarrow (\texttt{op}\, E)}$$

$$\textsc{DoPure}\quad \frac{(\Gamma \vdash c_1 : A\ !\ \emptyset) \rightsquigarrow E_1 \qquad (\Gamma, x : A \vdash c_2 : B\ !\ \emptyset) \rightsquigarrow E_2}{(\Gamma \vdash \texttt{do}\ x \leftarrow c_1\ ;\ c_2 : B\ !\ \emptyset) \rightsquigarrow (\texttt{let}\ x = E_1\ \texttt{in}\ E_2)}$$

$$\textsc{DoImpure}\quad \frac{(\Gamma \vdash c_1 : A\ !\ \Delta) \rightsquigarrow E_1 \qquad (\Gamma, x : A \vdash c_2 : B\ !\ \Delta) \rightsquigarrow E_2 \qquad \Delta \neq \emptyset}{(\Gamma \vdash \texttt{do}\ x \leftarrow c_1\ ;\ c_2 : B\ !\ \Delta) \rightsquigarrow (E_1 {>>=} \lambda x.E_2)}$$

$$\textsc{With}\quad \frac{(\Gamma \vdash v : \underline{C} \Rightarrow \underline{D}) \rightsquigarrow E_1 \qquad (\Gamma \vdash c : \underline{C}) \rightsquigarrow E_2}{(\Gamma \vdash \texttt{handle}\ c\ \texttt{with}\ v : \underline{D}) \rightsquigarrow (E_1\, E_2)}$$

Figure 2.15: Type-&-effect-directed purity aware compilation for computations [24]

Figure 2.16: Loop code testing program

```
1  effect Get: unit -> int
2  effect Put: int -> unit
3
4  let rec loop_state n =
5      if n = 0 then
6          ()
7      else
8          (#Put ((#Get ()) + 1); loop_state (n - 1))
9
10 let state_handler = handler
11 | val y -> (fun x -> x)
12 | #Get () k -> (fun s -> k s s)
13 | #Put s' k -> (fun _ -> k () s')
14
15 let test_state n =
16     (with state_handler handle loop_state n) 0
```

store and run algebraic effects on runtime. However, using a continuation multiple times requires an expensive copying operation of the continuation [6]. The other systems are the Handlers in Action implementation [11] and the Eff Directly in OCaml implementation [12].

The results are quite promising as the performance of fully optimised Eff becomes very similar to native code without algebraic effects. Comparing the performance against other systems also shows that our approach is consistently the fastest.

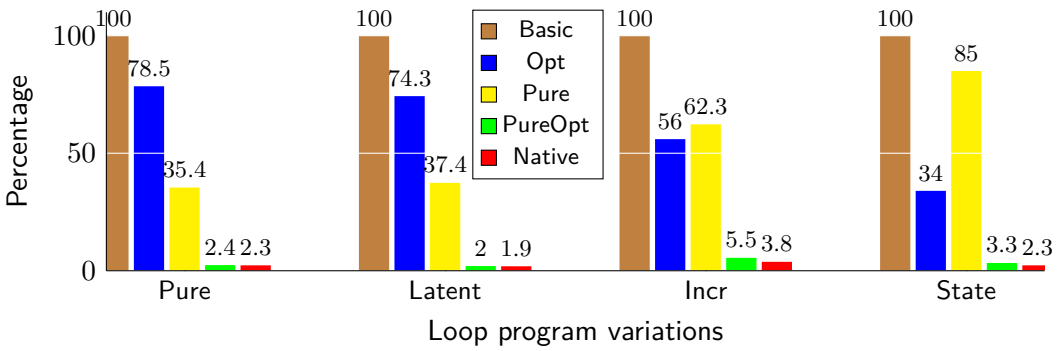

Figure 2.17: Relative run-times of Loops example [24]

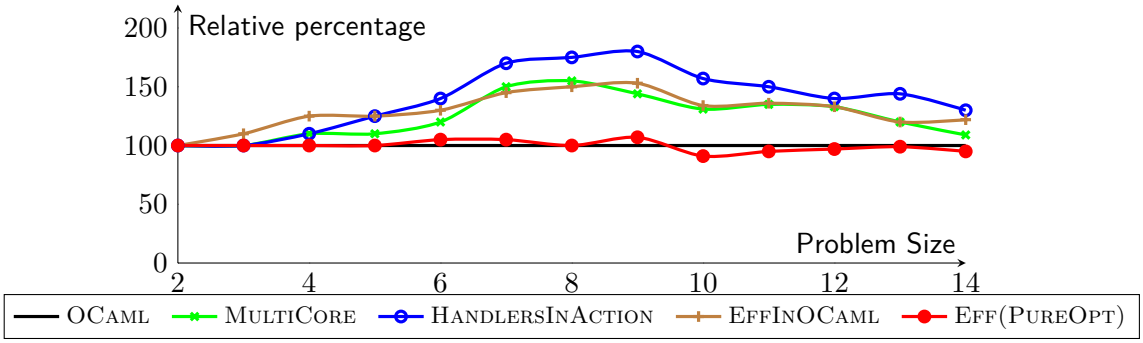

Figure 2.18: Results of running N-Queens for all solutions on multiple systems [24]

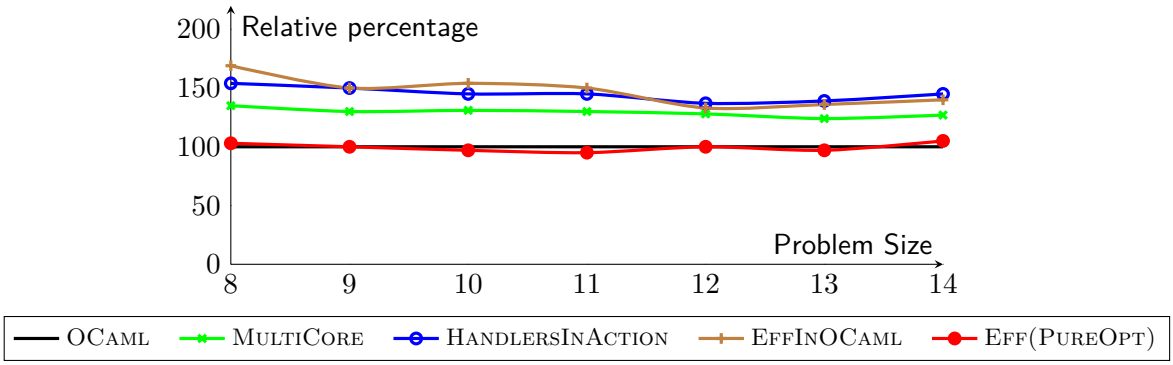

Figure 2.19: Results of running N-Queens for one solution on multiple systems [24]

2.3.4 Limitations

As said in the motivation of this chapter, there are some issues with the optimizations. The main problem is that the optimising compiler of Eff is very fragile. The main reason is that the subtyping system is unclear to work with. With the implementation of Eff, every term is annotated with a type. This type also contains subtyping constraints. Term-rewrite rules work, as indicated by the name, with terms. Transforming most terms is easy. For example, under the right circumstances when WITH-PURE is used, `handle c with h` is transformed into do $x \leftarrow c$; c_r. The only change that occurs is the "shape" of the term, but the type remains the same.

This is not always the case. With function specialization, there is the rule `handle f v with h` $\mapsto$ `let rec f' x = handle cf with h in f' v`. With this term rewrite rule, a new recursive function `f'` needs to be created which is a specialization of the existing function `f`. Thus the type of `f'` needs to be correctly calculated. Calculating this type not only requires the types of the different terms, but also the subtyping constraints. It is quite easy to make mistakes and calculate the wrong type. With the wrong type, further optimizations might not be executed, or compilation might go completely wrong and cause typing errors.

The main problem lies with the subtyping constraints and the implicit types of Eff. One possibility is to go to an explicitly typed calculus with Hindley-Milner based type inference. The effect system in such a system is based on row-typing [14, 8, 13]. Such an explicitly types calculus with row-based effects has been implemented in earlier research [7].

Another possibility lies with a coercion-based system. This system is an explicitly-typed calculus for algebraic effects and handlers with support for subtyping using coercion proofs. This system uses coercion proofs in order to make the subtyping constraints an explicit element of the terms of the language. [25]

2.3.5 Algebraic Subtyping

When looking at different possibilities, we noticed the PhD thesis of Stephen Dolan. [4]. Dolan provides a new subtyping based type system. While this system would not suffice for solving the issues with the optimizations, it does provide an interesting research direction for algebraic effects and handlers.

Algebraic Subtyping has support for subtyping, but eliminates the disadvantage of having constraints. By using union and intersection types, subtyping constraints are explicitly coded within a type. For example, `let twice f x = f (f x)` will be given the type: $(\beta \to \gamma) \to \alpha \to \gamma | \alpha \leqslant \beta, \gamma \leqslant \beta$. Algebraic subtyping will assign a different type to this term: $(\alpha \to \alpha \wedge \beta) \to \alpha \to \beta$. The two constraints are fully encoded within the type that the algebraic subtyping system infers.

The advantage of utilising algebraic subtyping compared to subtyping is mainly the full elimination of the subtyping. This has the result that types become smaller in size and much more readable for users. Thus such a system would also be an advantage within the context of algebraic effects and handlers.

The work in this thesis extends the algebraic subtyping system with algebraic effects and handlers. Chapter 3 explores the type system that algebraic subtyping provides and extends it in order to support algebraic effects and handlers. There will be clear mention of the novel elements, but the general guideline is that all algebraic effect and handler related work is novel, while the type system is not.

Chapter 4 extends the type inference algorithm from algebraic subtyping in order to support algebraic effects and handlers. The algorithm provided in the type inference chapter can infer very large types. These large types are not pleasent to look at and provide unnecessary overhead. Chapter 5 will extend simplification algorithms provided by the algebraic subtyping system.

Chapter 6 provides information about the implementation and difficulties that were found. The implementation was done within the Eff programming language. Chapter 6.5 evaluates the extended system using empirical evaluation. Correctness proofs have been added in Appendix A

Chapter 3

Algebraic Subtyping for Eff

This thesis proposes a type system which is based on the algebraic subtyping system, but is extended with algebraic effects. EffCore is the name that will be used for this system. Algebraic subtyping has support for subtyping, but eliminates the disadvantage of having constraints. By using union and intersection types, subtyping constraints are explicitly coded within a type. EffCore adds algebraic effects into this system while simultaneously still preserving the properties of algebraic subtyping.

In this chapter, comparisons will be made to both Eff and standard algebraic subtyping. This is due to the additional intent that Eff will be revised in order to support algebraic subtyping. In section 3.1, the types and terms of EffCore is given. In section 3.2 the equivalence between algebraic subtyping and regular subtyping will be explained. Section 3.3 provides the standard typing rules for EffCore and the final section, section 3.6, reformulates these typing rules into a proper representation for algebraic subtyping.

3.1 Types and terms

This section gives the terms and types of EffCore. A lot of the aspects that can be seen in this section are the required constructions needed for algebraic effects and for algebraic subtyping. The main point of interest, as well as the novelty, in this section is the representation that is given to the types of the dirt.

Terms Figure 3.1 shows the two kinds of terms in EffCore. Just like in Eff. there are values v and computations c. Computations are terms that can contain effects and these effects are denoted as operations Op which can be called.

The relevant change compared to Eff is that EffCore makes a distinction between let-bound variables and lambda-bound variables. This distinction was introduced by Dolan in order to simplify the algebraic subtyping approach [5]. By making this distinction, an explicit distinction can be made between monomorphic variables (lambda-bound) and polymorphic variables (let-bound) at the term level.

value v	$::=$	x	λ-variable
	$\mid$	$\hat{\mathbf{x}}$	let-variable
	$\mid$	`true`	true
	$\mid$	`false`	false
	$\mid$	$\lambda x.c$	function
	$\mid$	$\{$	handler
		`return` $x \mapsto c_r$,	return case
		$[\mathtt{Op}\ y\ k \mapsto c_{\mathtt{Op}}]_{\mathtt{Op} \in O}$	operation cases
		$\}$	
comp c	$::=$	$v_1\ v_2$	application
	$\mid$	`do` $\hat{\mathbf{x}} = c_1$; c_2	sequencing
	$\mid$	`let` $\hat{\mathbf{x}} = v$ `in` c	let
	$\mid$	`if` e `then` c_1 `else` c_2	conditional
	$\mid$	`return` v	returned val
	$\mid$	$\mathtt{Op}\ v$	operation call
	$\mid$	`handle` c `with` v	handling

Figure 3.1: Terms of EffCore

Types Figure 3.2 shows the types of EffCore. There are, like in Eff, two main sorts of types. There are (pure) types A, B and dirty types $\underline{C}, \underline{D}$. A dirty type is a pure type A tagged with a dirt Δ. The dirt represents the set of operations that can be called. It can also be an union or intersection of dirty types. The dirt Δ is an over-approximation of the operations that are actually called. In the next section, the relations between dirty intersections or unions and pure intersections or unions are explained.

The type $\underline{C} \Rightarrow \underline{D}$ is used for handlers because a handler takes an input computation $\underline{C}$, handles the effects in this computation and outputs computation $\underline{D}$ as the result. [19] The $\top$ and $\bot$ types are needed in order to properly compute the lattice of different types. Type intersections and type unions are also provided. [5]

Looking at all the changes to the types given in Chapter 2.2, the effects of the algebraic subtyping approach become apparant. Different types are added in order to support the subtyping. These are type variables, recursive type, top, bottom, intersection and union [5]. The novel element here is the combination of the algebraic effects and algebraic subtyping. There needs to be a way to bring the dirts into the algebraic subtyping framework. Since the recursive element is handled at the term level, there is no need for recursive dirts. Aside from this and the lack of a function and handler type, the dirts mirror the types.

typing contexts Γ	$::=$	$\epsilon \mid \Gamma, x : A \mid \Gamma, \hat{\mathbf{x}} : \forall \bar{\alpha}.B$	
monomorphic typing contexts Ξ	$::=$	$\epsilon \mid \Xi, x : A$	
polymorphic typing contexts Π	$::=$	$\epsilon \mid \Pi, \hat{\mathbf{x}} : [\Xi]A$	
(pure) type A, B	$::=$	$\texttt{bool}$	bool type
	$\mid$	$A \to \underline{C}$	function type
	$\mid$	$\underline{C} \Rightarrow \underline{D}$	handler type
	$\mid$	α	type variable
	$\mid$	$\mu\alpha.A$	recursive type
	$\mid$	$\top$	top
	$\mid$	$\bot$	bottom
	$\mid$	$A \sqcap B$	intersection
	$\mid$	$A \sqcup B$	union
dirty type $\underline{C}, \underline{D}$	$::=$	$A \,!\, \Delta$	
dirt Δ	$::=$	$\texttt{Op}$	operation
	$\mid$	δ	dirt variable
	$\mid$	$\emptyset$	empty dirt
	$\mid$	$\Delta_1 \sqcap \Delta_2$	intersection
	$\mid$	$\Delta_1 \sqcup \Delta_2$	union
All operations Ω	$::=$	$\bigsqcup \texttt{Op}_i \mid \texttt{Op}_i \in \Sigma$	

Figure 3.2: Types of EffCore

3.2 Equivalence to Subtyping

Algebraic subtyping is equivalent to standard subtyping constraints. [4] Figure 3.3 shows the equivalence rules between the different subtyping constraints and algebraic subtyping. $A_1 \leqslant A_2 \leftrightarrow A_1 \sqcup A_2 \equiv A_2$ shows that if a type A_1 is a subtype of A_2, then $A_1 \sqcup A_2$ is equivalent to A_2. This rule and the analogue of intersection are part of algebraic subtyping. The novel additions are $\Delta_1 \leqslant \Delta_2 \leftrightarrow \Delta_1 \sqcup \Delta_2 \equiv \Delta_2$ and $\Delta_1 \leqslant \Delta_2 \leftrightarrow \Delta_1 \equiv \Delta_1 \sqcap \Delta_2$. These rules show equivalence for dirts.

Figure 3.4 are equations of distributive lattices for types. These are entirely standard for algebraic subtyping. Figure 3.5 show the distributive lattices for dirts, which follow the format that algebraic subtyping uses for types. It is an important detail that the equations for the dirts mirror those of types, as we want the effect system to mirror the algebraic subtyping for types as much as possible.

Figure 3.6 shows equivalence rules for functions, handlers and dirty types. Equivalence rules regarding functions are exactly like they are in standard algebraic subtyping. Functions are contravariant in its argument types and covariant in its return types. This can also be seen in these equivalence equations. When we have a union of two function types, the argument types are intersected. When we have an intersection of two function

types,the argument types are unioned. Handlers are also contravariant in its argument types and covariant in its return types, and thus show the same behaviour.

The final two rules for dirty types $(\underline{C}_1 \sqcup \underline{C}_2) \equiv (A_1 \,!\, \Delta_1 \sqcup A_2 \,!\, \Delta_2) \equiv (A_1 \sqcup A_2) \,!\, (\Delta_1 \sqcup \Delta_2)$ and $(\underline{C}_1 \sqcap \underline{C}_2) \equiv (A_1 \,!\, \Delta_1 \sqcap A_2 \,!\, \Delta_2) \equiv (A_1 \sqcap A_2) \,!\, (\Delta_1 \sqcap \Delta_2)$ show the decomposition of dirty types into its two components, a type and a dirt. The union of two dirty types is equivalent to the union of its types and its dirts, and analogue for the intersection.

$$A_1 \leqslant A_2 \leftrightarrow A_1 \sqcup A_2 \equiv A_2$$

$$A_1 \leqslant A_2 \leftrightarrow A_1 \equiv A_1 \sqcap A_2$$

$$\Delta_1 \leqslant \Delta_2 \leftrightarrow \Delta_1 \sqcup \Delta_2 \equiv \Delta_2$$

$$\Delta_1 \leqslant \Delta_2 \leftrightarrow \Delta_1 \equiv \Delta_1 \sqcap \Delta_2$$

$$\underline{C}_1 \leqslant \underline{C}_2 \leftrightarrow \underline{C}_1 \sqcup \underline{C}_2 \equiv \underline{C}_2$$

$$\underline{C}_1 \leqslant \underline{C}_2 \leftrightarrow \underline{C}_1 \equiv \underline{C}_1 \sqcap \underline{C}_2$$

Figure 3.3: Relationship between Equivalence and Subtyping

$$A \sqcup A \equiv A \qquad A \sqcap A \equiv A$$

$$A_1 \sqcup A_2 \equiv A_2 \sqcup A_1 \qquad A_1 \sqcap A_2 \equiv A_2 \sqcap A_1$$

$$A_1 \sqcup (A_2 \sqcup A_3) \equiv (A_1 \sqcup A_2) \sqcup A_3 \qquad A_1 \sqcap (A_2 \sqcap A_3) \equiv (A_1 \sqcap A_2) \sqcap A_3$$

$$A_1 \sqcup (A_1 \sqcap A_2) \equiv A_1 \qquad A_1 \sqcap (A_1 \sqcup A_2) \equiv A_1$$

$$\bot \sqcup A \equiv A \qquad \bot \sqcap A \equiv \bot$$

$$\top \sqcup A \equiv \top \qquad \top \sqcap A \equiv A$$

$$A_1 \sqcup (A_2 \sqcap A_3) \equiv (A_1 \sqcup A_2) \sqcap (A_1 \sqcup A_3)$$

$$A_1 \sqcap (A_2 \sqcup A_3) \equiv (A_1 \sqcap A_2) \sqcup (A_1 \sqcap A_3)$$

Figure 3.4: Equations of distributive lattices for types

$$\Delta \sqcup \Delta \equiv \Delta \qquad \Delta \sqcap \Delta \equiv \Delta$$

$$\Delta_1 \sqcup \Delta_2 \equiv \Delta_2 \sqcup \Delta_1 \qquad \Delta_1 \sqcap \Delta_2 \equiv \Delta_2 \sqcap \Delta_1$$

$$\Delta_1 \sqcup (\Delta_2 \sqcup \Delta_3) \equiv (\Delta_1 \sqcup \Delta_2) \sqcup \Delta_3 \qquad \Delta_1 \sqcap (\Delta_2 \sqcap \Delta_3) \equiv (\Delta_1 \sqcap \Delta_2) \sqcap \Delta_3$$

$$\Delta_1 \sqcup (\Delta_1 \sqcap \Delta_2) \equiv \Delta_1 \qquad \Delta_1 \sqcap (\Delta_1 \sqcup \Delta_2) \equiv \Delta_1$$

$$\emptyset \sqcup \Delta \equiv \Delta \qquad \emptyset \sqcap \Delta \equiv \emptyset$$

$$\Omega \sqcup \Delta \equiv \Omega \qquad \Omega \sqcap \Delta \equiv \Delta$$

$$\Delta_1 \sqcup (\Delta_2 \sqcap \Delta_3) \equiv (\Delta_1 \sqcup \Delta_2) \sqcap (\Delta_1 \sqcup \Delta_3)$$

$$\Delta_1 \sqcap (\Delta_2 \sqcup \Delta_3) \equiv (\Delta_1 \sqcap \Delta_2) \sqcup (\Delta_1 \sqcap \Delta_3)$$

Figure 3.5: Equations of distributive lattices for dirts

$$(A_1 \to \underline{C}_1) \sqcup (A_2 \to \underline{C}_2) \equiv (A_1 \sqcap A_2) \to (\underline{C}_1 \sqcup \underline{C}_2)$$

$$(A_1 \to \underline{C}_1) \sqcap (A_2 \to \underline{C}_2) \equiv (A_1 \sqcup A_2) \to (\underline{C}_1 \sqcap \underline{C}_2)$$

$$(A_1 \Rightarrow \underline{C}_1) \sqcup (A_2 \Rightarrow \underline{C}_2) \equiv (A_1 \sqcap A_2) \Rightarrow (\underline{C}_1 \sqcup \underline{C}_2)$$

$$(A_1 \Rightarrow \underline{C}_1) \sqcap (A_2 \Rightarrow \underline{C}_2) \equiv (A_1 \sqcup A_2) \Rightarrow (\underline{C}_1 \sqcap \underline{C}_2)$$

$$(\underline{C}_1 \sqcup \underline{C}_2) \equiv (A_1 \,!\, \Delta_1 \sqcup A_2 \,!\, \Delta_2) \equiv (A_1 \sqcup A_2) \,!\, (\Delta_1 \sqcup \Delta_2)$$

$$(\underline{C}_1 \sqcap \underline{C}_2) \equiv (A_1 \,!\, \Delta_1 \sqcap A_2 \,!\, \Delta_2) \equiv (A_1 \sqcap A_2) \,!\, (\Delta_1 \sqcap \Delta_2)$$

Figure 3.6: Equations for function, handler and dirty types

3.3 Typing rules

Figure 3.7 defines the typing judgements for values and computations with respect to a standard typing context Γ. Most of the rules are standard. It is important to note that the typing context Γ contains variables with monomorphic types A and variables with polymorphic type schemes.

Values The rules for subtyping, constants, variables and functions are entirely standard. The difference between λ-variables x and let-variables $\hat{\mathbf{x}}$ becomes much more clear in these rules. λ-variables x means that the variable is bound by a λ abstraction, its type is monomorphic. In contrast, let-variables $\hat{\mathbf{x}}$ are bound by let constructs. Rule Var-$\forall$ shows that $\hat{\mathbf{x}}$ is polymorphic and is given a polymorphic type scheme $\forall\bar{\alpha}.A$.

A handler expression has type $A \,!\, \Delta \sqcap \mathcal{O} \Rightarrow B \,!\, \Delta$. An interesting detail is dirt of the argument of the handler $\Delta \sqcap \mathcal{O}$. The reason for the choice to use a $\sqcap$ is non-trivial. In chapter 4.1, the reason is explored more formally. Unformallly, the reason is that, in general, argument types utilise the intersection while return types utilise unions. Note that the intersection $\Delta \cap \mathcal{O}$ is not necessarily empty (with $\cap$ being the intersection of the operations, not to be confused with the $\sqcap$ type). The handler deals with the operations $\mathcal{O}$, but in the process may reintrodcue some of them.

Computations The rules for subtyping, application, conditional, return, operation, let and with have a straightforward definition. The interesting rule for the computations is Do. In the Do rule, the type aspect of the *do* computation is B, which is the same type as that of c_2. However, the dirt of the *do* computation is the union of the dirt from c_1 and the dirt from c_2. Side effects may occur even without the explicit usage of the variable $\hat{\mathbf{x}}$, thus we need to explicitly keep track of those specific side effects. This is done by taking the union.

3.4 Properties of the type system

Since the typing rules of EffCore mirror those from algebraic subtyping and from eff which are both based on the typing rules from ML, the familiar properties of those type systems also hold for EffCore.

3.4.1 Instantiation

explain proof instantiation

3.4.2 Weakening

explain proof weakening

3.4.3 Substitution

explain proof substitution

3.4.4 Soundness

explain proof soundness

3.5 Typing schemes and subsumption

Currently, the typing context Γ contains variables with monomorphic types A and variables with polymorphic type schemes. Separating this into two typing contexts, one containing variables with monomorphic types A and one containing variables with polymorphic type schemes will make the type inference easier. Thus, we need to reformulate the typing rules. [4]

Before we can reformulate the typing rules, there are some important concepts to introduce. Subsumption is the analogue of subtyping between two type schemes. Subsumption is an interesting case as the $\leqslant$ relation can be used between two environments when they assign alpha-equivalent type schemes to let-bound variables. Considering that the typing environments in EffCore only contains pure types and pure typing schemes, the typing environment is exactly alike to the typing environment from standard algebraic subtyping. This section therefore summarizes the required definitions from algebraic subtyping. Proofs for the subsumption are also not given in this thesis, [4]

The important definitions are given in Figure 3.8. Definition ΞDef shows that we use $ctxm$ as a typing context that only contains free λ-bound variables with monomorphic types. We also need the concept of typing schemes. A typing scheme is a pair of a monomorphic typing context Ξ and a type A. SubScheme and SubSchemeDirty extends the subtyping relation to typing schemes. The subtyping relation is covariant in the type A and contravariant in the typing context Ξ.

SubScheme requires both typing contexts to use exactly the same type schemes, instead of alpha-equivalent type schemes. $\leqslant^{\forall}$ is introduced in definition SubInst. This definitoon states that $[\Xi_2]A_2$ subsumes $[\Xi_1]A_1$ if there is some substitution ρ (that instantiates both type and dirt variables) for $[\Xi_2]A_2$ such that $\rho([\Xi_2]A_2) \leqslant [\Xi_1]A_1$. Definition SubstEq shows how a substitution is applied to a typing scheme. SubInstDirty is to SubSchemeDirty what SubInst is to SubScheme.

Two monomorphic typing contexts Ξ_1 and Ξ_2 have a greatest lower bound: $\Xi_1 \sqcap \Xi_2$ where $\mathrm{dom}(\Xi_1 \sqcap \Xi_2) = \mathrm{dom}(\Xi_1) \cup \mathrm{dom}(\Xi_2)$, and $(\Xi_1 \sqcap \Xi_2)(x) = \Xi_1(x) \sqcap \Xi_2(x)$, interpreting $\Xi_i(x) = \top$ if $x \in dom(\Xi_i)$ (for $i \in \{1, 2\}$). [4]

Two monomorphic typing contexts are equivalent if they subsume each other as seen in Eq. Definition WeakeningMono more concretely defines the $\leqslant^{\forall}$ in terms of the domain of the monomorphic typing contexts.

For polymorphic typing contexts, Π is used. Definition WeakeningPoly shows the $\leqslant^{\forall}$ in terms of the domain of the polymorphic typing contexts.

An interesting detail about the typing schemes is that they can be equivalent by $\equiv^{\forall}$ without being alpha-equivalent. Disregarding effects, let's take the `choose` function, which takes two arguments and returns one of the two randomly. With the Hindley-Milner type system, we would infer the typing scheme $\alpha \to \alpha \to \alpha$. Algebraic subtyping

will infer the typing scheme $\alpha \to \beta \to (\alpha \sqcup \beta)$. These two typing schemes are not alpha-equivalent. The second typing scheme subsumes the first by instantiation and the first typing scheme subsumes the second. Thus both typing schemes are equivalent by $\equiv^{\forall}$, but not alpha-equivalent.

3.6 Reformulated typing rules

Now, we can reformulate the typing rules from Figure 3.7. The reformulated typing rules are given in Figure 3.9. The SubVal and SubComp rules are used for both subtyping and instantiation. This is due to the $\leqslant^{\forall}$ relation.

Most rules are straightforward extensions from the reformulated typing rules from algebraic subtyping. The (polymorphic) typing context Π used for the reformulated rules assign typing schemes to let-bound variables. The reformulated typing rules are an alternative, but equivalent representation of the typing rules. The type inference algorithm described in Chapter 4 uses the reformulated typing rules.

3.6.1 Equivalence of original and reformulated typing rules

explain proof reformulated typing judgements

typing contexts $\Gamma ::= \epsilon \mid \Gamma, x : A \mid \Gamma, \hat{\mathbf{x}} : \forall \bar{\alpha}.B$

Expressions

$$\textsc{SubVal}\ \frac{\Gamma \vdash v : A \qquad A \leqslant B}{\Gamma \vdash v : B} \qquad \textsc{Var-}\lambda\ \frac{(x : A) \in \Gamma}{\Gamma \vdash x : A} \qquad \textsc{Var-}\forall\ \frac{(\hat{\mathbf{x}} : \forall \bar{\alpha}.A) \in \Gamma}{\Gamma \vdash \hat{\mathbf{x}} : A[\bar{A}/\bar{\alpha}]} \qquad \textsc{True}\ \frac{}{\Gamma \vdash \texttt{true} : \mathit{bool}}$$

$$\textsc{False}\ \frac{}{\Gamma \vdash \texttt{false} : \mathit{bool}} \qquad \textsc{Fun}\ \frac{\Gamma, x : A \vdash c : \underline{C}}{\Gamma \vdash \lambda x.c : A \to \underline{C}}$$

$$\textsc{Hand}\ \frac{\Gamma, x : A \vdash c_r : B\ !\ \Delta \qquad \left[(\texttt{Op} : A_{\texttt{Op}} \to B_{\texttt{Op}}) \in \Sigma \qquad \Gamma, y : A_{\texttt{Op}}, k : B_{\texttt{Op}} \to B\ !\ \Delta \vdash c_{\texttt{Op}} : B\ !\ \Delta\right]_{\texttt{Op} \in O}}{\Gamma \vdash \{\texttt{return}\ x \mapsto c_r, [\texttt{Op}\ y\ k \mapsto c_{\texttt{Op}}]_{\texttt{Op} \in O}\} : \qquad A\ !\ \Delta \sqcap O \Rightarrow B\ !\ \Delta}$$

Computations

$$\textsc{SubComp}\ \frac{\Gamma \vdash c : \underline{C} \qquad \underline{C} \leqslant \underline{D}}{\Gamma \vdash c : \underline{D}} \qquad \textsc{App}\ \frac{\Gamma \vdash v_1 : A \to \underline{C} \qquad \Gamma \vdash v_2 : A}{\Gamma \vdash v_1\, v_2 : \underline{C}}$$

$$\textsc{Cond}\ \frac{\Gamma \vdash v : \mathit{bool} \qquad \Gamma \vdash c_1 : \underline{C} \qquad \Gamma \vdash c_2 : \underline{C}}{\Gamma \vdash \texttt{if}\ v\ \texttt{then}\ c_1\ \texttt{else}\ c_2 : \underline{C}} \qquad \textsc{Ret}\ \frac{\Gamma \vdash v : A}{\Gamma \vdash \texttt{return}\ v : A\ !\ \emptyset}$$

$$\textsc{Op}\ \frac{(\texttt{Op} : A \to B) \in \Sigma \qquad \Gamma \vdash v : A}{\Gamma \vdash \texttt{Op}\, v : B\ !\ \texttt{Op}}$$

$$\textsc{Let}\ \frac{\Gamma \vdash v : A \qquad \Gamma, \hat{\mathbf{x}} : \forall \bar{\alpha}.A \vdash c : B\ !\ \Delta \qquad \alpha \notin \mathit{FTV}(\Gamma)}{\Gamma \vdash \texttt{let}\ \hat{\mathbf{x}} = v\ \texttt{in}\ c : B\ !\ \Delta}$$

$$\textsc{Do}\ \frac{\Gamma \vdash c_1 : A\ !\ \Delta_1 \qquad \Gamma, \hat{\mathbf{x}} : \forall \bar{\alpha}.A \vdash c_2 : B\ !\ \Delta_2 \qquad \alpha \notin \mathit{FTV}(\Gamma)}{\Gamma \vdash \texttt{do}\ \hat{\mathbf{x}} = c_1\ ;\ c_2 : B\ !\ (\Delta_1 \sqcup \Delta_2)}$$

$$\textsc{With}\ \frac{\Gamma \vdash v : \underline{C} \Rightarrow \underline{D} \qquad \Gamma \vdash c : \underline{C}}{\Gamma \vdash \texttt{handle}\ c\ \texttt{with}\ v : \underline{D}}$$

Figure 3.7: Typing of EffCore

ΞDef
Ξ contains free λ-bound variables

SubScheme
$[\Xi_2]A_2 \leqslant [\Xi_1]A_1 \leftrightarrow A_2 \leqslant A_1, \Xi_1 \leqslant \Xi_2$

SubSchemeDirty
$[\Xi_2]\underline{C}_2 \leqslant [\Xi_1]\underline{C}_1 \leftrightarrow \underline{C}_2 \leqslant \underline{C}_1, \Xi_1 \leqslant \Xi_2$

SubInst
$[\Xi_2]A_2 \leqslant^{\forall} [\Xi_1]A_1 \leftrightarrow \rho([\Xi_2]A_2) \leqslant [\Xi_1]A_1$ for some substitution ρ
(instantiate type and dirt variables)

SubInstDirty
$[\Xi_2]\underline{C}_2 \leqslant^{\forall} [\Xi_1]\underline{C}_1 \leftrightarrow \rho([\Xi_2]\underline{C}_2) \leqslant [\Xi_1]\underline{C}_1$ for some substitution ρ
(instantiate type and dirt variables)

SubstEq
$\rho([\Xi]A) \equiv [\rho(\Xi)]\rho(A)$

Eq
$[\Xi_2]A_2 \equiv^{\forall} [\Xi_1]A_1 \leftrightarrow [\Xi_2]A_2 \leqslant^{\forall} [\Xi_1]A_1, [\Xi_1]A_1 \leqslant^{\forall} [\Xi_2]A_2$

WeakeningMono
$\Xi_2 \leqslant^{\forall} \Xi_1 \leftrightarrow dom(\Xi_2) \supseteq dom(\Xi_1), \Xi_2(x) \leqslant^{\forall} \Xi_1(x) \mid x \in dom(\Xi_1)$

WeakeningPoly
$\Pi_2 \leqslant^{\forall} \Pi_1 \leftrightarrow dom(\Pi_2) \supseteq dom(\Pi_1), \Pi_2(\hat{\mathbf{x}}) \leqslant^{\forall} \Pi_1(\hat{\mathbf{x}}) \mid \hat{\mathbf{x}} \in dom(\Pi_1)$

Figure 3.8: Definitions for typing schemes and reformulated typing rules

$$\text{monomorphic typing contexts } \Xi ::= \epsilon \mid \Xi, x : A$$
$$\text{polymorphic typing contexts } \Pi ::= \epsilon \mid \Pi, \hat{\mathbf{x}} : [\Xi]A$$

Expressions

$$\textsc{SubVal}\ \frac{\Pi \Vdash v : [\Xi_1]A_1 \qquad [\Xi_1]A_1 \leqslant^{\forall} [\Xi_2]A_2}{\Pi \Vdash v : [\Xi_2]A_2} \qquad \textsc{Var-}\Xi\ \frac{}{\Pi \Vdash x : [x : A]A} \qquad \textsc{Var-}\Pi\ \frac{(\hat{\mathbf{x}} : [\Xi]A) \in \Pi}{\Pi \Vdash \hat{\mathbf{x}} : [\Xi]A}$$

$$\textsc{True}\ \frac{}{\Pi \Vdash \texttt{true} : []bool} \qquad \textsc{False}\ \frac{}{\Pi \Vdash \texttt{false} : []bool} \qquad \textsc{Fun}\ \frac{\Pi \Vdash c : [\Xi, x : A]\underline{C}}{\Pi \Vdash \lambda x.c : [\Xi](A \to \underline{C})}$$

$$\textsc{Hand}\ \frac{\Pi \Vdash c_r : [\Xi, x : A](B\ !\ \Delta) \qquad \left[(\texttt{Op} : A_{\texttt{Op}} \to B_{\texttt{Op}}) \in \Sigma \qquad \Pi \Vdash c_{\texttt{Op}} : [\Xi, y : A_{\texttt{Op}}, k : B_{\texttt{Op}} \to B\ !\ \Delta](B\ !\ \Delta)\right]_{\texttt{Op} \in O}}{\Pi \Vdash \{\texttt{return}\ x \mapsto c_r, [\texttt{Op}\ y\ k \mapsto c_{\texttt{Op}}]_{\texttt{Op} \in O}\} : [\Xi](A\ !\ \Delta \sqcap O \Rightarrow B\ !\ \Delta)}$$

Computations

$$\textsc{SubComp}\ \frac{\Pi \Vdash c : [\Xi_1]\underline{C}_1 \qquad [\Xi_1]\underline{C}_1 \leqslant^{\forall} [\Xi_2]\underline{C}_2}{\Pi \Vdash c : [\Xi_2]\underline{C}_2} \qquad \textsc{App}\ \frac{\Pi \Vdash v_1 : [\Xi](A \to \underline{C}) \qquad \Pi \Vdash v_2 : [\Xi]A}{\Pi \Vdash v_1\ v_2 : [\Xi]\underline{C}}$$

$$\textsc{Cond}\ \frac{\Pi \Vdash v : [\Xi]bool \qquad \Pi \Vdash c_1 : [\Xi]\underline{C} \qquad \Pi \Vdash c_2 : [\Xi]\underline{C}}{\Pi \Vdash \texttt{if}\ v\ \texttt{then}\ c_1\ \texttt{else}\ c_2 : [\Xi]\underline{C}}$$

$$\textsc{Ret}\ \frac{\Pi \Vdash v : [\Xi]A}{\Pi \Vdash \texttt{return}\ v : [\Xi](A\ !\ \emptyset)} \qquad \textsc{Op}\ \frac{(\texttt{Op} : A \to B) \in \Sigma \qquad \Pi \Vdash v : [\Xi]A}{\Pi \Vdash \texttt{Op}\ v : [\Xi](B\ !\ \texttt{Op})}$$

$$\textsc{Let}\ \frac{\Pi \Vdash v : [\Xi_1]A \qquad \Pi, \hat{\mathbf{x}} : [\Xi_1]A \Vdash c : [\Xi_2](B\ !\ \Delta)}{\Gamma \Vdash \texttt{let}\ \hat{\mathbf{x}} = v\ \texttt{in}\ c : [\Xi_1 \sqcap \Xi_2](B\ !\ \Delta)}$$

$$\textsc{Do}\ \frac{\Pi \Vdash c_1 : [\Xi_1](A\ !\ \Delta_1) \qquad \Pi, \hat{\mathbf{x}} : [\Xi_1]A \Vdash c_2 : [\Xi_2](B\ !\ \Delta_2)}{\Gamma \Vdash \texttt{do}\ \hat{\mathbf{x}} = c_1\ ;\ c_2 : [\Xi_1 \sqcap \Xi_2](B\ !\ (\Delta_1 \sqcup \Delta_2))}$$

$$\textsc{With}\ \frac{\Pi \Vdash v : [\Xi](\underline{C} \Rightarrow \underline{D}) \qquad \Pi \Vdash c : [\Xi]\underline{C}}{\Pi \Vdash \texttt{handle}\ c\ \texttt{with}\ v : [\Xi]\underline{D}}$$

Figure 3.9: Reformulated typing rules of EffCore

Chapter 4

Type Inference

This chapter describes the type inference algorithm for EffCore. Type inference has been brievely explained in Chapter 2.2.4. It is the process where types are automatically inferred by the compiler. Typically, a type inference algorithm uses contraint-based type inference rules. For EffCore, we cannot immediately jump into these rules. There are quite a few steps to go through first.

Typically, a type inference algortihm uses unification to solve the constraints generated by the type inference algorithm. In the case of a Hindley-Milner system, we can solve all rules generated by the type inference algorithm. With subtyping, not all rules are solved and a type may still contain constraints after inference has been completed. Algebraic subtyping uses an analogue for unification, called biunification, to encode the constraints into the intersection and union types.

In order to use biunification, we require polar types. The idea of polar types is to distinguish between input types and output types. Input types are used to describe inputs, while output types are used to describe outputs. Input and output types are represented as, respectively, negative and positive types in the terms of polar types.

If we ever have a program in which we have to choose to produce an output of type $\underline{C}_1$ or $\underline{C}_2$, the actual output type is $\underline{C}_1 \sqcup \underline{C}_2$. This could happen in the case of an `match` or `if` statement. In other situations, we may be in a situation where an input is used in a case where type $\underline{C}_1$ is required and in a case where type $\underline{C}_2$ is required, then the input is given $\underline{C}_1 \sqcap \underline{C}_2$ as its type. Equivalently, this can also happen with pure types A_1 and A_2.

In other words, input types only use intersections and output types only use unions. Polar types do not allow this convention to be broken. Making this restriction simplifies the problem of solving subtyping constraints greatly and allow them to be solved. [4, 20]

4.1 Polar types

Figure 4.1 shows the polar type of EffCore. The separation of the union $\sqcup$ and the intersection $\sqcap$ types can be clearly seen. With the exception of the handler type, all types are equivalent to the system of algebraic subtyping.

explain recursive types?

Polarity also extends to typing schemes as seen in the typing context Π. A polar typing scheme $[\Xi^-]A^+$ has a positive type A^+ and a monomorphic environment consisting of λ-bound variables with negative types.

The inference algorithm works only with polar typing schemes. The reader may wonder whether polar typing schemes are enough to infer a principal typing scheme. Dolan has shown that polar typing schemes do suffice, I extend his proof to incorporate algebraic effects and handlers in Chapter 4.3. [4]

4.2 Unification

To operate on polar type terms, we generalise from substitutions to bisubstitutions, which map type variables to a pair of a positive and a negative type term. The definitions for bisubstitions are given in Figure 4.2.

explain bisubstitions

explain constraint solving

explain constraint decomposition

explain biunification algorithm

4.3 Principal Type Inference

We introduce a judgement form $\Pi \triangleright e : [\Xi^-]A^+$, stating that $[\Xi^-]A^+$ is the principal typing scheme of e under the polar typing context Π.

explain type inference

polymorphic typing contexts Π	$::=$	$\epsilon \mid \Pi, \hat{\mathbf{x}} : [\Xi^-]A^+$	
(pure) type A^+, B^+	$::=$	`bool`	bool type
	$\mid$	$A^- \to \underline{C}^+$	function type
	$\mid$	$\underline{C}^- \Rightarrow \underline{D}^+$	**handler type**
	$\mid$	α	type variable
	$\mid$	$\mu\alpha.A^+$	recursive type
	$\mid$	$\bot$	bottom
	$\mid$	$A^+ \sqcup B^+$	union
dirty type $\underline{C}^+, \underline{D}^+$	$::=$	$A^+ \,!\, \Delta^+$	
(pure) type A^-, B^-	$::=$	`bool`	bool type
	$\mid$	$A^+ \to \underline{C}^-$	function type
	$\mid$	$\underline{C}^+ \Rightarrow \underline{D}^-$	**handler type**
	$\mid$	α	type variable
	$\mid$	$\mu\alpha.A^-$	recursive type
	$\mid$	$\top$	top
	$\mid$	$A^- \sqcap B^-$	intersection
dirty type $\underline{C}^-, \underline{D}^-$	$::=$	$A^- \,!\, \Delta^-$	
dirt Δ^+	$::=$	`Op`	operation
	$\mid$	δ	dirt variable
	$\mid$	$\emptyset$	empty dirt
	$\mid$	$\Delta_1^+ \sqcup \Delta_2^+$	union
dirt Δ^-	$::=$	`Op`	operation
	$\mid$	δ	dirt variable
	$\mid$	Ω	full dirt (all operations, top)
	$\mid$	$\Delta_1^- \sqcap \Delta_2^-$	intersection

Figure 4.1: Polar types of EffCore

Bisubstitution

$$\xi = [A^+/\alpha^+, A^-/\alpha^-, \Delta^+/\delta^+, \Delta^-/\delta^-]$$

$$\xi'(\alpha^+) = \alpha \qquad \xi'(\alpha^-) = \alpha \qquad \xi'(\delta^+) = \delta \qquad \xi'(\delta^-) = \delta \qquad \xi'(\_) = \_$$

$$\xi(\underline{C}^+) \equiv \xi(A^+ \mathbin{!} \Delta^+) \equiv \xi(A^+) \mathbin{!} \xi(\Delta^+) \qquad \xi(\underline{C}^-) \equiv \xi(A^- \mathbin{!} \Delta^-) \equiv \xi(A^-) \mathbin{!} \xi(\Delta^-)$$

$$\xi(\Delta_1^+ \sqcup \Delta_2^+) \equiv \xi(\Delta_1^+) \sqcup \xi(\Delta_2^+) \qquad \xi(\Delta_1^- \sqcap \Delta_2^-) \equiv \xi(\Delta_1^-) \sqcap \xi(\Delta_2^-)$$

$$\xi(\mathtt{Op}) \equiv \mathtt{Op} \qquad \xi(\mathtt{Op}) \equiv \mathtt{Op}$$

$$\xi(\emptyset) \equiv \emptyset \qquad \xi(\Omega) \equiv \Omega$$

$$\xi(A_1^+ \sqcup A_2^+) \equiv \xi(A_1^+) \sqcup \xi(A_2^+) \qquad \xi(A_1^- \sqcap A_2^-) \equiv \xi(A_1^-) \sqcap \xi(A_2^-)$$

$$\xi(\bot) \equiv \bot \qquad \xi(\top) \equiv \top$$

$$\xi(bool) \equiv bool \qquad \xi(bool) \equiv bool$$

$$\xi(A^- \to A^+) \equiv \xi(A^-) \to \xi(A^+) \qquad \xi(A^+ \to A^-) \equiv \xi(A^+) \to \xi(A^-)$$

$$\xi(A^- \Rightarrow A^+) \equiv \xi(A^-) \Rightarrow \xi(A^+) \qquad \xi(A^+ \Rightarrow A^-) \equiv \xi(A^+) \Rightarrow \xi(A^-)$$

$$\xi(\mu\alpha.A^+) \equiv \mu\alpha.\xi'(A^+) \qquad \xi(\mu\alpha.A^-) \equiv \mu\alpha.\xi'(A^-)$$

Figure 4.2: Bisubstitutions

Constructed (predicate):
constructed(A)

$$constructed(A \to \underline{C})$$

$$constructed(\underline{C} \Rightarrow \underline{D})$$

$$constructed(bool)$$

Figure 4.3: Constructed types

Atomic (partial function):
atomic$(A^+ \leqslant A^-) = \theta$, **atomic**$(\Delta^+ \leqslant \Delta^-) = \theta$

$$\frac{constructed(A^-) \qquad \beta \text{ not free in } A^-}{atomic(\beta \leqslant A^-) = [\beta \sqcap A^-/\beta^-]}$$

$$\frac{constructed(A^-) \qquad \beta \text{ free in } A^-}{atomic(\beta \leqslant A^-) = [\mu^-\alpha.(\beta \sqcap [\alpha/\beta^-](A^-))/\beta^-]}$$

$$\frac{constructed(A^+) \qquad \beta \text{ not free in } A^-}{atomic(A^+ \leqslant \beta) = [\beta \sqcup A^+/\beta^+]}$$

$$\frac{constructed(A^+) \qquad \beta \text{ free in } A^-}{atomic(A^+ \leqslant \beta) = [\mu^+\alpha.(\beta \sqcup [\alpha/\beta^+](A^+))/\beta^+]}$$

$$atomic(\beta \leqslant \gamma) = [\mu^-\alpha.(\beta \sqcap [\alpha/\beta^-](\gamma))/\beta^-] \equiv [\beta \sqcap \gamma/\beta^-] \equiv [\beta \sqcup \gamma/\gamma^+]$$

$$atomic(\delta \leqslant \mathtt{0p}) = [\delta \sqcap \mathtt{0p}/\delta^-]$$

$$atomic(\mathtt{0p} \leqslant \delta) = [\delta \sqcup \mathtt{0p}/\delta^+]$$

$$atomic(\delta_1 \leqslant \delta_2) = [\delta_1 \sqcup \delta_2/\delta_2^+] \equiv [\delta_1 \sqcap \delta_2/\delta_1^-]$$

Figure 4.4: Constraint solving

Subi (partial function):
subi$(A^+ \leqslant A^-) = C$, **subi**$(\Delta^+ \leqslant \Delta^-) = C$, **subi**$(\underline{C}^+ \leqslant \underline{C}^-) = C$

$$subi(A^+ \,!\, \Delta^+ \leqslant A^- \,!\, \Delta^-) = \{A^+ \leqslant A^-, \Delta^+ \leqslant \Delta^-\}$$

$$subi(A_1^- \to \underline{C}_1^+ \leqslant A_2^+ \to \underline{C}_2^-) = \{A_2^+ \leqslant A_1^-, \underline{C}_1^+ \leqslant \underline{C}_2^-\}$$

$$subi(\underline{C}_1^- \Rightarrow \underline{D}_1^+ \leqslant \underline{C}_2^+ \Rightarrow \underline{D}_2^-) = \{\underline{C}_2^+ \leqslant \underline{C}_1^-, \underline{D}_1^+ \leqslant \underline{D}_2^-\}$$

$$subi(bool \leqslant bool) = \{\}$$

$$subi(\mu\alpha.A^+ \leqslant A-) = \{[\mu\alpha.A^+/\alpha](A^+) \leqslant A^-\}$$

$$subi(A^+ \leqslant \mu\alpha.A^-) = \{A^+ \leqslant [\mu\alpha.A^-/\alpha]A^-\}$$

$$subi(A_1^+ \sqcup A_2^+ \leqslant A^-) = \{A_1^+ \leqslant A^-, A_2^+ \leqslant A^-\}$$

$$subi(A^+ \leqslant A_1^- \sqcap A_2^-) = \{A^+ \leqslant A_1^-, A^+ \leqslant A_2^-\}$$

$$subi(\bot \leqslant A^-) = \{\}$$

$$subi(A^+ \leqslant \top) = \{\}$$

$$subi(\mathtt{Op} \leqslant \mathtt{Op}) = \{\}$$

$$subi(\Delta_1^+ \sqcup \Delta_2^+ \leqslant \Delta^-) = \{\Delta_1^+ \leqslant \Delta^-, \Delta_2^+ \leqslant \Delta^-\}$$

$$subi(\mathtt{Op} \leqslant \mathtt{Op} \sqcap \Delta^-) = \{\}$$

$$subi(\mathtt{Op} \leqslant \Delta^- \sqcap \mathtt{Op}) = \{\}$$

$$subi(\mathtt{Op}_1 \leqslant \mathtt{Op}_2 \sqcap \Delta^-) = \{\mathtt{Op}_1 \leqslant \Delta^-\}$$

$$subi(\mathtt{Op}_1 \leqslant \Delta^- \sqcap \mathtt{Op}_2) = \{\mathtt{Op}_1 \leqslant \Delta^-\}$$

$$subi(\Delta^+ \leqslant \Delta_1^- \sqcap \Delta_2^-) = \{\Delta^+ \leqslant \Delta_1^-, \Delta^+ \leqslant \Delta_2^-\}$$

$$subi(\emptyset \leqslant \Delta^-) = \{\}$$

$$subi(\Delta^+ \leqslant \Omega) = \{\}$$

Figure 4.5: Constraint decomposition

Binunify(History, ContraintSet) = substitution

$$\textsc{Start}\quad biunify(C) = biunify(\emptyset; C)$$

$$\textsc{Empty}\quad biunify(H; \epsilon) = 1$$

$$\textsc{Redundant}\quad \frac{c \in H}{biunify(H; c :: C) = biunify(H; C)}$$

$$\textsc{Atomic}\quad \frac{atomic(c) = \theta_c}{biunify(H; c :: C) = biunify(\theta_c(H \cup \{c\}); \theta_c(C)) \cdot \theta_c}$$

$$\textsc{Decompose}\quad \frac{subi(c) = C'}{biunify(H; c :: C) = biunify(H \cup \{c\}; C' +\!\!+ C)}$$

Figure 4.6: Biunification algorithm

$$\text{monomorphic typing contexts } \Xi^- ::= \epsilon \mid \Xi^-, x : A^-$$
$$\text{polymorphic typing contexts } \Pi ::= \epsilon \mid \Pi, \hat{\mathbf{x}} : [\Xi^-]A^+$$

Expressions

$$\textsc{Var-}\Xi\ \frac{}{\Pi \triangleright x : [x : \alpha]\alpha} \qquad \textsc{Var-}\Pi\ \frac{(\hat{\mathbf{x}} : [\Xi^-]A^+) \in \Pi}{\Pi \triangleright \hat{\mathbf{x}} : [\Xi^-]A^+} \qquad \textsc{True}\ \frac{}{\Pi \triangleright \texttt{true} : []bool}$$

$$\textsc{False}\ \frac{}{\Pi \triangleright \texttt{false} : []bool} \qquad \textsc{Fun}\ \frac{\Pi \triangleright c : [\Xi^-]\underline{C}^+}{\Pi \triangleright \lambda x.c : [\Xi^- \setminus x](\Xi^-(x) \to \underline{C}^+)}$$

$$\textsc{Hand}\ \frac{\Pi \Vdash c_r : [\Xi_r^-](B^+ \,!\, \Delta^+) \qquad \left[(\texttt{Op} : A^+_{\texttt{Op}} \to B^-_{\texttt{Op}}) \in \Sigma \qquad \Pi \Vdash c_{\texttt{Op}} : [\Xi^-_{\texttt{Op}}](\underline{C}^+_{\texttt{Op}})\right]_{\texttt{Op} \in O}}{\begin{array}{c}\Pi \Vdash \{\texttt{return}\ x \mapsto c_r, [\texttt{Op}\ y\ k \mapsto c_{\texttt{Op}}]_{\texttt{Op} \in O}\} : \\ [\Xi_r^- \sqcap (\bigsqcap \Xi^-_{\texttt{Op}} | \texttt{Op} \in O)](\alpha_1 \,!\, \delta_1 \sqcap O \Rightarrow \alpha_2 \,!\, \delta_2)\end{array}}$$

$$\xi = biunify \left(\begin{array}{l} B^+ \,!\, \Delta^+ \leqslant \alpha_2 \,!\, \delta_2 \\ \alpha_1 \leqslant \Xi_r^-(x) \\ \delta_1 \leqslant \delta_2 \\ \left[\begin{array}{l} A^+_{\texttt{Op}} \leqslant \Xi^-_{\texttt{Op}}(y) \\ B^-_{\texttt{Op}} \to \underline{C}^+_{\texttt{Op}} \leqslant \Xi^-_{\texttt{Op}}(k) \\ \underline{C}^+_{\texttt{Op}} \leqslant \alpha_2 \,!\, \delta_2 \end{array} \right]_{\texttt{Op} \in O} \end{array} \right)$$

Figure 4.7: Type inference algorithm for expressions

$$\text{monomorphic typing contexts } \Xi^- ::= \epsilon \mid \Xi^-, x : A^-$$
$$\text{polymorphic typing contexts } \Pi ::= \epsilon \mid \Pi, \hat{\mathbf{x}} : [\Xi^-]A^+$$

Computations

$$\textsc{App}\ \frac{\Pi \triangleright v_1 : [\Xi_1^-]A_1^+ \qquad \Pi \triangleright v_2 : [\Xi_2^-]A_2^+}{\Pi \triangleright v_1\ v_2 : \xi([\Xi_1^- \sqcap \Xi_2^-](\alpha\ !\ \delta))}\xi = biunify(A_1^+ \leqslant A_2^+ \to (\alpha\ !\ \delta))$$

$$\textsc{Cond}\ \frac{\Pi \triangleright v : [\Xi_1^-]A^+ \qquad \Pi \triangleright c_1 : [\Xi_2^-]\underline{C}_1^+ \qquad \Pi \triangleright c_2 : [\Xi_3^-]\underline{C}_2^+}{\Pi \triangleright \texttt{if}\ v\ \texttt{then}\ c_1\ \texttt{else}\ c_2 : \xi([\Xi_1^- \sqcap \Xi_2^- \sqcap \Xi_3^-](\alpha\ !\ \delta))}\xi =$$

$$biunify\begin{pmatrix} A^+ \leqslant bool \\ \underline{C}_1^+ \leqslant (\alpha\ !\ \delta) \\ \underline{C}_2^+ \leqslant (\alpha\ !\ \delta) \end{pmatrix}$$

$$\textsc{Ret}\ \frac{\Pi \triangleright v : [\Xi^-]A^+}{\Pi \triangleright \texttt{return}\ v : [\Xi^-](A^+\ !\ \emptyset)}$$

$$\textsc{Op}\ \frac{(\texttt{Op} : A^+ \to B^-) \in \Sigma \qquad \Pi \triangleright v : [\Xi^-]A^+}{\Pi \triangleright \texttt{Op}\ v : [\Xi^-](B^+\ !\ \texttt{Op})}$$

$$\textsc{Let}\ \frac{\Pi \triangleright v : [\Xi_1^-]A^+ \qquad \Pi, \hat{\mathbf{x}} : [\Xi_1^-]A^+ \triangleright c : [\Xi_2^-](B^+\ !\ \Delta^+)}{\Gamma \triangleright \texttt{let}\ \hat{\mathbf{x}} = v\ \texttt{in}\ c : [\Xi_1^- \sqcap \Xi_2^-](B^+\ !\ \Delta^+)}$$

$$\textsc{Do}\ \frac{\Pi \triangleright c_1 : [\Xi_1^-](A^+\ !\ \Delta_1^+) \qquad \Pi, \hat{\mathbf{x}} : [\Xi_1^-]A^+ \triangleright c_2 : [\Xi_2^-](B^+\ !\ \Delta_2^+)}{\Gamma \triangleright \texttt{do}\ \hat{\mathbf{x}} = c_1\ ;\ c_2 : [\Xi_1^- \sqcap \Xi_2^-](B^+\ !\ (\Delta_1^+ \sqcup \Delta_2^+))}$$

$$\textsc{With}\ \frac{\Pi \triangleright v : [\Xi_1^-]\underline{C}_1^+ \qquad \Pi \triangleright c : [\Xi_2^-]\underline{C}_2^+}{\Pi \triangleright \texttt{handle}\ c\ \texttt{with}\ v : \xi([\Xi_1^- \sqcap \Xi_2^-](\alpha\ !\ \delta))}\xi = biunify(\underline{C}_1^+ \leqslant \underline{C}_2^+ \Rightarrow (\alpha\ !\ \delta))$$

Figure 4.8: Type inference algorithm for computations

Chapter 5

Simplification

intro simplification

5.1 Type Automata

intro type automata

5.2 Encoding

explain encoding

5.3 Decoding

explain decoding

Type Automata

$$\Sigma_F ::= \{d, r, dh, rh, t, e\} \qquad T ::= \{\langle\rightarrow\rangle, \langle\Rightarrow\rangle, \langle \mathsf{b}\rangle, \langle \mathsf{drty}\rangle\}$$
$$\Sigma ::= V^+ \cup V^- \cup \Sigma_F^+ \cup \Sigma_F^- \cup T^+ \cup T^- \qquad E ::= \emptyset \mid \epsilon \mid \Sigma \mid E + E \mid E.E$$

$$\mathcal{E}^+(\langle\rightarrow\rangle) = \langle\rightarrow\rangle^+ \qquad \mathcal{E}^+(\langle\Rightarrow\rangle) = \langle\Rightarrow\rangle^+ \qquad \mathcal{E}^+(\langle \mathsf{b}\rangle) = \langle \mathsf{b}\rangle^+$$
$$\mathcal{E}^+(\langle \mathsf{drty}\rangle) = \langle \mathsf{drty}\rangle^+ \qquad \mathcal{E}^-(\langle\rightarrow\rangle) = \langle\rightarrow\rangle^- \qquad \mathcal{E}^-(\langle\Rightarrow\rangle) = \langle\Rightarrow\rangle^-$$
$$\mathcal{E}^-(\langle \mathsf{b}\rangle) = \langle \mathsf{b}\rangle^- \qquad \mathcal{E}^-(\langle \mathsf{drty}\rangle) = \langle \mathsf{drty}\rangle^-$$

$$\mathcal{E}^+(\underline{C}) = \mathcal{E}^+(A \mathbin{!} \Delta) = t^+.\mathcal{E}^+(A) + e^+.\mathcal{E}^+(\Delta) + \mathcal{E}^+(\langle \mathsf{drty}\rangle)$$
$$\mathcal{E}^-(\underline{C}) = \mathcal{E}^-(A \mathbin{!} \Delta) = t^-.\mathcal{E}^-(A) + e^-.\mathcal{E}^-(\Delta) + \mathcal{E}^-(\langle \mathsf{drty}\rangle)$$

$$\mathcal{E}^+(\alpha) = \alpha^+ \qquad \mathcal{E}^-(\alpha) = \alpha^-$$

$$\mathcal{E}^+(A_1 \sqcup A_2) = \mathcal{E}^+(A_1) + \mathcal{E}^+(A_2) \qquad \mathcal{E}^-(A_1 \sqcap A_2) = \mathcal{E}^-(A_1) + \mathcal{E}^-(A_2)$$

$$\mathcal{E}^+(\bot) = \emptyset \qquad \mathcal{E}^-(\top) = \emptyset$$

$$\mathcal{E}^+(A \rightarrow \underline{C}) = d^+.\mathcal{E}^-(A) + r^+.\mathcal{E}^+(\underline{C}) + \mathcal{E}^+(\langle\rightarrow\rangle)$$
$$\mathcal{E}^-(A \rightarrow \underline{C}) = d^-.\mathcal{E}^+(A) + r^-.\mathcal{E}^-(\underline{C}) + \mathcal{E}^-(\langle\rightarrow\rangle)$$

$$\mathcal{E}^+(\underline{C} \Rightarrow \underline{D}) = dh^+.\mathcal{E}^-(\underline{C}) + rh^+.\mathcal{E}^+(\underline{D}) + \mathcal{E}^+(\langle\Rightarrow\rangle)$$
$$\mathcal{E}^-(\underline{C} \Rightarrow \underline{D}) = dh^-.\mathcal{E}^+(\underline{C}) + rh^-.\mathcal{E}^-(\underline{D}) + \mathcal{E}^-(\langle\Rightarrow\rangle)$$

$$\mathcal{E}^+(bool) = \mathcal{E}^+(\langle \mathsf{b}\rangle) \qquad \mathcal{E}^-(bool) = \mathcal{E}^-(\langle \mathsf{b}\rangle)$$

$$\mathcal{E}^+(\mu\alpha.A) = \Omega_\alpha^+(A)^*.\mathcal{E}^+([\bot/\alpha]A) \qquad \mathcal{E}^-(\mu\alpha.A) = \Omega_\alpha^-(A)^*.\mathcal{E}^-([\top/\alpha]A)$$

$$\mathcal{E}^+(\mathtt{0p}) = \langle \mathtt{0p}\rangle^+ \qquad \mathcal{E}^-(\mathtt{0p}) = \langle \mathtt{0p}\rangle^-$$

$$\mathcal{E}^+(\delta) = \delta^+ \qquad \mathcal{E}^-(\delta) = \delta^-$$

$$\mathcal{E}^+(\Delta_1 \sqcup \Delta_2) = \mathcal{E}^+(\Delta_1) + \mathcal{E}^+(\Delta_2) \qquad \mathcal{E}^-(\Delta_1 \sqcap \Delta_2) = \mathcal{E}^-(\Delta_1) + \mathcal{E}^-(\Delta_2)$$

Figure 5.1: Encoding types as type automata

Type Automata

$$\Sigma_F ::= \{d, r, dh, rh, t, e\} \qquad T ::= \{\langle\rightarrow\rangle, \langle\Rightarrow\rangle, \langle \mathsf{b} \rangle, \langle \mathsf{drty} \rangle\}$$
$$\Sigma ::= V^+ \cup V^- \cup \Sigma_F^+ \cup \Sigma_F^- \cup T^+ \cup T^- \qquad E ::= \emptyset \mid \epsilon \mid \Sigma \mid E + E \mid E.E$$

$$\Omega_\alpha^+(\underline{C}) = \Omega_\alpha^+(A \mathbin{!} \Delta) = t^+.\Omega_\alpha^+(A) + e^+.\Omega_\alpha^+(\Delta) \qquad \Omega_\alpha^-(\underline{C}) = \Omega_\alpha^-(A \mathbin{!} \Delta) = t^-.\Omega_\alpha^-(A) + e^-.\Omega_\alpha^-(\Delta)$$

$$\Omega_\alpha^+(\alpha) = \epsilon \qquad \Omega_\alpha^-(\alpha) = \epsilon$$

$$\Omega_\alpha^+(\beta) = \emptyset \qquad \Omega_\alpha^-(\beta) = \emptyset$$

$$\Omega_\alpha^+(A_1 \sqcup A_2) = \Omega_\alpha^+(A_1) + \Omega_\alpha^+(A_2) \qquad \Omega_\alpha^-(A_1 \sqcup A_2) = \Omega_\alpha^-(A_1) + \Omega_\alpha^-(A_2)$$

$$\Omega_\alpha^+(\bot) = \emptyset \qquad \Omega_\alpha^-(\top) = \emptyset$$

$$\Omega_\alpha^+(A \rightarrow \underline{C}) = d^+.\Omega_\alpha^-(A) + r^+.\Omega_\alpha^+(\underline{C}) \qquad \Omega_\alpha^-(A \rightarrow \underline{C}) = d^-.\Omega_\alpha^+(A) + r^-.\Omega_\alpha^-(\underline{C})$$

$$\Omega_\alpha^+(\underline{C} \Rightarrow \underline{D}) = dh^+.\Omega_\alpha^-(\underline{C}) + rh^+.\Omega_\alpha^+(\underline{D}) \qquad \Omega_\alpha^-(\underline{C} \Rightarrow \underline{D}) = dh^-.\Omega_\alpha^+(\underline{C}) + rh^-.\Omega_\alpha^-(\underline{D})$$

$$\Omega_\alpha^+(bool) = \emptyset \qquad \Omega_\alpha^-(bool) = \emptyset$$

$$\Omega_\alpha^+(\mathsf{0p}) = \emptyset \qquad \Omega_\alpha^-(\mathsf{0p}) = \emptyset$$

$$\Omega_\alpha^+(\delta) = \emptyset \qquad \Omega_\alpha^-(\delta) = \emptyset$$

$$\Omega_\alpha^+(\Delta_1 \sqcup \Delta_2) = \Omega_\alpha^+(\Delta_1) + \Omega_\alpha^+(\Delta_2) \qquad \Omega_\alpha^-(\Delta_1 \sqcup \Delta_2) = \Omega_\alpha^-(\Delta_1) + \Omega_\alpha^-(\Delta_2)$$

$$\Omega_\alpha^+(\mu\alpha.A) = \emptyset \qquad \Omega_\alpha^-(\mu\alpha.A) = \emptyset$$

$$\Omega_\alpha^-(\mu\beta.A) = \Omega_\beta^+(A)^*.\Omega_\alpha^+([\bot/\beta]A) \qquad \Omega_\alpha^-(\mu\beta.A) = \Omega_\beta^-(A)^*.\Omega_\alpha^-([\top/\beta]A)$$

Figure 5.2: Encoding recursive types as type automata

$$(A_1^+, A_1^-) + (A_2^+, A_2^-) = (A_1^+ \sqcup A_2^+, A_1^- \sqcap A_2^-)$$

$$(\underline{C}_1^+, \underline{C}_1^-) + (\underline{C}_2^+, \underline{C}_2^-) = (\underline{C}_1^+ \sqcup \underline{C}_2^+, \underline{C}_1^- \sqcap \underline{C}_2^-)$$

$$(A_1^+, A_1^-).(A_2^+, A_2^-) = \left\{ \begin{array}{cc} (\bot, \top) & \text{if } (A_2^+, A_2^-) = (\bot, \top) \\ \left(\begin{array}{c} [A_2^+/\mathcal{X}^+, A_2^-/\mathcal{X}^-]A_1^+, \\ [A_2^+/\mathcal{X}^+, A_2^-/\mathcal{X}^-]A_1^- \end{array} \right) & \text{otherwise} \end{array} \right\}$$

$$(\underline{C}_1^+, \underline{C}_1^-).(\underline{C}_2^+, \underline{C}_2^-) =$$

$$\left\{ \begin{array}{cc} (\bot \,!\, \emptyset, \top \,!\, \Omega) & \text{if } (\underline{C}_2^+, \underline{C}_2^-) = (\bot \,!\, \emptyset, \top \,!\, \Omega) \\ \left(\begin{array}{c} [\underline{C}_2^+/\mathcal{X}^+ \,!\, \mathcal{D}^+, \underline{C}_2^-/\mathcal{X}^- \,!\, \mathcal{D}^-]\underline{C}_1^+, \\ [\underline{C}_2^+/\mathcal{X}^+ \,!\, \mathcal{D}^+, \underline{C}_2^-/\mathcal{X}^- \,!\, \mathcal{D}^-]\underline{C}_1^- \end{array} \right) & \text{otherwise} \end{array} \right\}$$

$$(\underline{C}_1^+, \underline{C}_1^-).(A_2^+, A_2^-) =$$

$$\left\{ \begin{array}{cc} (\bot \,!\, \emptyset, \top \,!\, \Omega) & \text{if } (A_2^+, A_2^-) = (\bot, \top) \\ \left(\begin{array}{c} [A_2^+/\mathcal{X}^+ \,!\, \mathcal{D}^+, A_2^-/\mathcal{X}^- \,!\, \mathcal{D}^-]\underline{C}_1^+, \\ [A_2^+/\mathcal{X}^+ \,!\, \mathcal{D}^+, A_2^-/\mathcal{X}^- \,!\, \mathcal{D}^-]\underline{C}_1^- \end{array} \right) & \text{otherwise} \end{array} \right\}$$

$$\mathcal{R}(\emptyset) = (\bot, \top) \qquad \mathcal{R}(\epsilon) = (\mathcal{X}, \mathcal{X})$$

Figure 5.3: Decoding type automata concat, union and kleene star into types

$$\mathcal{R}(\alpha^+) = (\alpha, \top) \qquad \mathcal{R}(\alpha^-) = (\bot, \alpha)$$

$$\mathcal{R}(\langle\rightarrow\rangle^+) = (\top \rightarrow \bot, \top) \qquad \mathcal{R}(\langle\rightarrow\rangle^-) = (\bot, \top \rightarrow \bot)$$

$$\mathcal{R}(d^+) = (\mathcal{X} \rightarrow \bot, \top) \qquad \mathcal{R}(d^-) = (\bot, \mathcal{X} \rightarrow \bot)$$

$$\mathcal{R}(r^+) = (\top \rightarrow \mathcal{X}, \top) \qquad \mathcal{R}(r^-) = (\bot, \top \rightarrow \mathcal{X})$$

$$\mathcal{R}(dh^+) = (\mathcal{X} \mathbin{!} \mathcal{D} \Rightarrow \bot, \top) \qquad \mathcal{R}(dh^-) = (\bot, \mathcal{X} \Rightarrow \bot)$$

$$\mathcal{R}(rh^+) = (\top \Rightarrow \mathcal{X} \mathbin{!} \mathcal{D}, \top) \qquad \mathcal{R}(rh^-) = (\bot, \top \Rightarrow \mathcal{X})$$

$$\mathcal{R}(\langle\mathsf{b}\rangle^+) = (\mathit{bool}, \top) \qquad \mathcal{R}(\langle\mathsf{b}\rangle^-) = (\bot, \mathit{bool})$$

$$\mathcal{R}(\langle\mathsf{drty}\rangle^+) = (\bot \mathbin{!} \emptyset, \top) \qquad \mathcal{R}(\langle\mathsf{drty}\rangle^-) = (\bot, \bot \mathbin{!} \emptyset)$$

$$\mathcal{R}(\langle\mathtt{Op}\rangle^+) = (\bot \mathbin{!} \mathtt{Op}, \top \mathbin{!} \Omega) \qquad \mathcal{R}(\langle\mathtt{Op}\rangle^-) = (\bot, \bot \mathbin{!} \mathtt{Op})$$

$$\mathcal{R}(t^+) = (\mathcal{X} \mathbin{!} \emptyset, \top \mathbin{!} \Omega) \qquad \mathcal{R}(t^-) = (\bot, \mathcal{X} \mathbin{!} \emptyset)$$

$$\mathcal{R}(e^+) = (\bot \mathbin{!} \mathcal{D}, \top \mathbin{!} \Omega) \qquad \mathcal{R}(e^-) = (\bot, \bot \mathbin{!} \mathcal{D})$$

Figure 5.4: Decoding type automata into types

Chapter 6

Implementation & Evaluation

In order to have an emperical evaluation of the proposed type system, it has been implemented in Eff, a prototype functional programming language with algebraic effects and handlers. This chapter provides the necessary information required to implement the type system. Eff is an open-source system [23].

The scope of Eff is about 6000 lines of code with 1500 lines of comments. A lot of this code was already written, including, a lexer, parser, optimization engine, runtime and utilities. The code that implements the terms, types and type inference spans 2700 lines of code with 750 lines of comments. This shows that half of the entire codebase of Eff consists out of the type inference engine, which is huge in comparison to the other different components.

First, some general background information is given. More specifically, the pre-existing structure of Eff is brievely touched upon such that it is clear how to piece all the parts together. Afterwards, the structure of the types and terms are discussed.

The section for type inference consists out of two important parts. It contains information about the reformulated typing rules and the polarity of types. Secondly it contains information about the actual type inference and biunification algorithms. Finally, the simplification algorithm is discussed.

6.1 Overview

Eff is a prototype functional programming language implemented in OCaml. The type system as discussed in this thesis requires several additional components. Considering that this type system was implemented within Eff, these components already existed.

The lexer and parser are necessary components for any programming language. For Eff, these were implemented using `ocamllex` and `ocamlyacc`. The lexer and parser produces sugared untyped terms. In case of Eff, the sugared syntax contains both functions and lambda's. The sugared syntax is immediately desugared into an untyped

Figure 6.1: Term implementation

```
let annotate t sch loc = {
    term = t;
    scheme = sch;
    location = loc;
}

type expression = (plain_expression, Scheme.ty_scheme)
  ↪ annotation
and computation = (plain_computation, Scheme.dirty_scheme
  ↪ ) annotation
```

language containing only the necessary, basic components. The sugared syntax and the untyped language both contain only a single syntactic sort of terms, which groups together expressions and computations.

Once this process is completed, the untyped syntax is then converted into a typed language using the type system described in this thesis. This aspect is discussed in depth in later sections. The typed language can then be used for optimizations and outputting OCaml code. When outputting OCaml code, a free monad representation is used. [2, 24]

Eff provides an interpreted runtime environment. This runtime environment is based upon the untyped syntax. Having the runtime environment based upon the untyped syntax makes the interpreter more robust. As mentioned before, Eff is a prototype language and is therefor in a constant state of change. The biggest changes happen within the type system (or some syntactical changes). The interpreter has no specific need for a typed language as it is merely a tool to calculate the result of a program.

6.2 Types and terms

The terms and types of EffCore have a near identical representation as seen in the specification. There are two sorts of terms, expressions and computations as seen in Listing 6.1. Each term is annotated with a location and a scheme. The location refers to the location within the source code in order to be able to use that information when printing debug or error messages.

A scheme contains information about the type, the free variables that occur in this type called the context and a type used for subtyping constraints as seen in Listing 6.2. Types are represented in an identical representation as seen in the specification. Eff contains some additional types such as tuples and records. Like in the specification, there are two sorts of types, pure types and dirty types.

Figure 6.2: Type implementation

```
1 (* represents a context and contains all free variables
     ↪ that occur *)
2 type context = (Untyped.variable, Type.ty) OldUtils.assoc
3 (* represents a generic scheme *)
4 type 'a t = context * 'a * Unification.t
5 (* type scheme for expressions *)
6 type ty_scheme = Type.ty t
7 (* type scheme for computations *)
8 type dirty_scheme = Type.dirty t
```

Construction of typed terms happens through the use of "smart" constructors. This was introduced within the previous type system of Eff and, as it is a useful feature, has also been used for the implementation of this thesis. "Smart" constructors take already types subterms as arguments and contains the necessary logic to properly construct the annotated term.

For example, the "smart" constructor for the application is given in Listing 6.3. It calculates the location, constructs the term and uses another constructor for the creation of the required scheme. Within the "smart" constructor for the scheme, it can be seen that a constraint $ty\_e1 \leqslant ty\_e2 \rightarrow drty$ is made. $drty$ contains a fresh type variable and a fresh dirt variable. The type inference algorithm traverses the untyped terms and applies the corresponding smart constructors in order to construct the typed terms.

Additionally, Eff contains pattern terms which are used within the context of abstractions. For example, a pattern is used for `x` within `let x = e in c`. The usage of these patterns are an implementation choice that was made in the pre-existing code of Eff.

6.3 Type Inference

The type inference algorithm implementation traverses, as stated before, the untyped terms and applies the corresponding smart constructors in order to construct the typed terms. Each "smart" constructor contains the logic necessary to construct the terms. They make the fresh type and dirt variables, add the subtyping constraints and alter the context. The type inference algorithm shown in figure 4.7 and figure 4.8 are directly implemented using these "smart" constructors. Just like in the specification, the biunification happens at the end of each "smart" constructor (if applicable).

More interesting is the implementation of the biunification algorithm and polar types. From the aspect of the implementation, polar types are only useful for the biunification algorithm. Thus, most of the implementation completely ignores polar types. Polar types are implemented using a boolean flag. With bisubstitution, a type is traversed. If the polarity of the outer type is known, all other polarities can be deduced from this given

Figure 6.3: Smart constructor of application

```
let apply ?loc e1 e2 =
    let loc = backup_location loc [e1.Typed.location; e2.
      ↪ Typed.location] in
    let term = Typed.Apply (e1, e2) in
    let sch = Scheme.apply ~loc e1.Typed.scheme e2.Typed.
      ↪ scheme in
    Typed.annotate term sch loc

let Scheme.apply ~loc e1 e2 =
    let ctx_e1, ty_e1, cnstrs_e1 = e1 in
    let ctx_e2, ty_e2, cnstrs_e2 = e2 in
    let drty = Type.fresh_dirty () in
    let constraints = Unification.union cnstrs_e1
      ↪ cnstrs_e2  in
    let constraints = Unification.add_ty_constraint ~loc
      ↪ ty_e1 (Type.Arrow (ty_e2, drty)) constraints in
    solve_dirty (ctx_e1 @ ctx_e2, drty, constraints)
```

polarity. Thus, there is no need to completely convert a type into a polar type from the perspective of the implementation.

Another interesting aspect, which has not yet been discussed for the implementation, are the reformulated typing rules. The type inference algorithm shown in figure 4.7 and figure 4.8 are based on the reformulated typing rules. The main differences between the normal typing rules and the reformulated typing rules is that the reformulated typing rules distinguish between lambda-bound and let-bound variables.

The sugared syntax and the untyped syntax of Eff does not make this distinction. Thus the implementation has to make this distinction whenever it encounters a variable. Listing 6.4 shows part of the type inference algorithm (and an additional required function). The type inference algorithm implementation is context-sensitive. This means that there is an explicit context argument `ctx` that needs to be passed around. When an untyped variable is encountered, a simple lookup can be done within this context. This context argument is altered whenever a lambda is encountered.

The implementation of `Ctor.lambdavar` and `Ctor.letvar` is trivial. When the distinction is made, and it is found to be a let-bound variable, `get_var_scheme_env` is used to find the scheme of the let-bound variable. There is an additional state argument `st` that contains the polymorphic context. If the variable cannot be found in the polymorphic context, it means that we found this variable before it was bound (or that it is an unbound variable). A temporary scheme is created and the situation is resolved in a later stage, which might end up in a "unbound variable" error.

Figure 6.4: Distinguish lambda-bound and let-bound variables

```
1  (* Lookup the variable in the context, which includes
       ↪ only
2  lambda-bound variables.
3  If we can find the variable, it is lambda-bound.
4  Otherwise, check the (polymorphic) context
5  *)
6  begin match OldUtils.lookup x ctx with
7  | Some ty -> Ctor.lambdavar ~loc x ty, st
8  | None -> Ctor.letvar ~loc x (get_var_scheme_env ~loc st
       ↪ x), st
9  end
10
11 (* Lookup a type scheme for a variable in the typing
       ↪ environment
12 Otherwise, create a new scheme (and add it to the typing
       ↪ environment)
13 *)
14 let get_var_scheme_env ~loc st x =
15 begin match TypingEnv.lookup st.context x with
16     | Some ty_sch -> ty_sch
17     | None ->
18             let ty = Type.fresh_ty () in
19             let sch = Scheme.tmpvar ~loc x ty in
20             sch
21 end
```

6.4 Simplification

The simplification algorithm is partly left for future work. The simplification algortihm works in four stages. In the first stage, types are converted into type automata which are nondeterministic finite automata (NFA). These type automata need to be simplified which can be done using any standard simplification algortihm. However, most of these algorithms operate on deterministic finite automata (DFA) as opposed to NFA's. Thus in the second stage, the type automata is converted into a DFA. Afterwards, in the third stage, the DFA is simplified using any chosen simplification algorithm. Finally, the DFA is deconstructed into a type again.

The implementation for the second and third stage are left for future work. However, the first and last stage, the encoding and decoding have been implemented. In the definition of the automaton as seen in Listing 6.5, an initial and final state is required. The transition table is stored as a list. While transitions representing types and dirts could be stored within the same list, in order to simplify the encoding, there are two

Figure 6.5: Representation of type automata

```
type ('state,'letter) automaton = {
    initial    : 'state;
    final      : 'state;
    transition : ('state * 'letter * 'state list) list;
    transition_drt : ('state * 'letter * 'state list)
      ↪ list;
    currentState : 'state;
    prevtransition : ('state * 'letter * 'state list)
      ↪ list;
}

type alphabet =
    | Prim of (Type.prim_ty * bool)
    | Function of bool
    | Handler of bool
    | Alpha of (Params.ty_param * bool)
    | Domain of bool
    | Range of bool
    | Op of (OldUtils.effect * bool)
    | DirtVar of (Params.dirt_param * bool)

type statetype = int

type automatype = (statetype, alphabet) automaton
```

different lists. Just like there are two different data types for types and dirty types. The alphabet is a direct implementation from the specification. The representation of the actual states is of no importance. In the implementation, integers were chosen as we can very simply generate new unique states by incrementing a counter. The simplification should take place right after the biunification within each "smart" constructor.

6.5 Evaluation

We evaluate the implemented type inference engine on a number of benchmarks. First, the performance of EffCore is compared against the subtyping based system. Subtyping collects all constraints throughout the type inference, while EffCore solves each constraint immediately. I made the hypothesis that EffCore should be faster than subtyping.

Secondly, EffCore represents types in a smaller and cleaner format compared to subtyping (due to not having any explicit constraints anymore). Several programs are tested in both systems in order to manually compare them.

All benchmarks were run on a MacBook Pro with an 2.5 GHz Intel Core I7 processor and 16 GB 1600 MHz DDR3 RAM running Mac OS 10.13.4.

6.5.1 Performance Comparison

Our first evaluation, in figure 2.17, considers five different testing programs:

explanation interp, loop, parser, queens, range

1. `Interp`,
2. `Loop`,
3. `Parser`,
4. `Queens`,
5. `Range`,

These programs are compiled with three different type systems:

1. Subtyping type system, the "old" type system from the Eff programming language.
2. EffCore, the system proposed in this thesis, based upon extending algebraic subtyping with algebraic effects and handlers.
3. Untyped system, the Eff programming language without type inference.

The different systems are all implemented in the Eff programming language. Thus, they share many different aspects of the implementation, including parsing, lexing and the runtime. The Untyped system does not compute any types, while the other two systems do use type inference engines.

Figure 6.6 shows the time relative to the Untyped version for running each of the programs for 10,000 iterations. Overall, the performance of EffCore is superior to the performance of standard subtyping. There is an exception for the `Queens` program. The implementation of EffCore is very slow compared to the other implementations. This is a very strange anomaly as it was completely unexpected. I hypothesize that the lack of a simplification algorithm causes the huge slowdown. Because the simplification of the types never occurs, the types keep growing in size. Due to this, any constraints that are introduced during type inference are also very large in size, which causes the slowdown.

give and explain the results

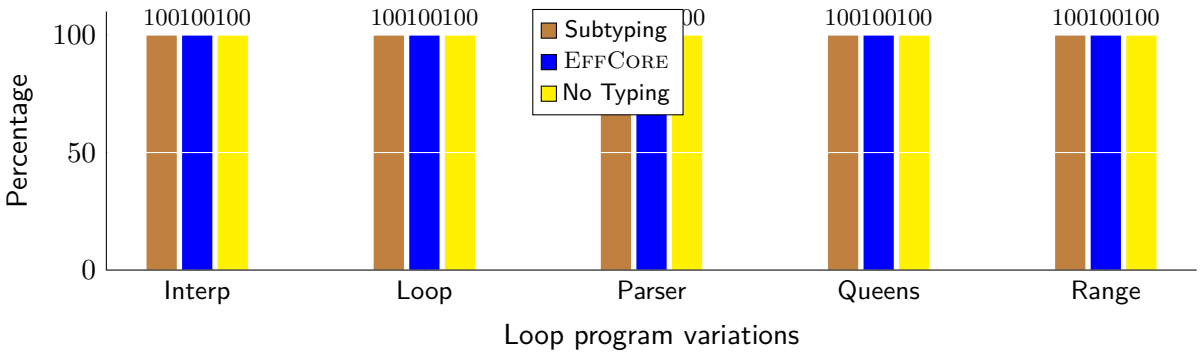

Figure 6.6: Relative run-times of testing programs

6.5.2 Type Inference Comparison

The main contribution of this thesis is to extend the algebraic subtyping algorithm with algebraic effects and handlers. The goal is to be able to infer types which are shorter, simpler and more human-readable than types inferred by standard subtyping systems. In this evaluation, several small programs were made and its types inferred by the subtyping and the algebraic subtyping system in order to compare both types.

Considering that the implementation does not yet support the simplification of types, the results are inaccurate. In order to provide a proper evaluation, the types have been manually simplified using the simplification algorithm. All three types are given in order to the current state of the implementation and to be able to evaluate the algebraic subtyping system.

give the evaluation results

Interp	Loop	Parser	Queens
$\Pi \triangleright x : [x : \alpha]\alpha$	$\Pi \triangleright x : [x : \alpha]\alpha$	$\Pi \triangleright x : [x : \alpha]\alpha$	$\Pi \triangleright x : [x : \alpha]\alpha$

Range
$\Pi \triangleright x : [x : \alpha]\alpha$

Figure 6.7: Produced types for Subtyping

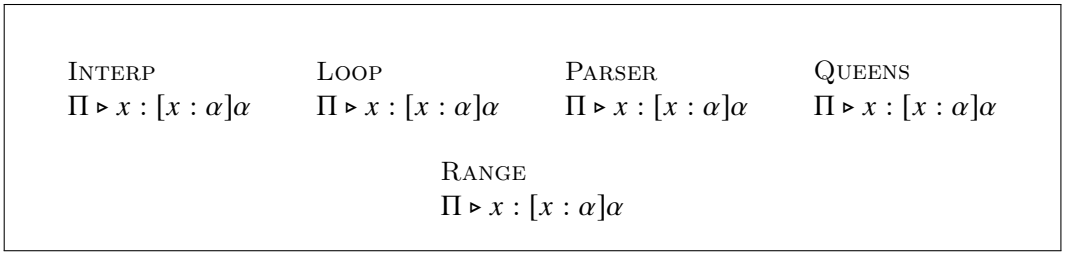

Figure 6.8: Produced types for EffCore

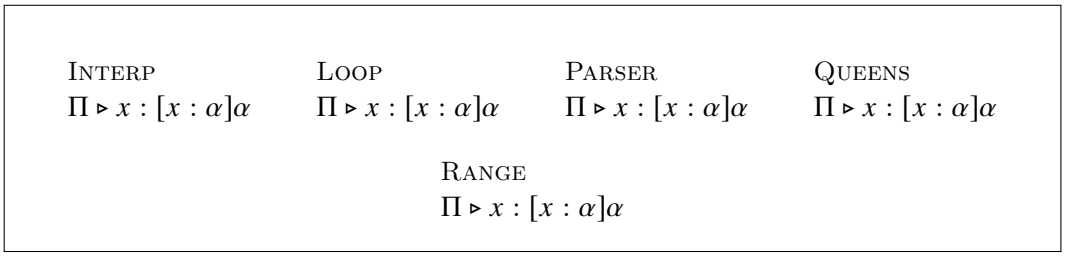

Figure 6.9: Produced types for Algebraic Subtyping

Chapter 7

Related Work

related work (hypothesis dolan effect system + explicit effect + row-based)

7.1 Algebraic Subtyping

7.2 Explicit Effect Subtyping

7.3 Row-Based Effect Typing

Chapter 8

Conclusion

8.1 Future Work

write future work

8.1.1 Simplification

8.1.2 Biunification with type automata

8.1.3 Optimization

8.2 Conclusion

write conclusion

Briefly recall what the goal of the work was. Summarize what you have done, summarize the results, and present conclusions. Conclusions include a critical assessment: where the original goals reached? Discuss the limitations of your work. Describe how the work could possibly be extended in the future, mitigating limitations or solving remaining problems.

Appendices

Appendix A

Proofs

A.1 Instantiation

proof instantiation

A.2 Weakening

proof weakening

A.3 Substitution

proof substitution

A.4 Soundness

proof soundness

A.5 Equivalence of Original and Reformulated Typing Rules

proof equivalence

A.6 Correctness of Biunification

proof correctness

A.7 Principality of Type Inference

proof principality

A.8 Simplifying Type Automata

proof type automata

Appendix B

Poster

Algebraic Subtyping for Algebraic Effects and Handlers

Axel Faes promotor: Tom Schrijvers advisor: Amr Hany Saleh

Introduction

Algebraic effect handlers
A feature for side effects and exception handlers on steroids [1, 4, 5]

Implemented in Eff programming language [6, 7]

Algebraic subtyping
A form of subtyping used in type inference systems [2]

```
let twice f x =
    f (f x)
```

With subtyping
twice : (β -> γ) -> α -> γ | α ≤ β, γ ≤ β

With algebraic subtyping
twice : (α -> α & β) -> α -> β

```
effect Op : unit -> int

let someFun b =
    handle (
        if (b == 0) then
            let a = #Op ()
            print a
        else
            print b
    ) with
        | #Op () cont -> cont 1
```

Research problem

Too many constraints
Constraints keep stacking and they become unwieldy.

Algebraic Subtyping
Does not use effect information.

Optimisations
Implementing effect-aware optimizing compiler is error-prone due to the many different constraints that remain unsolved in traditional subtyping systems.

How can effects be introduced within algebraic subtyping? How can such a system be implemented in the Eff programming language?

Extension

value v	::=	x	λ-variable
	|	$\hat{x}$	let-variable
	|	`true`	true
	|	`false`	false
	|	$\lambda x.c$	function
	|	{	**handler**
		`return` $x \mapsto c_r$,	**return case**
		$[\mathtt{Op}\ y\ k \mapsto c_{\mathtt{Op}}]_{\mathtt{Op} \in O}$	**operation cases**
		}	
comp c	::=	$v_1\ v_2$	application
	|	`do` $\hat{x} = c_1$; c_2	sequencing
	|	`let` $\hat{x} = v$ `in` c	let
	|	`if` e `then` c_1 `else` c_2	conditional
	|	`return` v	**returned val**
	|	`Op` v	**operation call**
	|	`handle` c `with` v	**handling**

(pure) type A, B	::=	`bool`	bool type
	|	$A \to \underline{C}$	function type
	|	$\underline{C} \Rightarrow \underline{D}$	**handler type**
	|	α	type variable
	|	$\mu\alpha.A$	recursive type
	|	$\top$	top
	|	$\bot$	bottom
	|	$A \sqcap B$	intersection
	|	$A \sqcup B$	union
dirty type $\underline{C}, \underline{D}$	::=	$A\ !\ \Delta$	
dirt Δ	::=	`Op`	operation
	|	δ	dirt variable
	|	$\emptyset$	empty dirt
	|	$\Delta_1 \sqcap \Delta_2$	intersection
	|	$\Delta_1 \sqcup \Delta_2$	union

$$(A_1 \to \underline{C}_1) \sqcup (A_2 \to \underline{C}_2) \equiv (A_1 \sqcap A_2) \to (\underline{C}_1 \sqcup \underline{C}_2)$$

Formulation of typing rules

Relationship to subtyping

Biunification algorithm
input <-> output

Type inference algorithm

Type simplification algorithm
Construct type automata
Convert to NFA
Minimization algorithm

Semantics of the dirt
Set operations?
(Op ⊔ Op2) ⊓ (Op ⊔ Op3)
=> Op

Input vs Output
⊔ for outputs
⊓ for inputs

$$\Delta_1 \leqslant \Delta_2 \leftrightarrow \Delta_1 \sqcup \Delta_2 \equiv \Delta_2$$

$$\Delta_1 \leqslant \Delta_2 \leftrightarrow \Delta_1 \equiv \Delta_1 \sqcap \Delta_2$$

Evaluation

Implementation
Eff programming language
Fully featured

Replace type inference engine
~2900 loc ⇔ ~5800

Proofs
Algorithms have been proven to be correct (see thesis appendix for the proofs)

System / Program	Algebraic Subtyping	Standard Subtyping
Interpreter	**1.550**	1.740
Loop	**1.285**	2.859
Parser	171.887	**11.791**
N-Queens	35.377	**5.362**
Range	**0.642**	1.058

Future Work

Optimizations engine
Adapting the effect-aware optimizing compiler for the algebraic-subtyping based system.

Simplification of types and effects
Implementing the extended algorithm to simplify types and effects using type automata.

Summary

Algebraic effects and handlers
These are a very active area of research. An important aspect is the development of an optimising compiler.

Research problem
Compilation is a slow process with difficult to read types due to the constraints without a strong, reliable type-&-effect system.

This thesis simplifies constraint generation for types ***AND EFFECTS*** by providing a new core language based upon extended **algebraic subtyping.**

References

[1] Andrej Bauer and Matija Pretnar. 2014. An Effect System for Algebraic Effects and Handlers. Logical Methods in Computer Science 10, 4 (2014). https://doi.org/10.2168/LMCS-10(4:9)2014

[2] Stephen Dolan and Alan Mycroft. 2017. Polymorphism, Subtyping, and Type Inference in MLsub. In Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017). ACM, New York, NY, USA, 60–72. https://doi.org/10.1145/3009837.3009882

[3] Sam Lindley, Conor McBride, and Craig McLaughlin. 2017. Do Be Do Be Do. In Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017). ACM, New York, NY, USA, 500–514. https://doi.org/10.1145/3009837.3009897

[4] Gordon D. Plotkin and Matija Pretnar. 2013. Handling Algebraic Effects. Logical Methods in Computer Science 9, 4 (2013). https://doi.org/10.2168/LMCS-9(4:23)2013

[5] Matija Pretnar. 2014. Inferring Algebraic Effects. Logical Methods in Computer Science 10, 3 (2014). https://doi.org/10. 2168/LMCS-10(3:21)2014

[6] Matija Pretnar. 2015. An introduction to algebraic effects and handlers. invited tutorial paper. Electronic Notes in Theoretical Computer Science 319 (2015), 19–35.

[7] Andrej Bauer and Matija Pretnar. 2015. Programming with algebraic effects and handlers. J. Log. Algebr. Meth. Program. 84, 1 (2015), 108–123. https://doi.org/10.1016/j.jlamp.2014.02.001

Acknowledgements

I would like to thank Amr Hany Saleh for his continuous guidance and help. I would also like to thank Matija Pretnar and Tom Schrijvers for their support during my research.

KU LEUVEN

DTAI
Declaratieve Talen en Artificiële Intelligentie

Bibliography

[1] A. Bauer and M. Pretnar. An effect system for algebraic effects and handlers. *Logical Methods in Computer Science*, 10(4), 2014.

[2] A. Bauer and M. Pretnar. Programming with algebraic effects and handlers. *J. Log. Algebr. Meth. Program.*, 84(1):108–123, 2015.

[3] E. Brady. Programming and reasoning with algebraic effects and dependent types. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, pages 133–144. ACM, 2013.

[4] S. Dolan. *Algebraic Subtyping*.

[5] S. Dolan and A. Mycroft. Polymorphism, subtyping, and type inference in mlsub. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, pages 60–72, New York, NY, USA, 2017. ACM.

[6] S. Dolan, L. White, K. Sivaramakrishnan, J. Yallop, and A. Madhavapeddy. Effective concurrency through algebraic effects. In *OCaml Workshop*, 2015.

[7] A. Faes and T. Schrijvers. A core language with row-based effects for optimised compilation. In *Student Research Competition*. ICFP, 2017.

[8] D. Hillerström and S. Lindley. Liberating effects with rows and handlers. In *Proceedings of the 1st International Workshop on Type-Driven Development*, TyDe 2016, pages 15–27, New York, NY, USA, 2016. ACM.

[9] D. Hillerström, S. Lindley, and K. Sivaramakrishnan. Compiling links effect handlers to the ocaml backend. In *OCaml Workshop*, 2016.

[10] J. R. Hindley and J. P. Seldin. *Introduction to Combinators and (lambda) Calculus*, volume 1. CUP Archive, 1986.

[11] O. Kammar, S. Lindley, and N. Oury. Handlers in action. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional programming*, ICFP '14, pages 145–158. ACM, 2013.

[12] O. Kiselyov and K. Sivaramakrishnan. Eff directly in ocaml. In *OCaml Workshop*, 2016.

[13] D. Leijen. Koka: Programming with row polymorphic effect types. *arXiv preprint arXiv:1406.2061*, 2014.

[14] D. Leijen. Type directed compilation of row-typed algebraic effects. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, pages 486–499, New York, NY, USA, 2017. ACM.

[15] J. C. Mitchell. Foundations for programming languages. 1996.

[16] B. C. Pierce. *Types and programming languages*. 2002.

[17] G. D. Plotkin and J. Power. Algebraic operations and generic effects. *Applied Categorical Structures*, 11(1):69–94, 2003.

[18] G. D. Plotkin and M. Pretnar. A logic for algebraic effects. In *Proceedings of the Twenty-Third Annual IEEE Symposium on Logic in Computer Science, LICS 2008, 24-27 June 2008, Pittsburgh, PA, USA*, pages 118–129. IEEE Computer Society, 2008.

[19] G. D. Plotkin and M. Pretnar. Handling algebraic effects. *Logical Methods in Computer Science*, 9(4), 2013.

[20] F. Pottier. *Type inference in the presence of subtyping: from theory to practice*. PhD thesis, INRIA, 1998.

[21] M. Pretnar. Inferring algebraic effects. *Logical Methods in Computer Science*, 10(3), 2014.

[22] M. Pretnar. An introduction to algebraic effects and handlers. invited tutorial paper. *Electronic Notes in Theoretical Computer Science*, 319:19–35, 2015.

[23] M. Pretnar. Eff. https://github.com/matijapretnar/eff, 2018.

[24] M. Pretnar, A. H. Saleh, A. Faes, and T. Schrijvers. Efficient compilation of algebraic effects and handlers. Technical Report CW 708, KU Leuven Department of Computer Science, 2017.

[25] A. H. Saleh, G. Karachalias, M. Pretnar, and T. Schrijvers. Explicit effect subtyping. In *European Symposium on Programming*, pages 327–354. Springer, 2018.

Master's thesis filing card

Student: Axel Faes

Title: Algebraic Subtyping for Algebraic Effects and Handlers

UDC:

Abstract:

Algebraic effects and handlers benefit from a custom type-&-effect system, a type system that also tracks which effects can happen in a program. Several such type-&-effect systems have been proposed in the literature, but all are unsatisfactory. Recently, Stephen Dolan (University of Cambridge, UK) presented a novel type system that combines subtyping and parametric polymorphism in a particulary attractive and elegant fashion. A cornerstone of his design are the algebraic properties that the subtyping relation should respect. In this work, a type-&-effect system is derived that extends Dolan's elegant type system with effect information. This type-&-effect system inherits Dolan's harmonious combination of subtyping (in our case induced by a lattice structure on the effect information) with parametric polymorphism and preserves all of its desirable properties (both low-level algebraic properties and high-level meta-theoretical properties like type soundness and the existence of principal types).

Thesis submitted for the degree of Master of Science in Engineering: Computer Science, option Artificial Intelligence

Thesis supervisor: Prof. dr. ir. Tom Schrijvers

Assessor: Amr Hany Shehata Saleh, Prof. dr. ir. Bart Jacobs

Mentor: Amr Hany Shehata Saleh