KU LEUVEN

FACULTEIT INGENIEURSWETENSCHAPPEN

Algebraic Subtyping for Algebraic Effects and Handlers

Axel Faes

Thesis voorgedragen tot het behalen van de graad van Master of Science in de ingenieurswetenschappen: computerwetenschappen, hoofdoptie Artificiële intelligentie

Promotor:
Prof. dr. ir. Tom Schrijvers

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

Begeleider:
Amr Hany Shehata Saleh

Academiejaar 2017 – 2018

Acknowledgements

This thesis has been quite a long journey. I was able to get invested into an interesting field of research and learn a lot of new things. Both about research, and about myself.

I would like to thank everybody who kept me busy and supported me the last year, especially my promoter, Tom Schrijvers and my daily advisor, Amr Hany Saleh. Without them, I wouldn't have been able to push myself forward as much as I did. 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). This type-&-effect system has been implemented in the Eff programming language in order to provide a proof-of-concept.

Nederlandstalige Samenvatting

Algebraische effecten en handlers zijn een nieuw formalisme om neveneffecten formeel te modelleren, ontworpen door Matija Pretnar (University of Ljubjana) en Gordon Plotkin (University of Edinburgh). Een voorbeeld van een neveneffect is non-determinisme of schrijven naar een bestand. Deze algebraische effect handlers kunnen vergelijkt worden met exception handlers uit Java. Echter, in Java heeft een exception handler enkel toegang tot de exception. Informatie over de besturingsstroom en de code-uitvoering op de plaats waar de exception gegooid is, is verloren.

Een algebraische effect handler heeft hier explicit toegang toe. De algebraische effect handler kan de code-uitvoering laten hervatten. Dit maakt effecten sterker en expressiever dan exceptions. Plotkin, Pretnar en Power hebben de Eff programmeertaal ontwikkelt. Deze programmeertaal maakt van algebraische effecten en handlers eerste-klas burgers.

Eff is een getypeerde programmeertaal. Elke getypeerde taal vereist een type systeem. Een type systeem bestaat uit een aantal regels die types toekennen aan de verschillende termen in de programmeertaal. Voor het optimaal gebruik van algebraische effecten en handlers is een type-&-effect systeem vereist. Dit systeem zorgt ervoor dat effect informatie (welke effecten gebruikt worden, alsook waar ze gebruikt worden) bijgehouden kan worden. Eff gebruikt een type-&-effect systeem dat gebruik maakt van subtyping constraints.

In zijn PhD thesis stelt Stephen Dolan (University of Cambridge, UK) een nieuw type systeem voor dat subtyping en parametrisch polymorphisme combineert. Dit type systeem kan subtyping constraints encoderen in een elegante structuur in de types. Dit type systeem wordt algebraic subtyping genoemd.

Aan de ene kant is het niet gemakkelijk om huidige type-&-effect systemen te gebruiken voor de optimalisatie van algebraische effecten en handlers. Dit komt door het zware gebruik van subtyping constraint. In recent onderzoek naar een geoptimaliseerde compiler voor Eff werd dit probleem ondervonden.

Aan de andere kant, Dolan's algebraic subtyping kan gezien worden als een mogelijke oplossing voor de keerzijde van subtyping constraints. Dolan vermeld algebraische effecten maar beknopt. Dolan's algebraic subtyping systeem uitbreiden met effecten kan een goed platform voorbrengen voor verder onderzoek richting algebraische effecten en handlers.

Het toevoegen van effect informatie is echter geen triviale taak. Om te zorgen dat het systeem correct blijft werken, moet de effect informatie zorgvuldig toegevoegd worden in een correcte representatie. Het is ook belangrijk om te zorgen dat de interne omgang met effecten in het systeem correct uitgevoerd wordt. Deze twee aspecten vormen de hoofdzakelijke vraagstelling van deze masterproef.

Om de vraagstelling zorgvuldig te kunnen benaderen, zijn we begonnen met een uitvoerige bestudering van de huidige literatuur. De theorie van programmeertalen vormt de ruggengraat van deze literatuur. Andere belangrijke componenten zijn de algebraische effecten en handlers alsook het algebraic subtyping systeem. Na de literatuurstudie is de integratie van de effect informatie in algebraic subtyping uitgevoerd. Verschillende eigenschappen zoals type instantiatie, type verzwakking, type substitutie, type soundness en principiële typering zijn bewezen. Uiteindelijk is het type inferentie algoritme ontworpen en is er een implementatie, een proof-of-concept, gemaakt.

De voornamelijkste contributie van deze thesis is the nieuwe type-&-effect systeem, EffCore. Dit systeem is een uitbreiding van Dolan's algebraic subtyping met effect informatie. Een volledige specificatie voor dit systeem is gegeven, inclusief termen, types, typeringsregels en equivalentie met subtyping. EffCore is afgeleid van de calculus van de Eff programmeertaal om de compatibiliteit met algebraische effecten en handlers te verzekeren. Dit betekent dat algebraische effecten en handlers eerste-klas burgers zijn in EffCore.

Het type inferentie algoritme van algebraic subtyping is uitgebreid om rekening te kunnen houden met de effect informatie. De nieuwheid van dit type inferentie algoritme bevindt zich in de inferentie van effect informatie en de inferentie van het handler type.

Algebraic subtyping bevat een type simplificatie algoritme dat geïnferreerde types naar een simpelere vorm herleidt. Dit algoritme is ook uitgebreid om te werken met effect informatie. Het simplificatie algoritme encodeerd types naar type automata en decodeerd ze achteraf. De type automata kunnen geconverteerd worden naar deterministische eindigetoestandsautomaat (DFA). Deze DFA kan versimpelt worden met standaard simplificatie algoritmes. Om het simplificatie algoritme uit te breiden met effect informatie, introduceren we het concept van type-&-effect automata, een uitbreidig van type automata.

De eigenschappen van EffCore zijn bewezen. Type instantiatie, type verzwakking, type substitutie en type soundness zijn eigenschappen van het type-&-effect systeem. Voor de type inferentie zijn de herformulatie van de typeringsregels en de principialiteit van de geïnferreerde types bewezen. De correctheid van de encodering en decodering van types en effecten naar en van type-&-effect automata zijn aangetoont. Deze bewijzen steunen op de bewijzen van Dolan in zijn thesis.

De laatste contributie is de implementatie van EffCore in de Eff programmeertaal. Deze implementaite bevat alle hoofdzakelijke kenmerken van een programmeertaal waaronder tuples, records en matching. Deze implementatie is en kan gebruikt worden

voor verdere empirische evaluatie. De implementatie is gebruikt om de performantie te testen in vergelijking met standaard subtyping. De implementatie bevat nog geen volledige implementatie van het simplificatie algoritme, hierdoor is evaluatie niet volledig accuraat. De resultaten tonen aan dat, mits het gebruik van het simplificatie algoritme, EffCore performanter is dan standaard subtyping. Dit moet echter uitgewezen worden door verder evaluatie.

Het idee van EffCore is dat het gebruikt kan worden als een verder platform voor onderzoek richting algebraische effecten en handlers. Hierdoor is er zeker de mogelijkheid naar verder onderzoek. Een eerste uitbreiding die mogelijk is, is de implementatie van het simplificatie algoritme en verder empirische evaluatie van het systeem.

Type-&-effect automata worden momenteel enkel gebruikt in het simplificatie algoritme. Dolan stelt echter een manier voor om type automata te gebruiken direct in de type inferentie om het systeem nog performanter te maken. Type-&-effect automata kunnen dus ook gebruikt worden in eenzelfde manier. Dit is echter niet verder uitgewerkt in deze masterproef.

Uiteindelijk is er de geoptimaliseerde compiler voor Eff. In de motivatie werd gesteld dat het ontwikkelen van zo'n geoptimaliseerde compiler bemoeilijkt wordt door het zware gebruik van subtyping constraints. Aangezien EffCore een uitbreiding vormt van algebraic subtyping en deze constraints kan oplossen wordt dit probleem gedeeltelijk opgelost. EffCore opent enkele deuren voor verder onderzoek naar betere optimalisatie technieken voor algebraische effecten en handlers met EffCore.

Ter conclusie kan er gestelt worden dat de doelstelling van de masterproef bereikt is. De uitkomst is een type-&-effect systeem, EffCore, en een implementatie. Ik toon aan dat algebraic subtyping gecombineerd kan worden met algebraische effecten en handlers. Uiteindelijk, EffCore en de implementatie vormen een springplank voor verder onderzoek.

List of Figures

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]. From a usability perspective, they are exception handlers on steroids. Exception handlers have no way to access the continuation after raising the exception. Effect handlers do have this access and can do multiple applications of this continuation. The explicit access and the multiple applications of the continuation is the aspect that makes algebraic effects very versatile.

Algebraic effects and handlers are becoming more popular, not only as a formalism but also as a practical feature in programming languages (e.g. the Koka language developed by Microsoft Research [12]). Plotkin, Pretnar and Power have developed the programming language Eff [17, 18], which makes algebraic effects and handlers first-class citizens. This means you can declare effects and handlers and pass them around as function arguments.

Figure 1.1 shows an example written in Eff. Users can define their own effects, in this case, a $Decide$ effect. The $choose$ function uses the effect to decide whether to return 10 or 20. $choose\_true$ uses a handler to handle the $choose$ function. k represents the continuation, which needs a boolean argument to be able to continue the remainder of the computation. In this example, it calls the continuation with $true$, thus $choose\_true$ returns the value 10.

1.1 Motivation

Any typed programming language requires a type system in order to function [16, Section 1.1]. A type system governs a set of rules that assigns types to the various constructions in that typed programming language. Algebraic effects and handlers benefit from a custom type-&-effect system, a type-&-effect system that also tracks which effects can happen in a program. Several such type-&-effect systems have been proposed in the literature [12, 19, 1].

Figure 1.1: The Eff programming language

```
effect Decide : unit -> bool;;

let choose () = if (#Decide ()) then 10 else 20;;

let choose_true =
  handle (choose ()) with
    | #Decide () k -> k true;;
```

In his December 2016 PhD thesis, 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. This system is called algebraic subtyping. [5]

On one hand, current type-&-effect systems for algebraic effects and handlers make optimizations more difficult because of the heavy use of subtyping constraints. This can be seen in recent research where an optimizing compiler for Eff was developed [24]. Within this research, the main hurdle here involved working with, instead of against, the type-&-effect system of Eff based on subtyping.

On the other hand, Dolan's algebraic subtyping system can be seen as a method to overcome the drawbacks of subtyping constrains . However, in his work, Dolan only mentions how to deal with effects briefly. Therefore, extending Dolan's algebraic subtyping system with effects can provide a better platform for research that requires heavy use of subtyping constraints such as optimization of Eff.

1.2 Research questions

From the motivation, we can conclude that current type-&-effect systems can have some drawbacks when it comes to some areas of research such as optimizations. Dolan proposes a type system which has the potential to solve these problem. However, Dolan's type system does not track which effects can happen in a program. Adding effects into a type system is not a trivial challenge. The method for keeping track of the effects needs to be compatible with the rest of the type system. Therefor, our research questions can be summarized as the following:

- How can Dolan's 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 type system?

1.3 Approach

The goal of this thesis, as well as it's main contribution, is to derive a type-&-effect system that extends Dolan's 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).

1. Study of the relevant literature and theoretical background. This includes general programming language theory, algebraic effects and handlers, and algebraic subtyping.
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 instantiation, type weakening, type substitution, type soundness, reformulating the typing rules, principal typing, and encoding and decoding in type automata.
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.

1.4 Results

A novel type-&-effect system, EffCore, is provided. This system extends Dolan's algebraic subtyping system with effect information. We give a full specification for this system including terms, types, typing rules and equivalence rules to subtyping. EffCore is derived from the calculus of the Eff programming language in order to ensure compatibility with algebraic effects and handlers. This also means that algebraic effects and handlers are first-class citizens in EffCore.

We have extended the type inference algorithm for algebraic subtyping to account for the effect information. The novelty in the type inference algorithm involves effect inference, as well as type inference for the handler type.

The type simplification algorithm for algebraic subtyping that simplifies inferred types has been extended to account for the effect information. This algorithm encodes types into type automata and afterwards decodes them into types again. These type automata can easily be converted into deterministic finite automata to use standard simplification algorithms. The extended simplification algorithm introduces type-&-effect automata which extend type automata.

We prove the properties of EffCore. For the type-&-effect system, this involves type instantiation, type weakening, type substitution and type soundness. For the type

inference algorithm, the reformulation of the typing rules and the principality of the inferred types have been proven. The encoding and decoding of types and effects into type-&-effect automata, an extension of type automata have been proven correct. Many of these proofs built on top of the proofs made in Dolan's thesis.

Finally, another result of this work 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. This implementation has been empirically evaluated by benchmarking the type inference performance against a subtyping-based system, the type-&-effect system previously used in Eff. The inferred types have also been manually compared in order to show the difference in interpretability.

1.5 Structure of the Thesis

Chapter 2 provides the required background in programming language theory, algebraic effects and optimizing Eff. How to read a type system specification is found in Section 2.1. This section uses the simply typed lambda calculus. Section 2.2 gives a thorough explanation of algebraic effects and handlers. The final section, the optimization of Eff, explains the research that led to this thesis. An explanation of the optimization techniques is given and the hurdles encountered during the research are explained.

Chapter 3, 4 and 5 define the EffCore type system. This is the main contribution 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 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. Most proofs are provided inline. Only the larger proofs are given in the Appendix. The instantiation proof is given in Appendix A. The soundness proof is given in Appendix B. Finally, the proof of equivalence of typing rules is given in Appendix C.

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. [16, Chapter 9]

2.1.1 Types and terms

Types and terms are necessary components of the *simply typed lambda calculus*. A simple 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. [9]

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].

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

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;;
```

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.

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.

Figure 2.4: Algebraic effect continuation usage example

```
1  effect Op : unit -> int;;
2
3  let someFun b =
4    handle (
5      if (b == 0) then
6        let a = #Op () in
7        print a
8      else
9        print b
10   ) with
11     | #Op () k -> k 1;;
```

Figure 2.5: Algebraic effect continuation example

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

[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
	$\mid$	$\texttt{true}$	true
	$\mid$	$\texttt{false}$	false
	$\mid$	$\lambda x.c$	function
	$\mid$	$\{$	handler
		$\quad \texttt{return}\ x \mapsto c_r,$	return case
		$\quad [\texttt{Op}\ y\ k \mapsto c_{\texttt{Op}}]_{\texttt{Op} \in \mathcal{O}}$	operation cases
		$\}$	
comp c	$::=$	$v_1\ v_2$	application
	$\mid$	$\texttt{do}\ x \leftarrow c_1\ ;\ c_2$	sequencing
	$\mid$	$\texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2$	conditional
	$\mid$	$\texttt{let rec}\ f\ x = c_1\ \texttt{in}\ c_2$	rec definition
	$\mid$	$\texttt{return}\ v$	returned val
	$\mid$	$\texttt{Op}\ v$	operation call
	$\mid$	$\texttt{handle}\ c\ \texttt{with}\ v$	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	$::=$	$\texttt{bool}$	bool type
	$\mid$	$A \to \underline{C}$	function type
	$\mid$	$\underline{C} \Rightarrow \underline{D}$	handler type
dirty type $\underline{C}, \underline{D}$	$::=$	$A\ !\ \Delta$	
dirt Δ	$::=$	$\{\texttt{Op}_1, \ldots, \texttt{Op}_n\}$	

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

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

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

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

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

Figure 2.9: Typing of expressions 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,

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

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

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

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 optimizations are brievely discussed. 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, 8]. Most implementations take the form of libraries [3, 10, 11]. 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].

Simplification

$$\textsc{App-Fun}\quad \frac{}{(\lambda x.c)\, v \rightsquigarrow c[v/x]}$$

Handler Reduction

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

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

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

Figure 2.12: Term Rewriting Rules [24]

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.

Several of the term rewriting rules that were created during the creation of the optimised compiler are 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

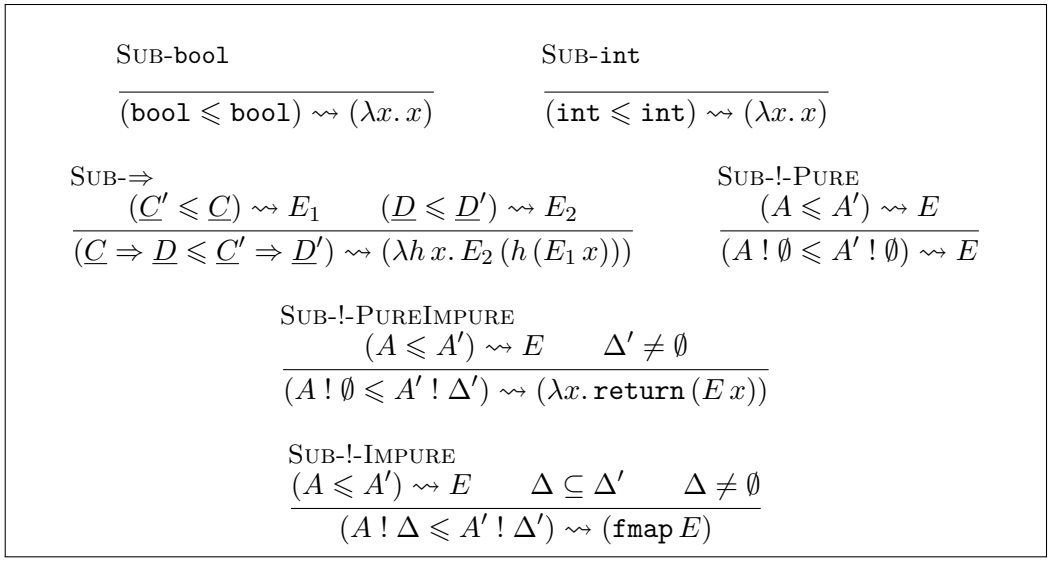

Figure 2.13: Subtyping induced coercions [24]

computation c is elaborated into an OCaml expression E. The different elaboration judgements for computations can be seen in Figure 2.14, the laboration judgements for expressions have been omitted. 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 needed to be evaluated empirically with several testing programs. Evaluation happened in two stages. In the first stage, the performance of the optimizing compiler was compared against hand-written OCaml code. In the second stage, the optimising compiler was compared against several other systems that support algebraic effects and handlers, such as Multicore OCaml.

The results were 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 an optimising compiler approach is consistently the fastest.

Computations

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

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

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

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

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

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

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

Figure 2.14: Type-&-effect-directed purity aware compilation for computations [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 [13, 7, 12]. Such an explicitly types calculus with row-based effects has been implemented in earlier research [6]. 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.

Chapter 3

Algebraic Subtyping for Eff

This chapter is the start of the novel work that is accomplished. 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
	$\mid$	$\hat{\mathtt{x}}$	let-variable
	$\mid$	$\mathtt{true}$	true
	$\mid$	$\mathtt{false}$	false
	$\mid$	$\lambda x.c$	function
	$\mid$	$\{$	handler
		$\quad \mathtt{return}\ x \mapsto c_r,$	$\quad$ return case
		$\quad [\mathtt{Op}\ y\ k \mapsto c_{\mathtt{Op}}]_{\mathtt{Op} \in \mathcal{O}}$	$\quad$ operation cases
		$\}$	
comp c	$::=$	$v_1\ v_2$	application
	$\mid$	$\mathtt{do}\ \hat{\mathtt{x}} = c_1\ ;\ c_2$	sequencing
	$\mid$	$\mathtt{let}\ \hat{\mathtt{x}} = v\ \mathtt{in}\ c$	let
	$\mid$	$\mathtt{if}\ e\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2$	conditional
	$\mid$	$\mathtt{return}\ v$	returned val
	$\mid$	$\mathtt{Op}\ v$	operation call
	$\mid$	$\mathtt{handle}\ c\ \mathtt{with}\ v$	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.

An important aspect is the semantics given to the dirt. The dirt intersection is used to

indicate that the operations are allowed to happen in an input, while dirt union is used in outputs. This, for example, applies to the handler type. A handler type $\underline{C} \Rightarrow \underline{D}$ will generally use dirt intersection in $\underline{C}$ and dirt union in $\underline{D}$. In Chapter 4.1, a restriction will be placed on the structure of types that makes this difference explicit.

Structuring dirts using intersections and dirts as defined by algebraic subtyping, we can take advantage of Dolan's framework. This includes polarity of types and biunification. If dirts were structured using set operations, we wouldn't be able to take advantage of this system.

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.

Finally, algebraic subtyping makes one restriction. Subtyping relationships between a function and a boolean, handler and boolean, and function and handler are not allowed. [4]

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

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

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

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

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

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

Figure 3.3: Relationship between Equivalence and Subtyping

3.3 Typing Rules

Figure 3.8 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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Figure 3.6: Equations for function, handler and dirty types

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

The first property is instantiation. Instantiation allows us to replace type variables with types through the means of a type derivation. Instantiation is a property that holds for Algebraic subtyping [4]. The proof given here is a straightforward extension from the standard proof.

The (pure) types as defined in Figure 3.2, with the exclusion of the handler type $\Rightarrow$, is equivalent to the types from algebraic subtyping. The typing rules from EffCore add some new rules regarding the handlers and the dirts. Excluding these new rules, the typing rules are also equivalent to the algebraic subtyping typing rules. The main distinction is the separation of the typing rules into expressions and computations. This separation is only used to make a distinction and thus does not change any definition. This means that the proofs are also extensions from the proofs from algebraic subtyping, rather than radically different proofs.

Two type schemes are alpha-equivalent if they only differ in the naming of variables. When two typing contexts are called equivalent, they have the same domain. They assign equal types to λ-bound variables and alpha-equivalent typing schemes to let-bound variables. The following proposition states that typing also respects this equivalence relation. This holds for both expressions and computations.

Proposition 1. *(Equivalence of typing contexts)*
If $\Gamma_1 \vdash v : A$ *and* $\Gamma_1 \equiv \Gamma_2$ *then* $\Gamma_2 \vdash v : A$
If $\Gamma_1 \vdash v : A$ *and* $\Gamma_1 \equiv \Gamma_2$ *then* $\Gamma_2 \vdash v : A$

The proof for this proposition is a straightforward induction on derivations. Considering that typing contexts in EffCore behave exactly the same as typing contexts from algebraic subtyping, the proof is omitted in this thesis as it did not change. Instantiation allows us to apply a substitution to a typing context and the pure or dirty type while preserving validity. A substitution can map typing and dirt variables to types and dirts.

Theorem 1. *(Instantiation of pure types) If* $\Gamma \vdash v : A$ *then* $\rho(\Gamma) \vdash v : \rho(A)$

Theorem 2. *(Instantiation of dirty types) If* $\Gamma \vdash c : \underline{C}$ *then* $\rho(\Gamma) \vdash c : \rho(\underline{C})$

The proofs for both of these theorems are made in Appendix A. There are no note-worthy cases in this proof regarding novelty of this thesis. However, considering EffCore has typing rules which did not exist in algebraic subtyping, the proof is given for completion.

3.4.2 Weakening

Weakening allows us to weaken any type derivation. This is done by making stronger assumptions in the typing context. More concretely, this means that the weaker typing context may contain more variable mapping than is actually necessary, it may assign

alpha-equivalent typing schemes to let-bound variables and it may assign subtypes to all λ-bound variables. We write $\Gamma_2 \leqslant \Gamma_1$ when $dom\Gamma_2 \supseteq dom\Gamma_1$ and $\forall \hat{\mathbf{x}} \in \Gamma_1 : \Gamma_2(\hat{\mathbf{x}}) \equiv \Gamma_2(\hat{\mathbf{x}})$ and $\forall x \in \Gamma_1 : \Gamma_2(x) \leqslant \Gamma_1(x)$.

Theorem 3. *(Weakening of pure types) If* $\Gamma_1 \vdash v : A$ *and* $\Gamma_2 \leqslant \Gamma_1$ *then* $\Gamma_2 \vdash v : A$

Theorem 4. *(Weakening of dirty types) If* $\Gamma_1 \vdash c : \underline{C}$ *and* $\Gamma_2 \leqslant \Gamma_1$ *then* $\Gamma_2 \vdash c : \underline{C}$

The proofs for both of these theorems are not given in their entirety. We can construct the proof by induction of the derivations which is mostly straightforward. True is an example of such a straightforward case, $\Gamma_1 \vdash \texttt{true} : bool$ and $\Gamma_2 \leqslant \Gamma_1$ trivially leads to $\Gamma_2 \vdash \texttt{true} : bool$.

The non-trivial cases are Var-λ, Var-$\forall$, Fun, Let Do and Hand. For all these cases, except for Hand, the original proof from Dolan still applies. For Var-λ, we have $(x : A) \in \Gamma_1$ which can be written as $\Gamma_1(x) = A$. Due to $\Gamma_2 \leqslant \Gamma_1$, we know that $\Gamma_2(x) \leqslant A$. By applying SubVal, we achieve our result. For Var-$\forall$, the resulting type schemes are alpha-equivalent, thus the case is valid.

For Fun, $\Gamma_2 \leqslant \Gamma_2$ implies that $\Gamma_2, x : A \leqslant \Gamma_1, x : A$, which causes the inductive hypothesis to apply. The exact same reasoning applies to the Hand case, where we use it for the value case and effect clauses. Finally, there is the Let and Do. The difficulty here is the condition that $\alpha \notin FTV(\Gamma)$. We can use Theorem 1 and Theorem 2 to make sure this condition applies. Afterwards, we can apply the weakening.

3.4.3 Substitution

EffCore contains several kinds of variables. For types, there are λ-bound variables and let-bound variables. For dirts, there dirt variables. We need two substitution theorems for, respectively, λ-bound and let-bound variables. However, we do not need a substitution theorem for dirt variables. The reasoning behind this is that the typing context can only contain pure types, but not dirty types.

EffCore models its algebraic effects and handlers to Eff. Eff executes effects as soon as they are seen. The code `let x = Op true`, will execute *effect Op* and the result of that effect will be stored in `x`. Thus, it would not make sense for a typing context to directly contain effects.

Theorem 5. *(Substitution λ-bound for pure types) If* $\Gamma \vdash v : A, \Gamma(x) = B$ *and* $\Gamma_x \vdash e' : B$ *then* $\Gamma_x \vdash v[e'/x] : A$

Theorem 6. *(Substitution λ-bound for dirty types) If* $\Gamma \vdash c : \underline{C}, \Gamma(x) = B$ *and* $\Gamma_x \vdash e' : B$ *then* $\Gamma_x \vdash c[e'/x] : \underline{C}$

EffCore does not change the monomorphic types (in the monomorhpic typing context) from algebraic subtyping. Thus the original substitution proof cannot be invalidated.

Handlers abstract variables in the same way that functions abstract variables and thus, cannot cause issues.

Theorem 7. *(Substitution let-bound for pure types) If* $\Gamma \vdash v : A, \Gamma(\hat{\boldsymbol{x}}) = \forall \bar{\alpha}.B$ *and* $\forall \bar{A} : \Gamma_{\hat{\boldsymbol{x}}} \vdash e' : B[\bar{A}/\alpha]$ *then* $\Gamma_x \vdash v[e'/\hat{\boldsymbol{x}}] : A$

Theorem 8. *(Substitution let-bound for dirty types) If* $\Gamma \vdash c : \underline{C}, \Gamma(\hat{\boldsymbol{x}}) = \forall \bar{\alpha}.B$ *and* $\forall \bar{A} : \Gamma_{\hat{\boldsymbol{x}}} \vdash e' : B[\bar{A}/\alpha]$ *then* $\Gamma_x \vdash c[e'/\hat{\boldsymbol{x}}] : \underline{C}$

EffCore does not change the polymorphic types (in the polymorphic typing context) from algebraic subtyping. Thus the original substitution proof cannot be invalidated.

3.4.4 Soundness

Now, we can proof the soundness of our system. This means that we need to show that typable programs in EffCore are valid and do not go wrong. This is done with small-step-call-by-value operational semantics, determined by a relation $c \longrightarrow c'$ defined in Figure 3.7. One peculiarity can be noticed in the `handle` $\texttt{Op}v$ `with` h rules. These rules contain an extra function $.\lambda y.c$ which represents an implicit continuation. This continuation is not visible to the user of EffCore and is only used for the operational semantics.

Theorem 9. *(Value inversion)*
If $\vdash v : A \to \underline{C}$ *then* $v = \lambda x.$
If $\vdash v : \underline{C} \Rightarrow \underline{D}$ *then* $v = \{\texttt{return}\ x \mapsto c_r, [\texttt{Op}\ y\ k \mapsto c_{\texttt{Op}}]_{\texttt{Op} \in \mathcal{O}}\}$
If $\vdash v : bool$ *then* $v \in \{\texttt{true}, \texttt{false}\}$

Theorem 10. *(Progress) If* $\vdash c : A$ *then* c *is either a* `return` v*, a* $\texttt{Op}v$ *or* $c \longrightarrow c'$ *for some* c'

Theorem 11. *(Preservation) If* $\vdash c : \underline{C}$ *and* $c \longrightarrow c'$ *then* $\vdash c' : \underline{C}$

These two theorems make up the soundness theorem for EffCore. The Progress theorem states that we do not get stuck when evaluating a computation. Either we end up with a `return` v or $\texttt{Op}v$, which means the evaluation has ended, or we can continue evaluating. Values v are not mentioned since they do not require further evaluation, hence the name values. The Preservation theorem states that the small-step operational semantics produce a valid term. Theorem 17 is an additonal theorem that helps us proof the other two theorems. The proofs are given in Appendix B.

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

$$\textsc{App}\quad (\lambda x.c)v \longrightarrow c[v/x] \qquad \textsc{Cond-True}\quad \texttt{if true then } c_1 \texttt{ else } c_2 \longrightarrow c_1$$

$$\textsc{Cond-False}\quad \texttt{if false then } c_1 \texttt{ else } c_2 \longrightarrow c_2 \qquad \textsc{Doin-C}\quad \frac{c_1 \longrightarrow c_1'}{\texttt{do } \hat{\mathbf{x}} = c_1 \texttt{ ; } c_2 \longrightarrow \texttt{do } \hat{\mathbf{x}} = c_1' \texttt{ ; } c_2}$$

$$\textsc{Doin-Ret}\quad \texttt{do } \hat{\mathbf{x}} = \texttt{return } v \texttt{ ; } c_2 \longrightarrow c_2[(\texttt{return } v)/x]$$

$$\textsc{Doin-Op}\quad \texttt{do } \hat{\mathbf{x}} = \texttt{Op}v \texttt{ ; } c_2 \longrightarrow c_2[(\texttt{Op}v)/x] \qquad \textsc{Let}\quad \texttt{let } \hat{\mathbf{x}} = v \texttt{ in } c \longrightarrow c[v/\hat{\mathbf{x}}]$$

$$\textsc{With-C}\quad \frac{c \longrightarrow c'}{\texttt{handle } c \texttt{ with } h \longrightarrow \texttt{handle } c' \texttt{ with } h}$$

$$\textsc{With-Ret}\quad \texttt{handle return } v \texttt{ with } h \longrightarrow c_r[v/x]$$

$$\textsc{With-Op-Handled}\quad \frac{h \text{ handles } \texttt{Op}}{\texttt{handle Op}v.\lambda y.c \texttt{ with } h \longrightarrow c_{\texttt{Op}}[v/x, (\lambda y.(\texttt{handle } c \texttt{ with } h))/k]}$$

$$\textsc{With-Op-Not-Handled}\quad \frac{h \text{ does not handle } \texttt{Op}}{\texttt{handle Op}v.\lambda y.c \texttt{ with } h \longrightarrow \texttt{Op}v.\lambda y.(\texttt{handle } c \texttt{ with } h)}$$

Figure 3.7: Small-step transition relation

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 3.9. 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 3.8. The reformulated typing rules are given in Figure 3.10. 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.

The equivalence of the original and reformulated typing rules are shown in Appendix C. The proof is made by creating a mapping between regular typing contexts and the monomorphic and polymorphic typing contexts.

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

Expressions

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

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

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

Computations

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

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

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

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

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

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

Figure 3.8: 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
$\Pi_2 \leqslant^{\forall} \Pi_1 \leftrightarrow dom(\Pi_2) \supseteq dom(\Pi_1), \Pi_2(\hat{\mathbf{x}}) \leqslant^{\forall} \Pi_1(\hat{\mathbf{x}}) \mid \hat{\mathbf{x}} \in dom(\Pi_1)$

WeakeningType
If $\Pi_1 \vdash v : [\Xi]A$ and $\Pi_2 \leqslant^{\forall} \Pi_1$ then $\Pi_2 \vdash v : [\Xi]A$
If $\Pi_1 \vdash c : [\Xi]\underline{C}$ and $\Pi_2 \leqslant^{\forall} \Pi_1$ then $\Pi_2 \vdash c : [\Xi]\underline{C}$

Figure 3.9: 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

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

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

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

Computations

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

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

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

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

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

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

Figure 3.10: 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]

To operate on polar type terms, we generalise from Hindley-Milner unification to bi-unification. Biunification is a subtyping-aware analogue for unification in EffCore. Unification is the process used to eliminate constraints.

We extend the biunification algorithm with support for algebraic effects and handlers. A unification algorithm uses substitutions in order to substitute type variables. Biunification uses bisubstitutions, an analogue for substitutions. Once we have the bisubstitutions, we build the constraint solving algorithm. This algorithm eliminates basic constraints by introducing a bisubstitution.

Not all constraints that are generated in a program are simple constraints. A constraint decomposition algorithm is needed for this purpose. Once we have the ability to decompose and solve constraints, we can build a biunification algorithm.

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.

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 4.6. [4]

4.2 Bisubstitutions

The bisubstitution algorithm is given in Figure 4.2. A bisubstitution $\xi = [A^+/\alpha^+, A^-/\alpha^-, \Delta^+/\delta^+, \Delta^-/\delta^-]$ maps positive occurrences of α to A^+, negative occurrences of α to A^-, positive occurrences of δ to Δ^+ and negative occurrences of δ to Δ^-.

The different rules in Figure 4.2 show how a bisubstitution moves through a type based on polarity. Most rules, except for the handler and the dirt related rules are standard for algebraic subtyping.

Bisubstitutions can be applied to typing schemes. $\xi([\Xi^-]A^+)$ applies the bisubstitution to A^+ and to every typ of the lambda-bound typing context Ξ^-.

4.3 Constraint Solving

Atomic constraints can be solved immediately. Atomic type constraints are constraints between type variables and constructed types. Atomic dirt constraints are constraints between dirt variables and the basic operation.

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

Constructed types are given in Figure 4.3. Functions, handlers and bools are constructed types. Constructed types from EffCore consists of the constructed types from algebraic subtyping suplemented with handlers.

Figure 4.4 shows the bisubstitutions that need to be used in order to solve each atomic constraint. Only the dirt related constraint solver rules are novel, the others remain equivalent to the rules from algebraic subtyping. Constraints of the form $\beta \leqslant A^-$ and $A^+ \leqslant \beta$ have two different solvers that can be used, depending on whether β is a free variable. This is necessary due to recursion. If β is not free, then no recursion needs to be introduced. These bisubstitutions introduce the union and intersection types into an inferred type.

Bisubstitution

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

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

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

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

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

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

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

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

$$\xi(\mathit{bool}) \equiv \mathit{bool} \qquad \xi(\mathit{bool}) \equiv \mathit{bool}$$

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

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

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

Figure 4.2: Bisubstitutions

Constructed (predicate):
constructed(A)

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

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

$$\mathit{constructed}(\mathit{bool})$$

Figure 4.3: Constructed types

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

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

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

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

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

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

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

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

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

Figure 4.4: Constraint solving

4.4 Constraint Decomposition

When presented with a non-atomic constraint, the biunification algorithm needs to decompose the constraint into subconstraints. It needs to do this until only atomic constraints remain. Figure 4.5 shows the constraint decomposition algorithm $subi$. $subi$ is a partial function meaning that any constraint which does not occur in this list will not decompose. Later we will see that any non-atomic constraint that cannot be decomposed, cannot be solved.

There are three inputs to $subi$. It accepts constraints between pure types, dirty types and dirts. The rules not relating to dirty types, handlers and dirts are equivalent to decomposition rules from algebraic subtyping.

Compared to the standard algebraic subtyping, the dirts have some peculiarities. For types, there is only one way to decompose $A^+ \leqslant A_1^- \sqcap A_2^-$. For dirts, there are multiple ways to decompose.

In the reformulated typing rules in Figure 3.10, handlers are given the type $A \,!\, \Delta \sqcap \mathcal{O} \Rightarrow$

$B \,!\, \Delta$. It is unusual to introduce intersections or unions directly. Here it is necessary since we want to explicitly state that the handler can handle certain operations.

Another reason why multiple rules are needed is due to the nature of dirts. A constraint $\mathtt{Op} \leqslant \mathtt{Op} \sqcap \Delta^-$ and $\mathtt{Op} \leqslant \Delta^- \sqcap \mathtt{Op}$ are immediately satisfied. This can be related to the fact that dirts represent an over-approximation of possible side-effects. While constraints such as $\mathtt{Op}_1 \leqslant \Delta^- \sqcap \mathtt{Op}_2$ and $\mathtt{Op}_1 \leqslant \mathtt{Op}_2 \sqcap \Delta^-$, we do not actually care about $\mathtt{Op}_2$ and are more interested in the $\mathtt{Op}_1 \leqslant \Delta^-$ aspect.

Another way to explain why these rules are needed is that dirts act different from types. When we have a typing constraint such as $unit \leqslant int$, we have a typing error in our program. Such an atomic constraint can be decomposed from $unit \leqslant A^- \sqcap int$. However, for dirts, $\mathtt{Op}_1 \leqslant \Delta^- \sqcap \mathtt{Op}_2$ can be interpreted as: can $\mathtt{Op}_1$ be handled by a handler which handles the effects $\Delta^- \sqcap \mathtt{Op}_2$.

Proposition 2. $\forall A^+, A^-, \Delta^+, \Delta^-$:
$A^+ \leqslant A^-$ *is an atomic constraint,* $A^+ \leqslant A^-$ *is unsatisfiable,* $subi(A^+ \leqslant A^-)$ *succeeds*
$\Delta^+ \leqslant \Delta^-$ *is an atomic constraint,* $\Delta^+ \leqslant \Delta^-$ *is unsatisfiable,* $subi(\Delta^+ \leqslant \Delta^-)$ *succeeds*

$subi$ is defined in such a way that any valid constraint can either be decomposed or is atomic. This means that, otherwise, the constraint has to be unsatisfiable. This can be trivially proven by case analysis over all possible instances of A^+ and A^-. All cases are trivial in nature (due to polar types) and the full formal proof has therefor been omitted. Constraint $A_1 \sqcap A_2 \leqslant A_3$ cannot be simply decomposed, but this constraint cannot exist due to the nature of polar types.

4.5 Biunification

The biunification algorithm solves a set of constraints and produces a bisubstitution that needs to be applied in order to solve for the constraints. The algorithm is shown in Figure 4.6. An additional argument H is used to act as a history so constraints that have already been seen can be skipped. The biunification algorithm is equivalent to the standard biunification algorithm from algebraic subtyping [4].

Start is used to introduce the history argument. The Empty rule is matched when all constraints have been solved. Redundant skips constraints that have already been seen. Atomic is used to solve atomic constraints. After solving the atomic constraint, a bisubstitution θ_c is obtained. This bisubstitution is applied to the constraint set that still needs to be solved, as well as to the history. In a recursive call, the other constraints are solved. The bisubstitution θ_c is concatenated to the bisubstitutions that will be obtained later, until it is concatenated to the neutral element 1. Concatenated bisubstitutions are, during the type inference, applied one by one. Decompose decomposes constraints, this is also the only rule that can fail in the case that it needs to decompose an unsatisfiable constraint.

Theorem 12. *If* $biunify(C)$ *fails, then* C *is unsatisfiable.*

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

This can be proven by induction on the rules of $biunify(C)$. The only way that a constraint can fail to be solved is that: the constraint has not yet been seen, is not atomic and cannot be decomposed.

Theorem 13. *If $biunify(C) = \xi$, then ξ solves C*

There are three kinds of constraints, constraints between pure types, between dirty types and between dirts. The types of EffCore are equivalent to the types of algebraic subtyping in combination with a handler type. Since the handler type behaves exactly like the function type, the original proof of this theorem (for pure types) cannot be invalidated. The structure for dirts is a subset of the structure of types (no functions, or handlers). Thus the proof also has to hold.

The original proof works by induction. Empty is trivial: if there are no constraints, an empty substitution solves them. The inductive hypothesis applies to Decompose since $c :: C$ and $subi(c) +\!\!+ C$ have the same atomic subconstraints and are thus equivalent. Dolan shows that bisubstitutions can be concatenated and uses that to proof that the inductive hypothesis also applied to the Atomic rule. The Redundant rule is also trivial, since by definition of that rule, the constraint has already been solved before.

Binunify(History, ContraintSet) = substitution

Start
$$biunify(C) = biunify(\emptyset; C)$$

Empty
$$biunify(H; \epsilon) = 1$$

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

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

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

Figure 4.6: Biunification algorithm

4.6 Principal Type Inference

We introduce a judgement form $\Pi \triangleright v : [\Xi^-]A^+$, stating that $[\Xi^-]A^+$ is the principal typing scheme of v under the polar typing context Π. Similarly, $\Pi \triangleright c : [\Xi^-]\underline{C}^+$ states that $[\Xi^-]\underline{C}^+$ is the principal typing scheme of v under the polar typing context Π.

Principality for most syntactic elements are equivalent to the principality from algebraic subtyping [4]. The novelty in presented in Chapter 4.6.2, this is the principality of the algebraic effects and handler related syntactic elements. The other cases for principality are included in order to provide the complete type inference algorithm.

The following subsections discuss the principality for all different constructs. The final type inference rules can be found in Figure 4.7 for expressions and Figure 4.8 for computations.

4.6.1 Principality for Functions

We begin with the principality of λ-bound variables, functions and applications. In Figure 4.7 and Figure 4.8, these are represented by the inference rules VAR-Ξ, FUN and APP.

λ-bound variables λ-bound variables x are typed with the VAR-Ξ from Figure 3.10. in this rule, a λ-bound variable x can be given the typing scheme $[x : A]A$. The principal typing scheme is the scheme that subsumes any typing scheme of the form $[x : A]A$, this is the typing scheme $[x : \alpha]\alpha$.

Functions Functions $\lambda x.c$ are typed with the FUN rule from Figure 3.10. This requires that c can be typed. Otherwise, if c cannot be typed, then $\lambda x.c$ cannot be typed. In that case, there is no principal type. If c can be typed, it has the principal polar typing scheme $[\Xi^-]\underline{C}^+$.

The general form of a typing derivation for $\lambda x.c$ is:

$$\textsc{Fun}\ \frac{\textsc{SubComp}\ \dfrac{\Pi \Vdash c : [\Xi_1^-]\underline{C}^+}{\Pi \Vdash c : [\Xi_2, x : A]\underline{C}}}{\Pi \Vdash \lambda x.c : [\Xi_2](A \to \underline{C})}$$

SUBCOMP can only be applied if $[\Xi_1^-]\underline{C}^+ \leqslant^\forall [\Xi_2, x : A]\underline{C}$. We can write this as: $[\Xi_1^- \setminus x, x : \Xi_1^-(x)]\underline{C}^+ \leqslant^\forall [\Xi_2, x : A]\underline{C}$. By function subtyping, this has to be equivalent to: $[\Xi_1^- \setminus x](\Xi_1^-(x) \to \underline{C}^+) \leqslant^\forall [\Xi_2](A \to \underline{C})$. From this, we can conclude that $[\Xi_1^- \setminus x](\Xi_1^-(x) \to \underline{C}^+)$ is the principal type of a function.

Applications Applications $v_1\, v_2$ are typed with the APP rule from Figure 3.10. Both v_1 and v_2 need to be typable in order to be able to type $v_1\, v_2$. Let the principal types of v_1 and v_2 be $[\Xi_1^-]A_1^+$ and $[\Xi_2^-]A_2^+$.

The general form of a typing derivation for $v_1\, v_2$ is:

$$\textsc{Fun}\ \frac{\textsc{SubVal}\ \dfrac{\Pi \Vdash v_1 : [\Xi_1^-]A_1^+}{\Pi \Vdash v_1 : [\Xi](B \to \underline{D})} \qquad \textsc{SubVal}\ \dfrac{\Pi \Vdash v_2 : [\Xi_2^-]A_2^+}{\Pi \Vdash v_2 : [\Xi]B}}{\Pi \Vdash v_1\, v_2 : [\Xi]\underline{D}}$$

SubVal can only be applied if $[\Xi_1^-]A_1^+ \leqslant^\forall [\Xi](B \to \underline{D})$ and $[\Xi_2^-]A_2^+ \leqslant^\forall [\Xi]B$. Since typing schemes are closed, $[\Xi_1^-]A_1^+$ and $[\Xi_2^-]A_2^+$ do not share typing variables. Therefor, we can use a (bi)substitution to find the most principal scheme. We get $\rho(A_1^+) \leqslant B \to \underline{D}$, $\rho(A_2^+) \leqslant B$, $\Xi \leqslant \rho(\Xi_1^-)$ and $\Xi \leqslant \rho(\Xi_2^-)$.

By introducing type variables α and β and a dirt variable δ and taking $B = \rho(\alpha)$ and $\underline{D} = \rho(\beta\ !\ \delta)$, we get other constraints $\rho(A_1^+) \leqslant \rho(\alpha \to (\beta\ !\ \delta))$, $\rho(A_2^+) \leqslant \alpha$ and $\rho([\Xi_1^- \sqcap \Xi_2^-](\beta\ !\ \delta)) \leqslant [\Xi]\underline{D}$. Any type that solves these constraints, types the application using the principal types for v_1 and v_2. The reason we see $[\Xi_1^- \sqcap \Xi_2^-]$ is that that set of variables is the least set that is needed.

In similar reason to the principality for λ-bound variables, we know that $\rho([\Xi_1^- \sqcap \Xi_2^-](\beta\ !\ \delta)$ is the most principal type for the application. The bisubstitution ρ does need to be chosen correctly for this. The correct choice is any bisubstitution ξ that solves the constraints $\{A_1^+ \leqslant (\alpha \to (\beta\ !\ \delta)), A_2^+ \leqslant \alpha\}$.

The bisubstitution ξ can be computed using the biunification algorithm. We can make one final optimization. We can use the following simplified constraint set $\{A_1^+ \leqslant (A_2^+ \to (\beta\ !\ \delta))\}$. α is only used twice. When $A_2^+ \leqslant \alpha$ is being solved, A_2^+ will be substituted into $A_1^+ \leqslant (\alpha \to (\beta\ !\ \delta))$, which becomes $A_1^+ \leqslant (A_2^+ \to (\beta\ !\ \delta))$.

4.6.2 Principality for Handlers

We continue with the principality of handlers, handling, return and operations. In Figure 4.7 and Figure 4.8, these are represented by the inference rules Hand, With, Ret and Op.

Handlers A handler $\{\texttt{return}\ x \mapsto c_r, [\texttt{Op}\, y\, k \mapsto c_{\texttt{Op}}]_{\texttt{Op} \in \mathcal{O}}\}$ is typed with the Hand rule from Figure 3.10. A handler has a value case $\texttt{return}\ x \mapsto c_r$ and some effect cases $\texttt{Op}\, y\, k \mapsto c_{\texttt{Op}}$ for each operation that the handler handles. If c_r or any of $c_{\texttt{Op}}$ isn't typeable, then the handler isn't typeable either. Otherwise, c_r has principal type $[\Xi_r^-](B^+\ !\ \Delta^+)$. Every $c_{\texttt{Op}}$ has principal type $[\Xi_{\texttt{Op}}^-](\underline{C}_{\texttt{Op}}^+)$.

The general form of a typing derivation for $\{\texttt{return}\ x \mapsto c_r, [\texttt{Op}\, y\, k \mapsto c_{\texttt{Op}}]_{\texttt{Op} \in \mathcal{O}}\}$ is:

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

SubComp can only be applied if $[\Xi_r^-](B^+ \,!\, \Delta^+) \leqslant^\forall [\Xi, x : A](B \,!\, \Delta)$ and $[\Xi_{\mathtt{Op}}^-](\underline{C}_{\mathtt{Op}}^+) \leqslant^\forall [\Xi, y : A_{\mathtt{Op}}, k : B_{\mathtt{Op}} \to B \,!\, \Delta](B \,!\, \Delta)$. We can write the first subtyping constraint as: $[\Xi_r^- \setminus x, x : \Xi_1^-(x)](B^+ \,!\, \Delta^+) \leqslant^\forall [\Xi, x : A]\underline{C}$. By function subtyping, this has to be equivalent to: $[\Xi_r^- \setminus x](\Xi_r^-(x) \to (B^+ \,!\, \Delta^+)) \leqslant^\forall [\Xi](A \to \underline{C})$. From this, we can conclude that $[\Xi_r^- \setminus x](\Xi_r^-(x) \to (B^+ \,!\, \Delta^+))$ is the principal type of the value case.

Going to the operation cases, the subtyping constraint is $[\Xi_{\mathtt{Op}}^-](\underline{C}_{\mathtt{Op}}^+) \leqslant^\forall [\Xi, y : A_{\mathtt{Op}}, k : B_{\mathtt{Op}} \to B \,!\, \Delta](B \,!\, \Delta)$. By reasoning in the same way, we can get the following: $[\Xi_{\mathtt{Op}}^- \setminus y \setminus k](\Xi_{\mathtt{Op}}^-(y) \to \Xi_{\mathtt{Op}}^-(k) \to \underline{C}_{\mathtt{Op}}^+) \leqslant^\forall [\Xi](A_{\mathtt{Op}} \to (B_{\mathtt{Op}} \to B \,!\, \Delta) \to B \,!\, \Delta)$.

In order to know the principal type of our handler, a few more steps are required. One aspect that we ignored up till now is that, for the operation cases, the right-hand side refers to the type of the handler. Ξ, $B \,!\, \Delta$ and A occur in $[\Xi](A \,!\, \Delta \sqcap \mathcal{O} \Rightarrow B \,!\, \Delta)$. In order to find the type of the handler, we need to use a similar reasoning as for the application. When this is done, we find that $\xi([\Xi_r^- \sqcap (\bigsqcap \Xi_{\mathtt{Op}}^- | \mathtt{Op} \in \mathcal{O})](\Xi_r^-(x) \,!\, \delta \sqcap \mathcal{O} \Rightarrow \alpha \,!\, \delta))$ with ξ solving the constraint set $\{B^+ \,!\, \Delta^+ \leqslant (\alpha \,!\, \delta), [A_{\mathtt{Op}}^+ \leqslant \Xi_{\mathtt{Op}}^-(y), (B_{\mathtt{Op}}^+ \to (\alpha \,!\, \delta)) \leqslant \Xi_{\mathtt{Op}}^-(k), \underline{C}_{\mathtt{Op}}^+ \leqslant (\alpha \,!\, \delta)]_{\mathtt{Op} \in \mathcal{O}}\}$ is the typing scheme of the handler.

We want the handler to explicitly contain the information that the input of the handler may contain the handled effects, hence the occurence of $\sqcap\mathcal{O}$. However, we can give a more formal explanation.

For this, we need to slightly change the typing rule Hand into:

$$\textsc{Hand}\ \frac{\left[\Psi_{\mathtt{Op}} \qquad (\mathtt{Op} : A_{\mathtt{Op}} \to B_{\mathtt{Op}}) \in \Sigma \qquad \Pi \Vdash c_{\mathtt{Op}} : [\Xi, y : A_{\mathtt{Op}}, k : B_{\mathtt{Op}} \to B \,!\, \Delta_2](B \,!\, \Delta_2)\right]_{\mathtt{Op} \in \mathcal{O}} \qquad \Psi_{\Delta_1} \qquad \Pi \Vdash c_r : [\Xi, x : A](B \,!\, \Delta_2)}{\Pi \Vdash \{\mathtt{return}\ x \mapsto c_r, [\mathtt{Op}\ y\ k \mapsto c_{\mathtt{Op}}]_{\mathtt{Op} \in \mathcal{O}}\} : [\Xi](A \,!\, \Delta_1 \Rightarrow B \,!\, \Delta_2)}$$

Ψ_{Δ_1} is used to make sure that any operation in the input dirt Δ_1 that is not handled, is also present in the output Δ_2. We can define Ψ_{Δ_1} as $\Delta_1 \leqslant \Delta_2$ since that subtyping relation encodes that necessary information.

$\Psi_{\mathtt{Op}}$ gives us information about the structure of the handler. When a handler can handle effect Op, we expect the handler type to convey that information. More specifically, we expect the handler type to be $A \,!\, \mathtt{Op} \Rightarrow B \,!\, \Delta_2$. When deriving the principal type, $\Psi_{\mathtt{Op}}$ will turn into a constraint. After the type and dirt variables have been introduced, the constraint takes the form $\Xi_r^-(x) \,!\, \mathtt{Op} \Rightarrow (\alpha \,!\, \delta) \leqslant \Xi_r^-(x) \,!\, \delta_1 \Rightarrow (\alpha \,!\, \delta)$.

This is still a rather complicated constraint which we can manually simplify. By handler subtyping, this constraint decomposes into $(\alpha \,!\, \delta) \leqslant (\alpha \,!\, \delta)$ and $\Xi_r^-(x) \,!\, \delta_1 \leqslant \Xi_r^-(x) \,!\, \mathtt{Op}$. The first constraint is trivial and can be removed. The second constraint can be decomposed further into $\Xi_r^-(x) \leqslant \Xi_r^-(x)$ and $\delta_1 \leqslant \mathtt{Op}$. Again, the first constraint is trivial. In the end, each $\Psi_{\mathtt{Op}}$ creates $\delta_1 \leqslant \mathtt{Op}$.

Both $\Delta_1 \leqslant \Delta_2$ and $\delta_1 \leqslant \mathtt{Op}$ are atomic. We can handle them using the following

constraint solving equations (from Figure 4.4).

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

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

Solving these constraints gives us:

$\xi([\Xi_r^- \sqcap (\bigsqcap \Xi_{\mathtt{Op}}^- | \mathtt{Op} \in \mathcal{O})](\Xi_r^-(x) \,!\, (\boldsymbol{\delta_2} \sqcap (\bigsqcap \mathtt{Op} | \mathtt{Op} \in \boldsymbol{\mathcal{O}})) \Rightarrow \alpha \,!\, \delta_2))$ with ξ solving the constraint set $\{B^+ \,!\, \Delta^+ \leqslant (\alpha \,!\, \delta_2), [A_{\mathtt{Op}}^+ \leqslant \Xi_{\mathtt{Op}}^-(y), (B_{\mathtt{Op}}^+ \to (\alpha \,!\, \delta_2)) \leqslant \Xi_{\mathtt{Op}}^-(k), \underline{C}_{\mathtt{Op}}^+ \leqslant (\alpha \,!\, \delta_2)]_{\mathtt{Op} \in \mathcal{O}}\}$.

$\delta_2 \sqcap (\bigsqcap \mathtt{Op} | \mathtt{Op} \in \mathcal{O})$ is the same as $\delta_2 \sqcap \mathcal{O}$. As an additional note, the constraint set allows the occurence of δ_2 in the output (the positive polarity occurence) to grow to account for effects introduced in the effect clauses themselves.

Handling A with `handle` c `with` v is typed with the With rule from Figure 3.10. If any of v and c are untypeable, then so is the with. If they are all typeable, they respectively have principal types $[\Xi_1^-]A^+$ and $[\Xi_2^-]\underline{C}^+$. By reasoning in the same way as the principal type for the application was found, we find that $\xi([\Xi_1^- \sqcap \Xi_2^-](\alpha \,!\, \delta))$ with ξ solving the constraint set $\{A_1^+ \leqslant (\underline{C}^+ \Rightarrow (\alpha \,!\, \delta))\}$ is the typing scheme of the with.

Return A return `return` v is typed with the Ret rule from Figure 3.10. If v cannot be typed, then `return` v is not typeable either. Otherwise, v has principal typing scheme $[\Xi^-]A^+$. The return takes v and adds an empty dirt. This means that `return` v has typing scheme $[\Xi^-](A^+ \,!\, \emptyset)$. Considering $\emptyset$ is also principal in nature, $[\Xi^-](A^+ \,!\, \emptyset)$ is the principal typing scheme of `return` v.

Operation An operation $\mathtt{Op}\, v$ is typed with the Do rule from Figure 3.10. If v cannot be typed, or the operation Op does not exist, $\mathtt{Op}\, v$ is untypeable. If v is typeable, it has the principal typing scheme $[\Xi^-]A^+$ while an operation Op has type $A^+ \to B^-$. An operation **always has a principal type**. $\mathtt{Op}\, v$ is typed with the resulting type of the operation, with the dirt being the operation. This yields the typing scheme $[\Xi^-](B^+ \,!\, \mathtt{Op})$. All components of this typing scheme are principal.

4.6.3 Principality for Booleans

Here we discuss principality of booleans and conditionals. In Figure 4.7 and Figure 4.8, these are represented by the inference rules True, False and Cond.

Booleans The booleans, `true` and `false` are typed with the True and False rules from Figure 3.10. These rules are facts and give the type *bool* to the values. considering *bool* is already principal, no additional work is required.

Conditionals A conditional $\texttt{if}\ v\ \texttt{then}\ c_1\ \texttt{else}\ c_2$ is typed with the COND rule from Figure 3.10. If any of v, c_1, c_2 are untypeable, then so is the conditional. If they are all typeable, they respectively have principal types $[\Xi_1^-]A^+$, $[\Xi_2^-]\underline{C}_1^+$ and $[\Xi_3^-]\underline{C}_2^+$.

By reasoning in the same way as the principal type for the application was found, we find that $\xi([\Xi_1^- \sqcap \Xi_2^- \sqcap \Xi_3^-](\alpha\ !\ \delta))$ with ξ solving the constraint set $\{A^+ \leqslant bool, \underline{C}_1^+ \leqslant (\alpha\ !\ \delta), \underline{C}_2^+ \leqslant (\alpha\ !\ \delta)\}$ is the typing scheme of the conditional.

4.6.4 Principality of Let-Binding

Finally, we discuss principality of let-bound variables, let-bindings and do-bindings. In Figure 4.7 and Figure 4.8, these are represented by the inference rules VAR-Π, LET and DO.

Let-bound variables A let-bound variable $\hat{\mathbf{x}}$ is typed with the VAR-Π rule from Figure 3.10. If $\hat{\mathbf{x}}$ is not part of Π, then $\hat{\mathbf{x}}$ is not typeable. Otherwise, $\hat{\mathbf{x}}$ is typeable with the principal typing scheme $\Pi(\hat{\mathbf{x}})$

Let-bindings A conditional $\texttt{let}\ \hat{\mathbf{x}} = v\ \texttt{in}\ c$ is typed with the LET rule from Figure 3.10. If v is not typeable, then $\texttt{let}\ \hat{\mathbf{x}} = v\ \texttt{in}\ c$ is not typeable either. Otherwise, v has principal type $[\Xi_1^-]A^+$.

The general form of a typing derivation for $\texttt{let}\ \hat{\mathbf{x}} = v\ \texttt{in}\ c$ is:

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

SUBVAL can only be applied if $[\Xi_1^-]A^+ \leqslant^{\forall} [\Xi_1]A$. With the weakening (WEAKENINGTYPE from Figure 3.9), we get $\Pi, \hat{\mathbf{x}} : [\Xi_1^-]A^+ \Vdash c : [\Xi_2](B\ !\ \Delta)$. With this, we can apply the inductive hypothesis to this case. The principal type of c is $[\Xi_2^-](B^+\ !\ \Delta^+)$. Thus, the principal type of $\texttt{let}\ \hat{\mathbf{x}} = v\ \texttt{in}\ c$ is straightforward: $[\Xi_1^- \sqcap \Xi_2^-](B^+\ !\ \Delta^+)$.

Do-bindings A do-binding $\texttt{do}\ \hat{\mathbf{x}} = c_1\ ;\ c_2$ is typed with the DO rule from Figure 3.10. The do-binding is rather interesting. The reasoning behind finding the principal type is a combination of the conditional and the let-binding. If c_1 is not typeable, then $\texttt{do}\ \hat{\mathbf{x}} = c_1\ ;\ c_2$ is not typeable either. Otherwise, c_1 has principal type $[\Xi_1^-](A^+\ !\ \Delta_1^+)$.

The general form of a typing derivation for $\texttt{do}\ \hat{\mathbf{x}} = c_1\ ;\ c_2$ is:

$$\textsc{Let}\ \frac{\textsc{SubComp}\ \dfrac{\Pi \Vdash c_1 : [\Xi_1^-](A^+\ !\ \Delta_1^+)}{\Pi \Vdash c_1 : [\Xi_1](A\ !\ \Delta)} \quad \textsc{SubComp}\ \dfrac{\Pi, \hat{\mathbf{x}} : [\Xi_1]A \Vdash c_2 : [\Xi_2](B\ !\ \Delta_2)}{\Pi, \hat{\mathbf{x}} : [\Xi_1]A \Vdash c_2 : [\Xi_2](B\ !\ \Delta)}}{\Gamma \Vdash \texttt{do}\ \hat{\mathbf{x}} = c_1\ ;\ c_2 : [\Xi_1 \sqcap \Xi_2](B\ !\ \Delta)}$$

SubComp can only be applied if $[\Xi_1^-](A^+ \,!\, \Delta_1^+) \leqslant^\forall [\Xi_1](A \,!\, \Delta)$. With the weakening (WeakeningType from Figure 3.9), we get $\Pi, \hat{\mathbf{x}} : [\Xi_1^-](A^+ \,!\, \Delta_1^+) \Vdash c : [\Xi_2](B \,!\, \Delta_2)$. With this, we can apply the inductive hypothesis to this case. The principal type of c_2 is $[\Xi_2^-](B^+ \,!\, \Delta_2^+)$.

For the let-binding, we could now find the principal type. However, we still have an issue with the Δ. By reasoning in the same way as the principal type for the application and conditional was found, we find that $\xi([\Xi_1^- \sqcap \Xi_2^-](B^+ \,!\, \delta))$ with ξ solving the constraint set $\{\Delta_1^+ \leqslant \delta, \Delta_2^+ \leqslant \delta\}$ is the typing scheme of the do-binding.

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

Expressions

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

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

$$\textsc{Hand}\ \frac{\Pi \triangleright c_r : [\Xi_r^-](B^+ \,!\, \Delta^+) \qquad \left[(\texttt{Op} : A_{\texttt{Op}}^+ \to B_{\texttt{Op}}^-) \in \Sigma \qquad \Pi \triangleright c_{\texttt{Op}} : [\Xi_{\texttt{Op}}^-](\underline{C}_{\texttt{Op}}^+)\right]_{\texttt{Op} \in \mathcal{O}}}{\begin{array}{c}\Pi \triangleright \{\texttt{return}\ x \mapsto c_r, [\texttt{Op}\ y\ k \mapsto c_{\texttt{Op}}]_{\texttt{Op} \in \mathcal{O}}\} : \\ [\Xi_r^- \sqcap (\bigsqcap \Xi_{\texttt{Op}}^- | \texttt{Op} \in \mathcal{O})](\Xi_r^-(x) \,!\, \delta \sqcap \mathcal{O} \Rightarrow \alpha \,!\, \delta)\end{array}}$$

$$\xi = biunify\left(\begin{array}{l} B^+ \,!\, \Delta^+ \leqslant \alpha \,!\, \delta \\ \left[\begin{array}{l} A_{\texttt{Op}}^+ \leqslant \Xi_{\texttt{Op}}^-(y) \\ B_{\texttt{Op}}^- \to (\alpha \,!\, \delta) \leqslant \Xi_{\texttt{Op}}^-(k) \\ \underline{C}_{\texttt{Op}}^+ \leqslant \alpha \,!\, \delta \end{array}\right]_{\texttt{Op} \in \mathcal{O}} \end{array}\right)$$

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

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

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

$$biunify\begin{pmatrix} A^+ \leqslant bool \\ \underline{C}_1^+ \leqslant (\alpha \,!\, \delta) \\ \underline{C}_2^+ \leqslant (\alpha \,!\, \delta) \end{pmatrix} \qquad \textsc{Ret}\quad \frac{\Pi \triangleright v : [\Xi^-]A^+}{\Pi \triangleright \mathtt{return}\ v : [\Xi^-](A^+ \,!\, \emptyset)}$$

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

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

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

$$biunify\begin{pmatrix} \Delta_1^+ \leqslant \delta \\ \Delta_2^+ \leqslant \delta \end{pmatrix}$$

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

Figure 4.8: Type inference algorithm for computations

Chapter 5

Simplification

EffCore inherits one big disadvantage from algebraic subtyping. Type systems with subtyping produce large types. More specifically, the size of infered types and the constraints grows linearly with the size of the program. Dolan provides a solution to this problem by proposing a simplification algorithm [4]. I have extended this algorithm in order to include algebraic effects and handlers.

The simplification algorithm encodes polar types into finite automata. There is a lot of research about simplifying deterministic finite automata (DFA). Once the automata has been simplified, it is decoded into a polar type. Equivalent types are converted into automata accepting equal languages.

In this chapter, type automata are extended into type-$-effect automata. Afterwards the encoding and decoding algorithms are extended to operate on type-$-effect automata rather than just type automata.

5.1 Type-&-Effect Automata

The regular language of the type-&-effect automata are defined in Figure 5.1. A type-&-effect automaton consists of a finite set of states Q, with a designated start state q_0. Each state is labeled with a set of head constructors and has a polarity. Head constructors represent the types of EffCore, as seen in Figure 3.2, excluding unions, intersections and the recursive type. Recursive types are represented as loops in the type-&-effect automaton. The set of head constructors represent a union or intersection of its elements depending on the polarity of that state. State transitions within a type-&-effect automaton are represented by Σ_F.

Σ_F consists of a function domain d, a function range r, a handler domain dh, a handler range rh, a type splitter t and a dirt splitter e. The domains and ranges are used to encode the domains and ranges of functions and handlers. The splitters are the novel element in type-&-effect automata. They are used to encode dirty types. When a dirty type $\underline{C} = A \,!\, \Delta$ needs to be encoded, a transition t is made to encode A and a transition

e is made to encode the dirt. Type and dirt variables are represented by the symbol V.

Types are encoded as regular expressions that need to be accepted by standard non-deterministic finite automata (NFA). A NFA does not label its states, thus all information is encoded into state transitions as can be seen in the definition of E.

Type-&-Effect Automata

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

Figure 5.1: Type-&-effect automata

5.2 Encoding

A mapping $\mathcal{E}$ is used to encode types as a regular expression. Figure 5.2 shows the main encoding algorithm. The first few cases map the head constructors into a regular expression. We can see how dirty types are encoded by the case $\mathcal{E}^+(\underline{C}) = \mathcal{E}^+(A \,!\, \Delta) = t^+.\mathcal{E}^+(A) + e^+.\mathcal{E}^+(\Delta) + \mathcal{E}^+(\langle \mathsf{drty} \rangle)$. The usage of t and e can be observed. $\langle \mathsf{drty} \rangle$ is used to indicate that that state in the representing NFA represents a dirty type, similar to the usage of $\langle\rightarrow\rangle$. The encoding of handlers follows the encoding of functions. The other rules are the standard rules used in Dolan's thesis. [4] The mapping Ω_α is used to help encode recursive types (Figure 5.3). There are no suprises in this mapping.

The type $\alpha \rightarrow bool \,!\, Op$ is converted into: $d^+.\alpha^- + r^+.(t^+.\langle \mathsf{b} \rangle^+ + e^+.\langle \mathtt{Op} \rangle^+ + \langle \mathsf{drty} \rangle^+) + \langle\rightarrow\rangle^+$.

Theorem 14. *(Equivalence of types) If $A_1^+ \equiv A_2^+$ then $\mathcal{E}^+(A_1^+)$ and $\mathcal{E}^+(A_2^+)$ are the same language and dually for negative types and dirty types.*

The extension made to the encoding algorithm accompasses two changes. The first change is the addition of the handlers. Since handlers behave in exactly the same way as functions, adding handlers cannot invalidate the proof made in Dolan's thesis.

The second change is the addition of dirty types. This change is a bit more tricky. There are two parts to look at. Firstly, let's look at the encoding of dirts. Dirts can be seen as a subset of types. More specifically, dirts are equivalent to types with the exclusion of functions, handlers and recursive types. The basic element $\mathtt{Op}$ behaves like $bool$ and dirt variables are equivalent to type variables. By taking Dolan's original proof, and removing functions, handlers and recursive types, we get a proof for the encoding of dirts.

Finally, the encoding of the dirty type. Taking Dolan's proof, and adding handlers, we know that the encoding of two equivalent types represent the same language (if we were

to ignore effects). We also know that encoding two equivalent dirts results in the same language. The key argument that is still needed is that dirty types always remain dirty types. This can be proven by case analysis.

If we have two equivalent types and one of the types is a function type, then the other type is also a function type. Two function types are equivalent if the domain and ranges are equivalent. We know that the domains (which is pure) are encoded into the same language. For the dirty type, there is only one way to encode the dirty type, which is to use the $\mathcal{E}^+(\underline{C}) = \mathcal{E}^+(A \,!\, \Delta) = t^+.\mathcal{E}^+(A) + e^+.\mathcal{E}^+(\Delta) + \mathcal{E}^+(\langle \mathsf{drty} \rangle)$ case. A similar logic can be used for handler types. Thus, equivalent dirty types are encoded into the same language.

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

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

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

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

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

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

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

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

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

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

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

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

Figure 5.2: Encoding types as type automata

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

$$\Omega^+_\alpha(\alpha) = \epsilon \qquad \Omega^-_\alpha(\alpha) = \epsilon$$

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

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

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

$$\Omega^+_\alpha(A \to \underline{C}) = d^+.\Omega^-_\alpha(A) + r^+.\Omega^+_\alpha(\underline{C}) \qquad \Omega^-_\alpha(A \to \underline{C}) = d^-.\Omega^+_\alpha(A) + r^-.\Omega^-_\alpha(\underline{C})$$

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

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

$$\Omega^+_\alpha(\mathtt{0p}) = \emptyset \qquad \Omega^-_\alpha(\mathtt{0p}) = \emptyset$$

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

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

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

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

Figure 5.3: Encoding recursive types as type automata

5.3 Decoding

Decoding is accomplished by the mapping $\mathcal{R}$, as seen in Figure 5.6. This function acts as the inverse of $\mathcal{E}$. Considering that regular languages do no distinguish between positive and negative types, the mapping $\mathcal{R}$ produces both a positive and negative type. When a regular expression of an automaton representing a positive type A^+ is decoded, $\mathcal{R}$ produces $(A^+, \top)$. Similarly, when a regular expression of an automaton representing a negative type A^- is decoded, $\mathcal{R}$ produces $(\bot, A^-)$.

Symbols representing type constructors are mapped to the greatest or least type according to polarity. Symbols representing fields are mapped to a context, which is a type marking

the use of a particular field by the particular type variable.

Interpreting union, concatenation and the Kleene star is done with the definitions from Figure 5.4 and Figure 5.5. It is interesting to note that there are three ways to solve concatenation: $(A_1^+, A_1^-).(A_2^+, A_2^-)$ and $(\underline{C}_1^+, \underline{C}_1^-).(A_2^+, A_2^-)$, $(\underline{C}_1^+, \underline{C}_1^-).(\underline{C}_2^+, \underline{C}_2^-)$. $(A_1^+, A_1^-).(A_2^+, A_2^-)$ is used for situations where we need to decode $d^+.\alpha^-$. $(\underline{C}_1^+, \underline{C}_1^-).(A_2^+, A_2^-)$ is used for $t^+.\langle \mathsf{b} \rangle^+$. In this case, we want to decode the pure part of a dirty type. Since we need to merge this pure part with the dirty part later, the pure part is already converted into a dirty type. $(\underline{C}_1^+, \underline{C}_1^-).(\underline{C}_2^+, \underline{C}_2^-)$ is used for situations like $e^+.\langle \mathtt{Op} \rangle^+$. These three formulas make sure that the decoded types have the correct form (pure vs dirty).

Theorem 15. $\mathcal{R}(\mathcal{E}^+(A^+)) \equiv (A^+, \top)$ *(and dually for dirty types)*
$\mathcal{R}(\mathcal{E}^-(A^-)) \equiv (\bot, A^-)$ *(and dually for dirty types)*

The proof can be done by induction on the syntax of A. All cases are trivial and straightforward. The complexity arises from recursive types. However, that part of the proof remains equivalent to the original proof.

Theorem 16. *if E_1 and E_2 represent equal regular languages, then $\mathcal{R}(E_1) \equiv \mathcal{R}(E_2)$*

The proof can be made in the same way that the proof for Theorem 14 was made. The extension made to the encoding algorithm accompasses two changes. The first change is the addition of the handlers. Since handlers behave in exactly the same way as functions, adding handlers cannot invalidate the proof made in Dolan's thesis. Secondly, dirts are subsets of types, and the proof holds. Encoding dirty types and decoding dirty types can only occur in one way.

Now that we know that equal languages represent equal types, any algorithm that operates on languages while preserving equality, also preserve equality of the types that are being represented. Any algorithm for simplifying finite automata can be used to simplify type-&-effect automata. A popular option is to convert a NFA that needs to be simplified into a DFA using the subset construction. This DFA can be simplified using any standard algorithm such as Hopcroft's algorithm.

$$(A^+, A^-)^* = \begin{pmatrix} \mu^+\alpha.\mathcal{X} \sqcup [\alpha/\mathcal{X}^+, \mu^-\beta.(\mathcal{X} \sqcap [\alpha/\mathcal{X}^+, \beta/\mathcal{X}^-](A^-))/\mathcal{X}^-]A^+, \\ \mu^-\beta.\mathcal{X} \sqcap [\mu^+\alpha.(\mathcal{X} \sqcup [\alpha/\mathcal{X}^+, \beta/\mathcal{X}^-](A^+))/\mathcal{X}^+, \beta/\mathcal{X}^-]A^- \end{pmatrix}$$

$$(A_1^+, A_1^-) + (A_2^+, A_2^-) = (A_1^+ \sqcup A_2^+, A_1^- \sqcap A_2^-) \text{ (dually for dirty types } \underline{C})$$

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

Figure 5.4: Decoding type automata concat, union and kleene star into types

$$(A_1^+, A_1^-).(A_2^+, A_2^-) = (\bot, \top) \text{ if } (A_2^+, A_2^-) =$$

$$(\bot, \top) \text{ else } \left(\begin{array}{c} [A_2^+/\mathcal{X}^+, A_2^-/\mathcal{X}^-]A_1^+, \\ [A_2^+/\mathcal{X}^+, A_2^-/\mathcal{X}^-]A_1^- \end{array} \right)$$

$$(\underline{C}_1^+, \underline{C}_1^-).(\underline{C}_2^+, \underline{C}_2^-) = (\bot \,!\, \emptyset, \top \,!\, \Omega) \text{ if } (\underline{C}_2^+, \underline{C}_2^-) = (\bot \,!\, \emptyset, \top \,!$$

$$\Omega) \text{ else } \left(\begin{array}{c} [\underline{C}_2^+/\mathcal{X}^+ \,!\, \mathcal{D}^+, \underline{C}_2^-/\mathcal{X}^- \,!\, \mathcal{D}^-]\underline{C}_1^+, \\ [\underline{C}_2^+/\mathcal{X}^+ \,!\, \mathcal{D}^+, \underline{C}_2^-/\mathcal{X}^- \,!\, \mathcal{D}^-]\underline{C}_1^- \end{array} \right)$$

$$(\underline{C}_1^+, \underline{C}_1^-).(A_2^+, A_2^-) = (\bot \,!\, \emptyset, \top \,!\, \Omega) \text{ if } (A_2^+, A_2^-) =$$

$$(\bot, \top) \text{ else } \left(\begin{array}{c} [A_2^+/\mathcal{X}^+ \,!\, \mathcal{D}^+, A_2^-/\mathcal{X}^- \,!\, \mathcal{D}^-]\underline{C}_1^+, \\ [A_2^+/\mathcal{X}^+ \,!\, \mathcal{D}^+, A_2^-/\mathcal{X}^- \,!\, \mathcal{D}^-]\underline{C}_1^- \end{array} \right)$$

Figure 5.5: Decoding type automata concatenation into types

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

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

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

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

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

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

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

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

$$\mathcal{R}(\langle\mathtt{0p}\rangle^+) = (\bot \,!\, \mathtt{0p}, \top \,!\, \Omega) \qquad \mathcal{R}(\langle\mathtt{0p}\rangle^-) = (\bot, \bot \,!\, \mathtt{0p})$$

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

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

Figure 5.6: 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

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

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)
```

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 $ty\_e1 \leqslant ty\_e2 \to drty$ 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 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
2     only 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
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 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

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
```

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 6.6, considers five different testing programs:

1. `Interp`, an arithmetic interpreter that uses algebraic effects to handle division by zero cases.
2. `Loop`, contains 5 different looping programs that executes a loop n times. It is the same program as explained in Chapter 2.3.3.
3. `Parser`, an arithmetic parser that parser and evaluates a list of characters representing an arithmetic equation. Based on modular interpreters from [14].
4. `Queens`, a solver for the n-queens problem. N queens need to be placed on an n by n chessboard in such a way that no queen can attack another queen.
5. `Range`, an implementation of the range function that takes as input a number and returns a list containing numbers ranging from 1 to the input.

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. The numbers are given in percentages. For example, for the `Interp` example, EffCore took just a bit over 2 times as long (222.74%) in order to run compared to the Untyped version.

Overall, the performance of EffCore is superior to the performance of standard subtyping. There is an exception for the `Parser` and `Queens` programs. The implementation of EffCore is very slow compared to the other implementations for these two programs. 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.

Some advantages of algebraic subtyping can be seen with the `Loop` and `Range` programs. Subtyping took considerably more time compared to algebraic subtyping. Overall, I would say that algebraic subtyping is faster than subtyping, if the simplification algorithm is implemented.

	EffCore	Subtyping	Untyped
Interp	222.74	261.47	100
Loop	282.74	471.07	100
Parser	27387.71	1177.23	100
Queens	6402.66	753.47	100
Range	109.26	155.10	100

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 expected to perform worse without the simplification. 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.

The small programs that were used to compare the inferred types are the `Loop` program from the previous section (Chapter 6.5.1). The code from the `Loop` program that is being inferred is given in Figure 6.7. The inferred types for the Looping program can be seen in Figure 6.9, Figure 6.10 and Figure 6.11. It can clearly be seen that, without simplification, inferred types can be huge. When solving the subtyping constraints, bisubstitutions do not remove type or dirt variables, since that variable might still be important. This is a big difference in comparison to regular unification and substitutions within the Hindley-Milner type systems. Due to the equality relation, they can completely remove type and dirt variables.

After simplification, the types become much smaller and much more readable. For loop and test_state, the inferred types for subtyping and for the manually simplified EffCore are very similar. The main difference can be seen for state_handler,

where constraints are not present in the inferred types for manually, simplified EffCore. Looking at the manually, simplified EffCore types, the difference between intersections and unions is visible. Intersections occur in input types (with negative polarity), while unions occur in output types (with positive polarity).

A second example has been constructed in Figure 6.8. This is a simple example that uses an if-statement to decide between two branches. Both branches return a tuple. In EffCore, the inferred (simplified) type is $(\alpha_2 - (\delta_2) \to bool) - (\bot) \to \alpha_2 - (\bot) \to \alpha_1 - (\delta_2) \to (\alpha_1 \sqcup \alpha_2 \times (unit - (Op2 \sqcup Op) \to unit \sqcup int))$. In this type, $(\alpha_1 \sqcup \alpha_2 \times (unit - (Op2 \sqcup Op) \to unit \sqcup int))$ is equivalent to $(\alpha_1 \times (unit - (Op2) \to unit)) \sqcup (\alpha_2 \times (unit - (Op) \to int))$. This code is not typable in standard Eff. This is due to the operations. The first branch returns a tuple containing a function that calls effect Op which returns an int, the other branch returns a tuple containing a function that calls effect $Op2$ which returns an unit. In EffCore, the causes the return type of the function to be $unit \sqcup int$. This type does not "exist" in the current type system, but for extensibility reasons, we want to keep it. We might add a type later that is a $unit \sqcup int$. Standard Eff simply rejects this function.

Figure 6.7: Loop code type inference program

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

Figure 6.8: Simple code type inference program

```
1  effect Op: unit -> int
2  effect Op2: int -> unit
3
4  let select p v d =
5    if (p v) then
6      (v, (fun () -> #Op ()))
7    else
8      (d, (fun () -> #Op2 ()))
```

LOOP
$$int - \{Get, Put\} \to unit$$

STATE_HANDLER
$$\alpha \,!\, \{Get, Put|\delta_1\} \Rightarrow (int - \{Get, Put|\delta_2\} \to int) \,!\, \{Get, Put|\delta_3\}$$
$$|\delta_1 \leqslant \delta_3, \delta_1 \leqslant \delta_2, \delta_1 = \top, \delta_2 = \top, \delta_3 = \top$$

TEST_STATE
$$int - \{Get, Put\} \to int$$

Figure 6.9: Produced types for Subtyping in the Loop program

LOOP
$$(\alpha_1 \sqcap \alpha_2 \sqcap \alpha_3 \sqcap \alpha_4 \sqcap \alpha_5 \sqcap \alpha_6 \sqcap \alpha_7 \sqcap int) - (\delta_1 \sqcup \delta_2 \sqcup \delta_3 \sqcup Get \sqcup Put) \to \alpha_8 \sqcup unit$$

STATE_HANDLER
$$(\alpha_1 \sqcap \alpha_2) \,!\, (\delta_1 \sqcap \delta_2 \sqcap Get \sqcap Put) \Rightarrow (\alpha_3 \sqcup \alpha_4 - (\bot) \to \alpha_4) \sqcup$$
$$(\alpha_5 - (\delta_4 \sqcup \delta_3) \to \alpha_6) \sqcup (\alpha_7 \sqcap \alpha_8 \sqcap int \sqcap int \sqcap \alpha_8 \sqcap int - (\delta_6 \sqcup \delta_5) \to \alpha_9) \,!\, (\delta_1)$$

TEST_STATE
$$(\alpha_1 \sqcap \alpha_2 \sqcap \alpha_3 \sqcap \alpha_4 \sqcap \alpha_5 \sqcap \alpha_6 \sqcap \alpha_7 \sqcap \alpha_8 \sqcap \alpha_9 \sqcap \alpha_{10} \sqcap \alpha_{11}$$
$$\sqcap \alpha_{12} \sqcap \alpha_{13} \sqcap \alpha_{14} \sqcap \alpha_{15} \sqcap \alpha_{16} \sqcap int) - (\delta_2 \sqcup Get \sqcup Put \sqcup \delta_1) \to \alpha_{17} \sqcup int)$$

Figure 6.10: Produced types for EffCore in the Loop program

LOOP
$$int - (Get \sqcup Put) \to unit$$

STATE_HANDLER
$$\alpha_1 \,!\, (Get \sqcap Put) \Rightarrow (int \to int \,!\, (Get \sqcup Put)) \,!\, (\delta_1)$$

TEST_STATE
$$int - (Get \sqcup Put) \to int$$

Figure 6.11: Produced types for EffCore with manual simplification in the Loop program

Chapter 7

Related Work

As explained in Chapter 2.3.4, we were initially looking for solutions to the implicit typing of Eff. Several possible solutions were brievely mentioned. There is row-based typing with an explicitly typed calculus with Hindley-Milner based type inference. The other solution is explicit effect subtyping which uses coercion proofs. In this chapter, we take a closer look at those possible solutions.

Additionally, in his PhD thesis, Stephen Dolan hypothesises how an effect system can be integrated within algebraic subtyping. This chapter compares Dolan's hypothesis with EffCore. This will show that there are differences between the two systems.

7.1 Row-Based Effect Typing

Row-based effect typing is an explicitly typed language with row-based effects. [6] Rather than using subtyping, polymorphism is utilised. Aside from this, the type system for row-based effect typing remains quite similar to the calculus of Eff. There are some small changes, two additional terms are introduced, a type abstraction and a type application, which are used to make the typing explicit. Additionally, a different representation for dirts is used. This can be seen in Figure 7.1.

instead of representing the dirt as a simply set of operations, the dirt is represented as a record. Records look similar to sets, it can contain operations and ends with either a row variable or a closing dot. Type inference is implemented with Hindley-Milner based type inference. Type inference on dirts is implemented like type inference on records.

7.2 Explicit Effect Subtyping

Explicit effect subtyping is an explicitly-typed polymorphic core calculus with support for algebraic effect handlers using a subtyping-based type-&-effect system. It uses coercion proofs in order to make the subtyping constraints explicit in the terms of the calculus. [25]

Terms

value v	$::=$	$\Lambda\alpha.c$	type abstraction
comp c	$::=$	$v\,A$	type application

Types

dirt Δ	$::=$	$\{\mathsf{R}\}$	
R	$::=$	$\mathtt{Op} : R$	row
	$\mid$	δ	row variable
	$\mid$	.	end of row

Figure 7.1: Row-based effect typing

There are 3 different calculi used in explicit effect subtyping. The first calculus ImpEff is implicitly typed and uses implicit effecting. This is elaborated into a second calculus ExEff which is explicitly typed and has explicit effecting. Explicit typing and explicit effecting is accomplished by using coercion proofs. For example, in the SubVal rule a proof γ is explicitly added in order to make typing and effecting explicit, the SubVal rule is written as:

SubVal
$$\frac{\Gamma \vdash v : A \qquad \Gamma \vdash \gamma : A \leqslant B}{\Gamma \vdash v \triangleright \gamma : B}$$

The program `let f = fun g -> g ()` has type $\forall\alpha_1, \alpha_2, \delta_1, \delta_2.\alpha_1 \leqslant \alpha_2 \Rightarrow \delta_1 \leqslant \delta_2 \Rightarrow (unit \to \alpha_1 \,!\, \delta_1) \to \alpha_2 \,!\, \delta_2$. This program has the following elaboration in ExEff: `let f =` $\Lambda\alpha_1.\Lambda\alpha_2.\Lambda\delta_1.\Lambda\delta_2.\Lambda(\omega_1 : \alpha_1 \leqslant \alpha_2).(\omega_2 : \delta_1 \leqslant \delta_2)$. `fun (g` $\hookrightarrow$ `: unit` $\to \alpha_1 \,!\, \delta_1$ `-> g () |>` $\omega_1 \,!\, \omega_2$. This shows how subtyping constraints of both types and effects are made explicit.

The third calculus is an explicitly typed calculus with no effecting called SkelEff. This calculus is used as an intermediate platform to elaborate into languages which do not support algebraic effects and handlers.

7.3 Algebraic Subtyping

In his PhD thesis, Dolan brievely discusses effect systems for algebraic subtyping. He mentions that the algorithms developed in his PhD thesis can also be used for effect systems. [4]

By introducing a new kind ξ of effects and adding them to the definition of function types, an effect system can be implemented. Function types change from their original algebraic notation $\mathcal{T}^{op} \times \mathcal{T}$ into $\mathcal{T}^{op} \times \xi \times \mathcal{T}$.

Adding a new kind has several consequences. Instead of using a single algebra of types $\mathcal{T}$, two algebras are used. One for types $\mathcal{T}$ and one for ξ. This change does not change

the theory that is used for algebraic subtyping. The algebra of ξ has a bottom and top element, as well as a union and intersection. The only point that is still open for interpretation is the basic element of the algebra.

Dolan states that one possible definition for the basic element of ξ is as a set of labels. Other possibilities include defining the basic element of ξ to be a sum type which can refer to other types.

With this information, Dolan gives a plan about how effect systems can be integrated within algebraic subtyping. In contrast to Dolan, rather than starting from an algebraic perspective, I started from a syntactic perspective. I implemented dirts Δ to follow the same syntactic structure as the types. In the end, I ended up with the same algebraic structure for effects as ξ.

The main difference is my choice of the basic element of ξ. I made the choice to have each possible operation be a basic element of ξ. The were several reasons for this choice. From the typing rules, shown in Figure 3.8, only the $\textsc{Op}$ rule introduces effects. Sequencing and handlers utilise the union and intersection types, and the type inference algorithm takes care of the rest. This means that there is no need to use a set of labels as the basic element.

The other big difference between the system introduced in this thesis, $\textsc{EffCore}$ and Dolan's plan, is that this thesis also mapped the algebraic subtyping system onto the calculus of $\textsc{Eff}$, using the distinction between expressions and computations.

The work in this thesis should be seen as complementary to Dolan's PhD thesis, as the results from both theses do not contradict eachother. A specific effect system that utilises Dolan's algebraic subtyping is fully described and formalised in this thesis.

Chapter 8

Conclusion

The main goal of this thesis is to extend algebraic subtyping such that it can operate on algebraic effects and handlers. There is a lot of research going towards extending various type systems to type-&-effect systems in order to give programming language developers more options to utilise algebraic effects and handlers.

Algebraic subtyping offers the advantage and expressivity of subtyping, without the disadvantage of unsolved subtyping constraints. The lack of this disadvantage also extends to effects.

By closely following the approach of algebraic subtyping, a type-&-effect system was constructed that still has the properties of algebraic subtyping extended with semantics for algebraic subtyping for effects. This thesis offers a type-&-effect system that is based on the calculus of Eff. A biunification algorithm, and by extension the bisubstitutions, constraint handling and polar types have been extended to include algebraic effects and handlers. A type inference algorithm, using the biunification algorithm, has been proposed. The simplification of types using type automata has been extended to the simplification of type and effects using type-&-effect automata.

This type-&-effect system, called EffCore, has been implemented in the Eff programming language. The implementation has more features compared to the formalisation. Records, matching and tuples have been implemented. The simplification algorithm has not been included in the final implementation. The implementation has been evaluated using performance testing and comparing inferred types.

The original goals have been achieved. The focus lied on the design of a type-&-effect system derived from Dolan's algebraic subtyping as well as the proving of the properties of this system. The type inference algorithm, the implementation and the emperical evaluation were added to prove the correctness and improve the usability of our work.

We added a type inference algorithm, an implementation and the emperical evaluation into the main scope of this thesis. Which helped me to ground the work as well as deliver a more complete package. Due to this, the proofs are not as rigorous as initially planned.

In his PhD thesis, Dolan focusses heavily on order theory, semirings, Kleene algebra and category theory. Using order theory, semirings, Kleene algebra and category theory to help construct proofs, in addition to including all the optional components would have been too big of a scope. Some proofs in this thesis therefor heavily rely on the proofs made by Dolan in his thesis.

In conclusion, EffCore and an implementation have been delivered. I have shown that algebraic subtyping can indeed be extended to include algebraic effects and handlers. In addition, EffCore and the implementation provide a platform for further research.

8.1 Future Work

Even though this thesis has come to a close, there are still opportunities for future work. Some of the following aspects can be considered low hanging fruit, while others build on top of and go beyond the research in this thesis.

8.1.1 Simplification algorithm implementation

As mentioned in Chapter 6, the simplification algorithm has not been implemented in the Eff programming language. Implementing the simplification algorithm would yield interesting results in the emperical evaluation. One aspect of the emperical evaluation is the performance benchmarking of the algebraic subtyping approach compared to subtyping. Some benchmarks, the `Parser` and Queens programs, performed badly when compared to subtyping. I made the hypothesis that this is due to the lack of simplification that causes an internal build-up of type variables. By implementing the simplification algorithm, this hypothesis can be tested.

8.1.2 Biunification with Type Automata

Extending on the previous point, there is one aspect of the simplification algorithm that has not been extended to algebraic effects and handlers. This is biunification with type automata. In his thesis, Dolan uses the type automata to implement biunification, replacing the standard biunification algorithm. This algorithm is straightfoward to extend in order to use type-&-effect automata. Yet again, comparing the performance of this biunification algorithm compared to the current version could yield interesting results.

8.1.3 Optimization

Originally, as stated in Subsection 2.3.4, we were looking for type-&-effect system that could efficiently be used in the development of an optimising compiler. Algebraic subtyping, and by extension, EffCore, is partly such a type-&-effect system. An optimising compiler for EffCore does not have to deal with subtyping constraints anymore, because they get solved during the type inference using biunification. This opens the door for better optimization techniques using EffCore.

Appendices

Appendix A

Proof of Instantiation

Proof of Theorem 1. We prove the theorem by induction on derivations, which is straightforward for most of the cases.

- SubVal

  If $\dfrac{\Gamma \vdash v : A \qquad A \leqslant B}{\Gamma \vdash v : B}\ \textsc{SubVal}$ then $\dfrac{\rho(\Gamma) \vdash v : \rho(A) \qquad \rho(A \leqslant B)}{\rho(\Gamma) \vdash v : \rho(B)}\ \textsc{SubVal}$

  This is straightforward since: $\rho(A \leqslant B) \equiv \rho(A) \leqslant \rho(B)$, in which case the statement is valid.

- True

  If $\dfrac{}{\Gamma \vdash \texttt{true} : bool}\ \textsc{True}$ then $\dfrac{}{\rho(\Gamma) \vdash \texttt{true} : \rho(bool)}\ \textsc{True}$

  This is also a straightforward case. $\rho(bool) \equiv bool$, thus the case is trivially true.

- False

  If $\dfrac{}{\Gamma \vdash \texttt{false} : bool}\ \textsc{False}$ then $\dfrac{}{\rho(\Gamma) \vdash \texttt{false} : \rho(bool)}\ \textsc{False}$

  This is also a straightforward case. $\rho(bool) \equiv bool$, thus the case is trivially true.

- Var-λ

  If $\dfrac{(x : A) \in \Gamma}{\Gamma \vdash x : A}\ \textsc{Var-}\lambda$ then $\dfrac{(x : \rho(A)) \in \rho(\Gamma)}{\Gamma \vdash x : \rho(A)}\ \textsc{Var-}\lambda$

  This is another straightforward case. If $(x : A) \in \Gamma$ then $(x : \rho(A)) \in \rho(\Gamma)$ is trivially true since ρ only substitutes type and dirt variables and does not change types.

- Var-∀

  If $\dfrac{\text{Var-}\forall \quad (\hat{\mathbf{x}} : \forall \bar{\alpha}.A) \in \Gamma}{\Gamma \vdash \hat{\mathbf{x}} : A[\bar{A}/\bar{\alpha}]}$ then $\dfrac{\text{Var-}\forall \quad (\hat{\mathbf{x}} : \forall \bar{\alpha}.\rho'(A)) \in \rho(\Gamma)}{\rho(\Gamma) \vdash \hat{\mathbf{x}} : \rho'(A)[\rho(\bar{A})/\bar{\alpha}]}$

  This case is nontrivial. This typing rule is not changed in EffCore compared to algebraic subtyping. Thus, the same reasoning for algebraic subtyping applies here [4]. Since ρ and ρ' perform the same substitution, excluding $\bar{\alpha}$, $\rho'(A)[\rho(\bar{A})/\bar{\alpha}] \equiv \rho(A[\bar{A}/\bar{\alpha}])$.

- Fun

  If $\dfrac{\text{Fun} \quad \Gamma, x : A \vdash c : \underline{C}}{\Gamma \vdash \lambda x.c : A \to \underline{C}}$ then $\dfrac{\text{Fun} \quad \rho(\Gamma, x : A) \vdash c : \rho(\underline{C})}{\rho(\Gamma) \vdash \lambda x.c : \rho(A \to \underline{C})}$

  This is another trivial case. $\rho(\Gamma, x : A) \equiv \rho(\Gamma), x : \rho(A)$ and $\rho(A \to \underline{C}) \equiv \rho(A) \to \rho(\underline{C})$.

- Hand

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

  $$\text{then } \frac{\text{Hand} \quad \rho(\Gamma, x : A) \vdash c_r : \rho(B \,!\, \Delta) \qquad \Big[(\mathtt{Op} : \rho(A_{\mathtt{Op}} \to B_{\mathtt{Op}})) \in \Sigma \quad \Gamma, y : \rho(A_{\mathtt{Op}}), k : \rho(B_{\mathtt{Op}}) \to \rho(B \,!\, \Delta) \vdash c_{\mathtt{Op}} : \rho(B \,!\, \Delta)\Big]_{\mathtt{Op} \in \mathcal{O}}}{\rho(\Gamma) \vdash \{\mathtt{return}\ x \mapsto c_r, [\mathtt{Op}\, y\, k \mapsto c_{\mathtt{Op}}]_{\mathtt{Op} \in \mathcal{O}}\} : \qquad \rho(A \,!\, \Delta \sqcap \mathcal{O} \Rightarrow B \,!\, \Delta)}$$

  This case looks intimidating at first, due to the typing of the handler. However, it is another trivial case. This case plays out in exactly the same way as functions. $\rho(\Gamma, x : A) \equiv \rho(\Gamma), x : \rho(A)$ and $\rho(A \,!\, \Delta \sqcap \mathcal{O} \Rightarrow B \,!\, \Delta) \equiv \rho(A \,!\, \Delta \sqcap \mathcal{O}) \Rightarrow \rho(B \,!\, \Delta)$.

□

Proof of Theorem 2. We prove the theorem by induction on derivations, which is straightforward for most of the cases.

- SubComp

  If $\dfrac{\text{SubComp} \quad \Gamma \vdash c : \underline{C} \qquad \underline{C} \leqslant \underline{D}}{\Gamma \vdash c : \underline{D}}$ then $\dfrac{\text{SubVal} \quad \rho(\Gamma) \vdash c : \rho(\underline{C}) \qquad \rho(\underline{C} \leqslant \underline{D})}{\rho(\Gamma) \vdash c : \rho(\underline{D})}$

This is straightforward since: $\rho(\underline{C} \leqslant \underline{D}) \equiv \rho(\underline{C}) \leqslant \rho(\underline{D})$, in which case the statement is valid.

- SubComp

$$\text{If } \frac{\Gamma \vdash v_1 : A \to \underline{C} \qquad \Gamma \vdash v_2 : A}{\Gamma \vdash v_1\, v_2 : \underline{C}}\ (\textsc{App}) \text{ then } \frac{\rho(\Gamma) \vdash v_1 : \rho(A \to \underline{C}) \qquad \rho(\Gamma) \vdash v_2 : \rho(A)}{\rho(\Gamma) \vdash v_1\, v_2 : \rho(\underline{C})}\ (\textsc{App})$$

This is straightforward since: $\rho(A \to \underline{C}) \equiv \rho(A) \to \rho(\underline{C})$, in which case the statement is valid.

- Cond

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

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

This case is trivially valid.

- Ret

$$\text{If } \frac{\Gamma \vdash v : A}{\Gamma \vdash \texttt{return}\ v : A\ !\ \emptyset}\ (\textsc{Ret}) \text{ then } \frac{\rho(\Gamma) \vdash v : \rho(A)}{\rho(\Gamma) \vdash v : \rho(A\ !\ \emptyset)}\ (\textsc{Ret})$$

This case is trivially valid. In this case, you can see the substition being applied to a dirty type. $\rho(A\ !\ \emptyset) \equiv \rho(A)\ !\ \emptyset$.

- Op

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

This case is trivially valid. It is very similar to the Ret case.

- Let

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

$$\frac{\rho(\Gamma) \vdash v : \rho_2(A) \qquad \rho(\Gamma), \hat{\mathbf{x}} : \rho_2(\forall \bar{\alpha}.A) \vdash c : \rho(B\ !\ \Delta) \qquad \alpha \notin FTV(\Gamma)}{\rho(\Gamma) \vdash \texttt{let}\ \hat{\mathbf{x}} = v\ \texttt{in}\ c : \rho(B\ !\ \Delta)}\ (\textsc{Let})$$

This case is nontrivial. This typing rule is not changed in EffCore compared to algebraic subtyping. Thus, the same reasoning for algebraic subtyping applies here [4]. The tricky part of this rule is $\alpha \notin FTV(\Gamma)$. In order to deal with this, we use ρ_2 which maps α to a β which is not free in $\rho(\Gamma)$ and behaves exactly like ρ otherwise. Due to $\alpha \notin FTV(\Gamma)$, $\rho \equiv \rho_2$. We can construct a ρ' like in case

Var-$\forall$ so that $\rho(\hat{\mathbf{x}}) = \forall\bar{\alpha}.\rho'(A)$. $\forall\bar{\alpha}.\rho'(A)$ is alpha-equivalent to $\forall\bar{\beta}.\rho_2(A)$, thus $\rho(\Gamma), \hat{\mathbf{x}} : \rho_2(\forall\bar{\alpha}.A) \equiv \rho(\Gamma, \hat{\mathbf{x}} : \forall\bar{\alpha}.A)$ according to proposition 1.

- Do

  If $\dfrac{\text{Do}\quad \Gamma \vdash c_1 : A \,!\, \Delta \qquad \Gamma, \hat{\mathbf{x}} : \forall\bar{\alpha}.A \vdash c_2 : B \,!\, \Delta \qquad \alpha \notin FTV(\Gamma)}{\Gamma \vdash \texttt{do}\ \hat{\mathbf{x}} = c_1\ ;\ c_2 : B \,!\, \Delta}$ then

  $\dfrac{\text{Do}\quad \rho(\Gamma) \vdash v : \rho_2(A \,!\, \Delta) \qquad \rho(\Gamma), \hat{\mathbf{x}} : \rho_2(\forall\bar{\alpha}.A) \vdash c : \rho(B \,!\, \Delta) \qquad \alpha \notin FTV(\Gamma)}{\rho(\Gamma) \vdash \texttt{let}\ \hat{\mathbf{x}} = v\ \texttt{in}\ c : \rho(B \,!\, \Delta)}$

  This case is nontrivial. This typing rule is not changed in EffCore compared to algebraic subtyping. Thus, the same reasoning for algebraic subtyping applies here [4]. The reasoning for this case is identical to the Let case.

- With

  If $\dfrac{\text{With}\quad \Gamma \vdash v : \underline{C} \Rightarrow \underline{D} \qquad \Gamma \vdash c : \underline{C}}{\Gamma \vdash \texttt{handle}\ c\ \texttt{with}\ v : \underline{D}}$ then $\dfrac{\text{With}\quad \rho(\Gamma) \vdash v : \rho(\underline{C} \Rightarrow \underline{D}) \qquad \rho(\Gamma) \vdash c : \rho(\underline{C})}{\rho(\Gamma) \vdash \texttt{handle}\ c\ \texttt{with}\ v : \rho(\underline{D})}$

  This case is a straightforward case. $\rho(\underline{C} \Rightarrow \underline{D}) \equiv \rho(C) \Rightarrow \rho(\underline{D})$, which makes the case trivial.

□

Appendix B

Proof of Soundness

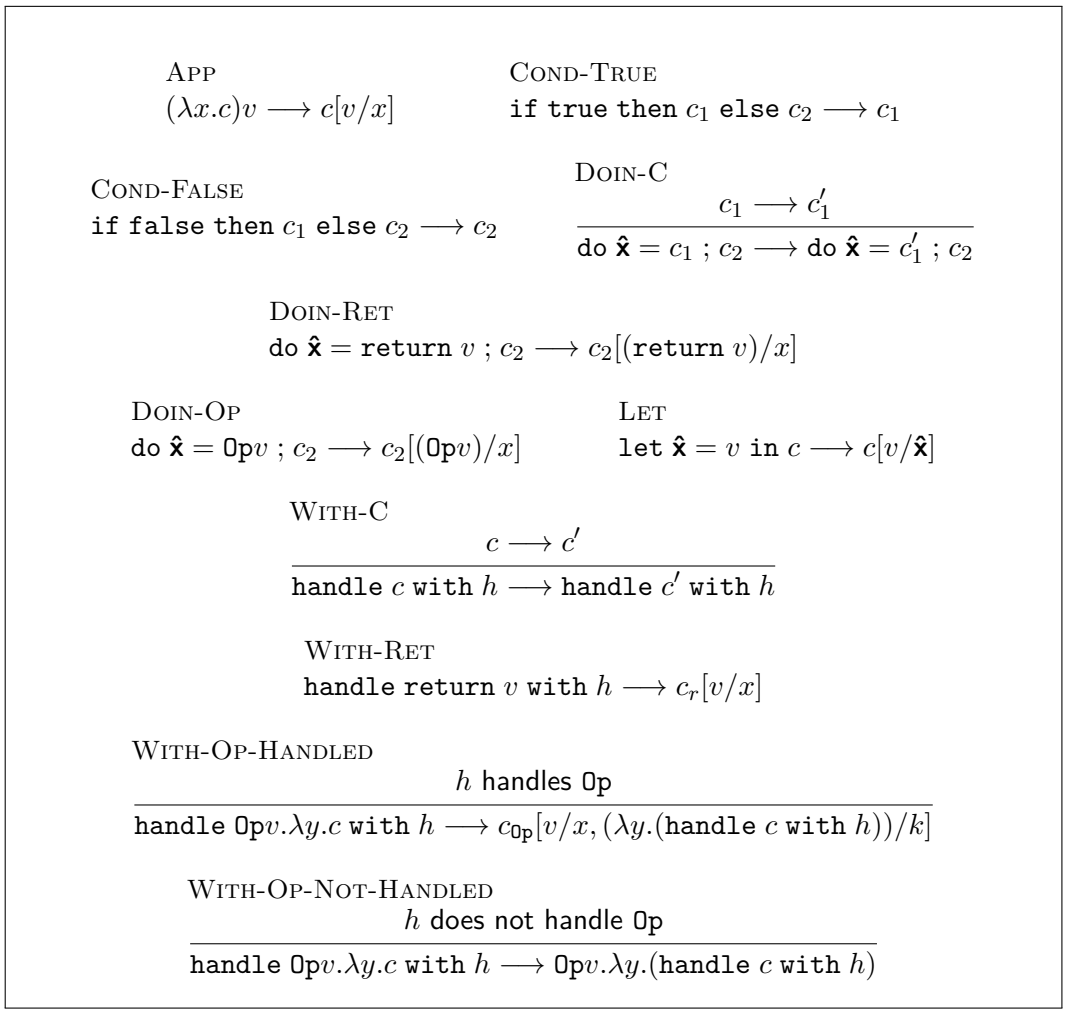

Figure B.1: Small-step transition relation

Theorem 17. *(Value inversion)*
If $\vdash v : A \to \underline{C}$ *then* $v = \lambda x.$
If $\vdash v : \underline{C} \Rightarrow \underline{D}$ *then* $v = \{\texttt{return}\ x \mapsto c_r, [\texttt{Op}\ y\ k \mapsto c_{\texttt{Op}}]_{\texttt{Op} \in \mathcal{O}}\}$
If $\vdash v : \mathit{bool}$ *then* $v \in \{\texttt{true}, \texttt{false}\}$

The proof for this can be fully extended from the proof giving in Dolan's thesis. Values can only be typed with specific rules. More specifically, only False, True, Fun and Hand type values. If none of these can be used, the SubVal rule needs to be called. This means that a subtyping relationship between a function and a boolean, handler and boolean or function and handler needs to exist. This is prohibited according to Chapter 3.2. Thus, the rules give in the theorem are the only possible options.

Theorem 18. *(Progress) If* $\vdash c : A$ *then* c *is either a* $\texttt{return}\ v$*, a* $\texttt{Op}v$ *or* $c \longrightarrow c'$ *for some* c'

We start with proving Theorem 10. This is done by induction on the typing derivation of c. The proof explicitly uses computations. This means that several trivial cases, the values, are already handled. Every value case returns a value. This means we only need to look at App, Cond, Ret, Op, Let, Do and With.

The proof for the App, Cond and Let are similar to the proof made by Dolan, the others are novel proofs.

App
$$\frac{\Gamma \vdash v_1 : A \to \underline{C} \qquad \Gamma \vdash v_2 : A}{\Gamma \vdash v_1\ v_2 : \underline{C}}$$

The application states that v_1 and v_2 are already values. v_1 also need to have the function type and thus v_1 is a function (or a variable pointing to a function) by the value inversion Theorem 17. This means that we can progress $v_1\ v_2$ using the small-step transition relation $(\lambda x.c)v \longrightarrow c[v/x]$.

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

v is already a value. By the value inversion Theorem 17, v is either `true` or `false`. This means that we can progress the conditional using one of the small-step transition relations.

Ret
$$\frac{\Gamma \vdash v : A}{\Gamma \vdash \texttt{return}\ v : A\ !\ \emptyset}$$

`return` v is a trivial case, since it means the evaluation has ended.

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

$\mathtt{Op}\, v$ is a trivial case, since it means the evaluation has ended.

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

v is already a value. Thus we can progress using the small-step transition relation $\mathtt{let}\ \hat{\mathbf{x}} = v\ \mathtt{in}\ c \longrightarrow c[v/\hat{\mathbf{x}}]$.

DO
$$\frac{\Gamma \vdash c_1 : A \,!\, \Delta \qquad \Gamma, \hat{\mathbf{x}} : \forall \bar{\alpha}.A \vdash c_2 : B \,!\, \Delta \qquad \alpha \notin FTV(\Gamma)}{\Gamma \vdash \mathtt{do}\ \hat{\mathbf{x}} = c_1\ ;\ c_2 : B \,!\, \Delta}$$

c_1 is not a value, unlike in the previous case. By the induction hypothesis, there are three possibilities. Either c_1 is a $\mathtt{return}\ v$ or $\mathtt{Op}v$, in which case we can progress. Otherwise, we need to be able to progress c_1 in order to progress the $\mathtt{do}\ \hat{\mathbf{x}} = c_1\ ;\ c_2$.

WITH
$$\frac{\Gamma \vdash v : \underline{C} \Rightarrow \underline{D} \qquad \Gamma \vdash c : \underline{C}}{\Gamma \vdash \mathtt{handle}\ c\ \mathtt{with}\ v : \underline{D}}$$

Finally, there is the $handle$. v is already a value and is, by value inversion, a handler. There are four possibilities by the induction hypothesis in order to progress. If c is a $\mathtt{return}\ v$, we can progress by stepping into the value case. If c is a $\mathtt{Op}v$, then we need to check whether v handles $\mathtt{Op}$ or not. In either case we can progress. If $\mathtt{Op}$ is handled, we can call the matching effect clause. A hidden function $\lambda y.c$ is used to denote the continuation. This is expanded into $\lambda y.(\mathtt{handle}\ c\ \mathtt{with}\ h)$ such that other operations within c can be handled. If $\mathtt{Op}$ is not handled, we return the operation and change the continuation into $\lambda y.(\mathtt{handle}\ c\ \mathtt{with}\ h)$ in order to ensure that, should the operation be handled later, the value case of the handler is still called. If c is not $\mathtt{return}\ v$ and $\mathtt{Op}v$, then in order to progress, we need to be able to progress c.

Theorem 19. *(Preservation) If* $\vdash c : \underline{C}$ *and* $c \longrightarrow c'$ *then* $\vdash c' : \underline{C}$

In order to proof Theorem 11, induction on the small-step relation $c \longrightarrow c'$ is required. These proofs are made using derivation trees. The proof for the APP, COND-TRUE and COND-FALSE and LET are equivalent to the proof made by Dolan. The other rules are equivalent to the operational semantics from EFF [21]. Since these rules are inherited from EFF and algebraic subtyping, and no other changes have occured, the proof still holds.

Appendix C

Equivalence of original and reformulated typing rules

In order to show the equivalence between the original and reforumulated typing rules, we require a means of converting standard typing contexts into polymorphic typing contexts. This is done using two functions p and m as seen in Figure C.1. These two functions are equivalent to their algebraic subtyping counterparts.

$$m(\epsilon) = \epsilon \qquad p(\epsilon) = \epsilon$$

$$m(\Gamma, x : A) = m(\Gamma), x : A \qquad p(\Gamma, x : A) = p(\Gamma)$$

$$m(\Gamma, \hat{\mathbf{x}} : \forall \bar{\alpha}.A) = m(\Gamma) \qquad p(\Gamma, \hat{\mathbf{x}} : \forall \bar{\alpha}.A) = p(\Gamma), \hat{\mathbf{x}} : [m(\Gamma)]A \quad (\alpha \text{ not free in } \Gamma)$$

$$r(\epsilon) = \epsilon$$

$$r(\Pi, \hat{\mathbf{x}} : [\Xi]A) = (r(\Pi) \sqcap \Xi), \hat{\mathbf{x}} : \forall \bar{\alpha}.A$$

Figure C.1: Conversion function for typing contexts

Theorem 20. *If* $\Gamma \vdash v : A$ *then* $p(\Gamma) \vdash v : [m(\Gamma)]A$
If $\Gamma \vdash c : \underline{C}$ *then* $p(\Gamma) \vdash c : [m(\Gamma)]\underline{C}$

The proof for this case is a straighforward derivation on the typing rules. Most cases are trivial since they do not change the typing context. The exceptions are VAR-$\forall$, VAR-λ, FUN, HAND, SUBVAL, SUBCOMP, LET, and DO. Most of these proofs remain unchanged from Dolan's proof. SUBVAL and SUBCOMP are similar rules operating on pure and dirty types, thus the proof for a regular SUB rule applies. The only new

rule is Hand. The same reasoning for the Fun rule applies to the handler. Since $p(\Gamma, x : A) = p(\Gamma)$ and $m(\Gamma, x : A) = m(\Gamma), x : A$ and analogous for the variables y and k in the effect clauses.

Theorem 21. *If* $\Pi \vdash v : [\Xi]A$ *then* $r(\Pi) \sqcap \Xi \vdash v : A$
If $\Pi \vdash c : [\Xi]\underline{C}$ *then* $r(\Pi) \sqcap \Xi \vdash c : \underline{C}$

The proof for this case is a straighforward derivation on the typing rules. Most cases are trivial since they do not change the typing context. The exceptions are Var-Ξ, Var-Π, Fun, Hand, SubVal, SubComp, Let, and Do. Most of these proofs remain unchanged from Dolan's proof. Again, we can apply the reasoning from Fun to Hand. By the induction hypothesis, $r(\Pi) \sqcap (\Xi, x : A) \vdash c_r : B \mid \Delta$ which is the result. An analogous reasoning can be used for the effect clauses.

Theorem 22. *If* $\vdash v : A$ *then* $\vdash v : []A$
If $\vdash c : \underline{C}$ *then* $\vdash c : []\underline{C}$

Using Theorem 20 and Theorem 21, for the two directions, this theorem holds.

Appendix D

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

[7] 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.

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

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

[10] 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.

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

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

[13] 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.

[14] S. Liang, P. Hudak, and M. Jones. Monad transformers and modular interpreters. In *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, pages 333–343, New York, NY, USA, 1995. 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.

Fiche masterproef

Student: Axel Faes

Titel: Algebraic Subtyping for Algebraic Effects and Handlers

UDC: 681.3

Korte inhoud:

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). This type-&-effect system has been implemented in the Eff programming language in order to provide a proof-of-concept.

Thesis voorgedragen tot het behalen van de graad van Master of Science in de ingenieurswetenschappen: computerwetenschappen, hoofdoptie Artificiële intelligentie

Promotor: Prof. dr. ir. Tom Schrijvers

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

Begeleider: Amr Hany Shehata Saleh