KU LEUVEN

FACULTEIT INGENIEURSWETENSCHAPPEN

Algebraic Subtyping for Algebraic Types and Effects

Axel Faes

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

Thesis supervisor:
Prof. dr. ir. Tom Schrijvers

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

Mentor:
Amr Hany Shehata Saleh

Academic year 2017 – 2018

Preface

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

Axel Faes

Contents

Abstract

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

List of Figures

List of Tables

Chapter 1

Introduction

The specification for a type-&-effect system with algebraic subtyping for algebraic effects and handlers is given in this document. The formal properties of this system are studied in order to find which properties are satisfied compared to other type-&-effect systems. The proposed type-&-effect system builds on two very recent developments in the area of programming language theory.

Algebraic subtyping

In his December 2016 PhD thesis, Stephen Dolan (University of Cambridge, UK), has presented a novel type system that combines subtyping and parametric polymorphism in a particulary attractive and elegant fashion. A cornerstone of his design are the algebraic properties that the subtyping relation should respect.

Algebraic effects and handlers

These 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). This approach is gaining a lot of traction, not only as a formalism but also as a practical feature in actual programming languages (e.g. the Koka language developed by Microsoft Research). We are collaborating with Matija Pretnar on the efficient implementation of one such language, called Eff. Axel Faes has contributed to this collaboration during a project he did for the Honoursprogramme of the Faculty of Engineering Science.

1.1 Motivation

Algebraic effects and handlers benefit from a custom type-&-effect system, a type system that also tracks which effects can happen in a program. Several such type-&-effect systems have been proposed in the literature, but all are unsatisfactory. We attribute this to the lack of the elegant properties of Dolan's type system. Indeed the existing type-&-effect systems are not only theoretically unsatisfactory, but they are also awkward to implement and use in practice.

Research questions

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

1.2 Goals

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

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

1.3 Results

Describe what the resulting product is and how it is useful or provides an advantage over other solutions.

Chapter 2

Simply Typed Lambda Calculus

2.1 Programming language theory

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

The work described in this thesis uses several aspects from programming language theory. An important subdiscipline that is extensively used is type theory. Type theory is used to formaly describe type systems. A type system is a set of rules that are used to define the shape of meaningful programs. The *simply typed lambda calculus* will be used to show and explain the necessary background that is required for further chapters. The *simply typed lambda calculus* is the simplest and most elementary form of all typed languages. [9]

2.2 Types and terms

Terms

Figure 2.1 shows the three sorts of terms of the *simply typed lambda calculus*. A variable by itself is already a term. 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. [5]

Types

Since the *simply typed lambda calculus* is being used, types are needed. As seen in figure 2.2, there are two types, the base type and the function type. 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.

$$\begin{array}{rcll} \text{terms } t & ::= & x & \text{variable} \\ & | & \texttt{true} & \text{true} \\ & | & \texttt{false} & \text{false} \\ & | & \lambda x : T.t & \text{function} \\ & | & t_1\, t_2 & \text{application} \end{array}$$

Figure 2.1: Terms of simply typed lambda calculus

$$\begin{array}{rcll} \text{type } T & ::= & bool & \text{base type} \\ & | & T \rightarrow T & \text{function type} \end{array}$$

Figure 2.2: Types of simply typed lambda calculus

2.3 Typing rules

As stated above, a method is needed to place constraints on the programming language. This is done with typing rules or types judgements. The typing rules for the *simply typed lambda calculus* are given in figure 2.3.

$$\text{typing contexts } \Gamma \quad ::= \quad \epsilon \;|\; \Gamma, x : T$$

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

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

Figure 2.3: Typing of simply typed lambda calculus

The first rules to take note of are the True and False rules. These are facts and 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 $T_1 \rightarrow T_2$ and T_1, then the application $t_1\, t_2$ will have the type T_2.

An inference rule can be read in multiple ways. 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. [9]

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.

2.4 Other extensions

Now, a full specification of the *simply typed lambda calculus* is given. However, there are many extensions that can be added onto this language. In the next chapter, Eff will be discussed. Eff is a language which can be described as a modification of the *simply typed lambda calculus* with algebraic effects and handlers. Eff is also uses subtyping rules, this concept will also be further explained in the next chapter. After this, algebraic subtyping will be added to the language.

Of course, just a specification does not have much meaning. Certain aspects or properties could be proved in order to show that they do (or do not) hold in the given language. 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.

Chapter 3

Algebraic effects and handlers (Eff)

Algebraic effect handling is a very active area of research. Implementations of algebraic effect handlers are becoming available and the theory is actively being developed. The type-&-effect system that is used in EFF is based on subtyping and algebraic effect handlers [1]. The *simply typed lambda calculus* is used as a basis for EFF. Let us start with a simple example in order to show what algebraic effects and handlers are. With this example, the differences with the *simply typed lambda calculus* can also be shown.

In the example below, 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 it leaves behind after being handled.

```
effect DivisionByZero : unit -> empty;;

let divide a b =
  if (b == 0) then
    #DivisionByZero ()
  else
    a / b;;

let safeDivide a b =
  handle (divide a b) with
    | #DivisionByZero () k -> 0;;
```

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 and are dirty, while terms that cannot contain effects are called expressions and are pure. 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.

To reiterate, in order to extend *simply typed lambda calculus* to Eff, several terms need to be added. A term is required in order to call effects and handle effects. Of course, we need to be able to have handlers as well. [12]

3.1 Types and terms

Terms

Figure 3.1 shows the two kinds of terms in Eff. As explained before, there are values v and computations c. Computations are terms that can contain effects. Effects are denoted as operations Op which can be called. [10]

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]

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

Figure 3.1: Terms of Eff

Types

Figure 3.2 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 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 $\underline{C}$, handles the effects in this computation and outputs computation $\underline{D}$ as the result [10]. 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	$::=$	$\mathtt{bool}$	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	$::=$	$\{\mathtt{Op}_1, \ldots, \mathtt{Op}_n\}$	

Figure 3.2: Types of Eff

3.2 Subtyping

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 $A \,!\, \Delta \leq A \,!\, \Delta'$ on dirty types. As dirty types can occur inside pure types, we also get a derived subtyping judgement on pure types. Both judgements are defined in Figure 3.3. Observe that, as usual, subtyping is contravariant in the argument types of functions and handlers, and covariant in their return types. [11]

Subtyping

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

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

Figure 3.3: Subtyping for pure and dirty types of Eff

3.3 Typing rules

Figure 3.4 defines the typing judgements for values and computations with respect to a standard typing context Γ.

Values

The rules for subtyping, variables, and functions are entirely standard. For constants we assume a signature Σ that assigns a type A to each constant $\mathtt{k}$, which we write as

$(\mathtt{k} : A) \in \Sigma$.

A handler expression has type $A \,!\, \Delta \cup O \Rightarrow B \,!\, \Delta$ iff all branches (both the operation cases and the return case) have dirty type $B \,!\, \Delta$ and the operation cases cover the set of operations O. Note that the intersection $\Delta \cap O$ is not necessarily empty. The handler deals with the operations O, but in the process may re-issue some of them (i.e., $\Delta \cap O$).

When typing operation cases, the given signature for the operation $(\mathtt{Op} : A_{\mathtt{Op}} \to B_{\mathtt{Op}}) \in \Sigma$ determines the type $A_{\mathtt{Op}}$ of the parameter x and the domain $B_{\mathtt{Op}}$ of the continuation k. As our handlers are deep, the codomain of k should be the same as the type $B \,!\, \Delta$ of the cases. [11]

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 $\underline{D}$. [11]

$$\text{typing contexts } \Gamma \ ::= \ \epsilon \ | \ \Gamma, x : A$$

Expressions

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

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

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

Computations

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

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

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

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

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

Figure 3.4: Typing of Eff

Chapter 4

Core Language (EffCore)

EffCore is a specialization of Eff. The subtyping system from Eff is replaced with algebraic subtyping [3].

4.1 Types and terms

Terms

Figure 4.1 shows the two types of terms in EffCore. Just like in Eff. there are values v and computations c. Computations are terms that can contain effects. Effects are denoted as operations Op which can be called. The only 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 [3]. By making this distinction, a distinction can be made between monomorphic variables (lambda-bound) and polymorphic variables (let-bound) at the term level.

Types

Figure 4.2 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 finite set of operations Δ, which we call dirt, that can be called. It can also be an union or intersection of dirty types. In further sections, the relations between dirty intersections or unions and pure intersections or unions are explained. The finite set Δ is an over-approximation of the operations that are actually called. Row variables are introduced as well as intersection and unions. The .(DOT) is used to close rows that do not end with a row variable. 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. [10]

However, now the effects of the algebraic subtyping approach become apparant. Different types are added in order to support the subtyping. These are a type variables, recursive type, top, bottom, intersection and union [3]. The novel element here is the combination of the algebraic effects and algebraic subtyping. There needs to be a way

value v	$::=$	x	λ-variable
	$\mid$	$\hat{\mathtt{x}}$	let-variable
	$\mid$	`true`	true
	$\mid$	`false`	false
	$\mid$	$\lambda x.c$	function
	$\mid$	$\{$	handler
		$\mathtt{return}\ x \mapsto c_r,$	return case
		$[\mathtt{Op}\ y\ k \mapsto c_{\mathtt{Op}}]_{\mathtt{Op} \in O}$	operation cases
		$\}$	
comp c	$::=$	$v_1\ v_2$	application
	$\mid$	$\mathtt{do}\ \hat{\mathtt{x}} = c_1\ ;\ c_2$	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 4.1: Terms of EffCore

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.

4.2 Type system

TODO: explain equivalance rules

4.3 Typing rules

Figure 4.7 defines the typing judgements for values and computations with respect to a standard typing context Γ.

Values

TODO: explain typing judgements: values

Computations

TODO: explain typing judgements: computations

4.4 Reformulated typing rules

TODO: explain reformulated typing judgements

$$
\begin{array}{rcll}
\text{typing contexts } \Gamma & ::= & \epsilon \mid \Gamma, x : A \mid \Gamma, \hat{\mathbf{x}} : \forall \bar{\alpha}.B & \\
\text{monomorphic typing contexts } \Xi & ::= & \epsilon \mid \Xi, x : A & \\
\text{polymorphic typing contexts } \Pi & ::= & \epsilon \mid \Pi, \hat{\mathbf{x}} : [\Xi]A & \\
\text{(pure) type } A, B & ::= & \texttt{bool} & \text{bool type} \\
& \mid & A \to \underline{C} & \text{function type} \\
& \mid & \underline{C} \Rightarrow \underline{D} & \text{handler type} \\
& \mid & \alpha & \text{type variable} \\
& \mid & \mu\alpha.A & \text{recursive type} \\
& \mid & \top & \text{top} \\
& \mid & \bot & \text{bottom} \\
& \mid & A \sqcap B & \text{intersection} \\
& \mid & A \sqcup B & \text{union} \\
\text{dirty type } \underline{C}, \underline{D} & ::= & A \,!\, \Delta & \\
\text{dirt } \Delta & ::= & \mathtt{Op} & \text{operation} \\
& \mid & \delta & \text{dirt variable} \\
& \mid & \emptyset & \text{empty dirt} \\
& \mid & \Delta_1 \sqcap \Delta_2 & \text{intersection} \\
& \mid & \Delta_1 \sqcup \Delta_2 & \text{union} \\
\text{All operations } \Omega & ::= & \bigsqcup \mathtt{Op}_i | \mathtt{Op}_i \in \Sigma &
\end{array}
$$

Figure 4.2: Types of EffCore

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

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

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

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

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

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

Figure 4.3: Relationship between Equivalence and Subtyping

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

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

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

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

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

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

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

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

Figure 4.4: Equations of distributive lattices for types

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

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

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

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

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

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

Figure 4.5: Equations for function, handler and dirty 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 4.6: Equations of distributive lattices for dirts

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

Expressions

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

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

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

Computations

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

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

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

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

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

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

Figure 4.7: Typing of EffCore

ΞDEF
Ξ contains free λ-bound variables

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

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 + dirt vars)

INTER

$$\frac{\begin{array}{c} dom(\Xi_1 \sqcap \Xi_2) = dom(\Xi_1) \cup dom(\Xi_2) \\ (\Xi_1 \sqcap \Xi_2)(x) = \Xi_1(x) \sqcap \Xi_2(x), \text{ interpreting } \Xi_i(x) = \top \text{ if } x \in dom(\Xi_i) \ (\text{for } i \in \{1,2\}) \end{array}}{\Xi_1 \text{ and } \Xi_2 \text{ have greatest lower bound: } \Xi_1 \sqcap \Xi_2}$$

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

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

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

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

Figure 4.8: Definitions for typing schemes and reformulated typing rules

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

Expressions

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

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

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

Computations

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

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

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

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

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

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

Figure 4.9: Reformulated typing rules of EffCore

Chapter 5

Proofs

TODO: Big part todo in second semester

5.1 Instantiation

5.2 Weakening

5.3 Substitution

5.4 Soundness

Chapter 6

Type Inference

6.1 Polar types

6.2 Unification

To operate on polar type terms, we generalise from substitutions to bisubstitutions, which map type variables to a pair of a positive and a negative type term. The definitions for bisubstitions are given in Figure 6.2.

TODO: explain substitutions and bisubstitions

TODO: rewrite, in essence explain why equivalence of the parameterisation is important " Thus, when manipulating constraints, an ML type checker need only preserve equivalence of the set of instances, and not equivalence of the parameterisation. This freedom is not much used in plain ML, since unification happens to preserve equivalence of the parameterisation. However, this freedom is what allows MLsub to eliminate subtyping constraints.

For all positive type terms $A+$ and variables, there exist positive type terms A^+_α and A^+_g such that $A^+_\alpha \in \bot, \alpha$, α is guarded in A^+_g, and A^+ is equivalent to $A^+_\alpha \sqcup A^+_g$.

For all negative type terms $A-$ and variables, there exist negative type terms A^-_α and A^-_g such that $A^-_\alpha \in \top, \alpha$, α is guarded in A^-_g, and A^- is equivalent to $A^-_\alpha \sqcap A^-_g$. "

6.3 Principal Type Inference

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

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)
	$\mid$	$\Delta_1^- \sqcup \Delta_2^-$	union
	$\mid$	$\Delta_1^- \sqcap \Delta_2^-$	intersection

Figure 6.1: Polar types of EffCore

$$\textsc{Bisubstitution}$$
$$\xi = [A^+/\alpha^+, A^-/\alpha^-, \Delta^+/\delta^+, \Delta^-/\delta^-]$$

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

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

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

$$\xi(\mathtt{0p}) \equiv \mathtt{0p} \qquad \xi(\Delta_1^- \sqcup \Delta_2^-) \equiv \xi(\Delta_1^-) \sqcup \xi(\Delta_2^-)$$

$$\xi(\emptyset) \equiv \emptyset \qquad \xi(\mathtt{0p}) \equiv \mathtt{0p}$$

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

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

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

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

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

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

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

Figure 6.2: Bisubstitutions

$$\forall\alpha\forall\beta.\alpha \to \beta \to \alpha \qquad \forall\beta\forall\alpha.\alpha \to \beta \to \alpha$$

$$\{\alpha \to \beta \to \alpha \mid \alpha, \beta \text{types}\} \qquad \{\alpha \to \beta \to \alpha \mid \beta, \alpha \text{types}\}$$

Figure 6.3: Parameterisation and typing

$$\mu^{+}\alpha.A^{+} = \mu\alpha.A_{g}^{+} \qquad \mu^{-}\alpha.A^{-} = \mu\alpha.A_{g}^{-}$$

Figure 6.4: Polar recursive type

Constructed (predicate):
constructed(A)

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

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

$$constructed(bool)$$

Figure 6.5: Constructed types

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

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

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

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

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

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

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

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

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

Figure 6.6: Constraint solving

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Figure 6.7: Constraint decomposition

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 6.8: Biunification algorithm

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

Expressions

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

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

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

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

Figure 6.9: Type inference algorithm for expressions

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

Computations

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

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

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

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

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

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

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

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

Figure 6.10: Type inference algorithm for computations

Chapter 7

Implementation

TODO: describe implementation (language, size, ...)

Describe the approach itself, in such detail that a reader could also implement this approach if s/he wished to do that.

Chapter 8

Evaluation

TODO: second semester, after experiments Novel approaches to problems are often evaluated empirically. Describe the evaluation process in such detail that a reader could reproduce the results. Describe in detail the setup of an experiment. Argue why this experiment is useful, and what you could learn from it. Be precise about what you want to measure, or about the hypothesis that you are testing. Discuss and interpret the results in terms of your experimental questions. Summarize the conclusions of the experimental evaluation.

Chapter 9

Conclusion

TODO: second semester

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

Appendices

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] 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.

[4] 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.

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

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

[7] 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.

[8] S. Lindley, C. McBride, and C. McLaughlin. Do be do be do. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, pages 500–514, New York, NY, USA, 2017. ACM.

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

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

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

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

[13] D. Rémy. Theoretical aspects of object-oriented programming. chapter Type Inference for Records in Natural Extension of ML, pages 67–95. MIT Press, Cambridge, MA, USA, 1994.

Master's thesis filing card

Student: Axel Faes

Title: Algebraic Subtyping for Algebraic Types and Effects

UDC:

Abstract:

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

Thesis supervisor: Prof. dr. ir. Tom Schrijvers

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

Mentor: Amr Hany Shehata Saleh