# Algebraic subtyping for algebraic effects and handlers

Axel Faes student : undergraduate ACM Student Member: 2461936 Department of Computer Science KU Leuven axel.faes@student.kuleuven.be Amr Hany Saleh
daily advisor
Department of Computer Science
KU Leuven
amrhanyshehata.saleh@kuleuven.be

Tom Schrijvers

promotor

Department of Computer Science

KU Leuven

tom.schrijvers@kuleuven.be

## **Abstract**

**Keywords** algebraic effect handler, algebraic subtyping, effects, optimised compilation

# 1 Introduction

# 2 Background (EFF)

The type-&-effect system that is used in EFF is based on subtyping and dirty types [1].

## 2.1 Types and terms

**Terms** Figure 1 shows the two types of terms 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.

**Types** Figure 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  $\Delta$ , which we call dirt, that can be called. This finite set  $\Delta$  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.

```
value v := x
                                                  variable
                                                  constant
                                                  function
                 fun x \mapsto c
                                                  handler
                     return x \mapsto c_r,
                                                      return case
                     [\operatorname{Op} y\, k \mapsto c_{\operatorname{Op}}]_{\operatorname{Op} \in O}
                                                      operation cases
                 }
comp c ::= v_1 v_2
                                                  application
                 let rec f x = c_1 in c_2
                                                  rec definition
                 \operatorname{return} v
                                                  returned val
                 0p υ
                                                  operation call
                 do x \leftarrow c_1; c_2
                                                  sequencing
                 handle c with v
                                                  handling
```

Figure 1. Terms of Eff

(pure) type 
$$A, B ::= bool \mid int$$
 basic types 
$$\mid A \to \underline{C} \qquad \text{function type}$$
 
$$\mid \underline{C} \Rightarrow \underline{D} \qquad \text{handler type}$$
 dirty type  $\underline{C}, \underline{D} ::= A ! \Delta$  
$$\text{dirt } \Delta ::= \{\mathsf{Op}_1, \ldots, \mathsf{Op}_n\}$$

Figure 2. Types of Eff

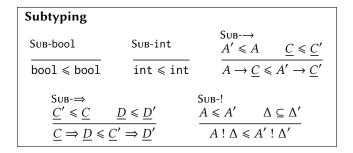


Figure 3. Subtyping for pure and dirty types of Eff

## 2.2 Type System

#### 2.2.1 Subtyping

The dirty type A !  $\Delta$  is assigned to a computation returning values of type A and potentially calling operations from the set  $\Delta$ . This set  $\Delta$  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. Observe that, as usual, subtyping is contravariant in the argument types of functions and handlers, and covariant in their return types.

#### 2.2.2 Typing rules

Figure 4 defines the typing judgements for values and computations with respect to a standard typing context  $\Gamma$ .

**Values** The rules for subtyping, variables, and functions are entirely standard. For constants we assume a signature  $\Sigma$  that assigns a type A to each constant k, which we write as  $(k : A) \in \Sigma$ .

1

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 (Op:  $A_{\rm Op} \to B_{\rm Op}$ )  $\in \Sigma$  determines the type  $A_{\rm Op}$  of the parameter x and the domain  $B_{\rm Op}$  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.

**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  $\operatorname{Op} v$  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  $\underline{D}$ .

# 3 Core language

 $\sqcup$  is for outputs,  $\sqcap$  is for inputs.

- 3.1 Types and terms
- 3.2 Type system
- 3.2.1 Subtyping
- 3.2.2 Typing rules
- 4 Elaboration
- 5 Proofs
- 6 Conclusion

## Acknowledgments

I would like to thank Amr Hany Saleh for his continuous guidance and help.

## 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

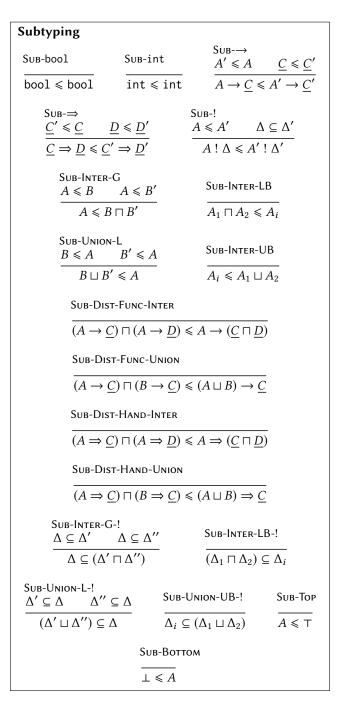
Figure 4. Typing of Eff

```
value v := x
                                               variable
                k
                                               constant
            |\lambda x.c|
                                               function
                                               type abstraction
                \Lambda \alpha.c
                                               handler
                   return x \mapsto c_r,
                                                  return case
                   [\mathsf{Op}\,y\,k\mapsto c_{\mathsf{Op}}]_{\mathsf{Op}\in O}
                                                  operation cases
                                               application
\mathsf{comp}\,\,c \quad ::= \,v_1\,v_2
                if e then c_1 else c_2
                                               conditional
                                               type application
                let rec f x = c_1 in c_2
                                               rec definition
                return v
                                               returned val
                0pv
                                               operation call
                do x \leftarrow c_1; c_2
                                               sequencing
                handle c with v
                                               handling
```

Figure 5. Terms of the core language

```
(pure) type A, B ::= bool \mid int
                                                basic types
                           A \rightarrow \underline{C}
                                                function type
                                                handler type
                            \underline{C} \Rightarrow \underline{D}
                                                type variable
                             \alpha
                             \forall \alpha.C
                                                polytype
                                                top
                                                bottom
                            A \sqcap B
                                                intersection
                          A \sqcup B
                                                union
 dirty type C, \underline{D} ::= A ! \Delta
             dirt \Delta ::= \{R\}
                  R ::= Op; R
                                                row
                                                row variable
                                                closed row
                                                intersection
                            R_1 \sqcap R_2
                                                union
                            R_1 \sqcup R_2
```

Figure 6. Types of the core language



**Figure 7.** Subtyping for pure and dirty types of the core language

$$\begin{aligned} & \text{typing contexts } \Gamma ::= \epsilon \mid \Gamma, x : A \\ \textbf{Expressions} \\ & \frac{\text{VAL}}{\Gamma \vdash v : A} \quad A \leqslant B \\ \hline & \frac{(x : A) \in \Gamma}{\Gamma \vdash x : A} \quad \frac{(k : A) \in \Sigma}{\Gamma \vdash k : A} \\ \hline & \frac{\text{FUN}}{\Gamma \vdash k : A} \quad \frac{(x : A) \in \Gamma}{\Gamma \vdash k : A} \quad \frac{(k : A) \in \Sigma}{\Gamma \vdash k : A} \\ \hline & \frac{\text{FUN}}{\Gamma \vdash k : A} \quad \frac{(x : A) \in \Gamma}{\Gamma \vdash k : A} \quad \frac{\text{Type Abstraction}}{\Gamma, \alpha \vdash c : C} \\ \hline & \frac{\Gamma, \alpha \vdash c : C}{\Gamma \vdash \lambda \alpha.c} : \forall \alpha.C \\ \hline & \frac{C}{\Gamma \vdash \lambda x.c} : A \to C \\ \hline & \frac{C}{\Gamma \vdash \lambda x.c} : A \to C \\ \hline & \frac{C}{\Gamma \vdash \lambda \alpha.c} : \frac{C}{\Gamma \vdash \lambda \alpha.c} : \forall \alpha.C \\ \hline & \frac{C}{\Gamma \vdash \lambda x.c} : A \to C \\ \hline &$$

Figure 8. Typing of the explicitly typed language