A type system for algebraic effects and handlers with dynamic instances

Albert ten Napel

Contents

Chapter 1

Introduction

In this thesis we will devise a type and effect systems that can type some programs that use dynamic instances for algebraic effects and handlers.

1.1 Problem statement

1.2 Proposed solution

1.3 Thesis structure

Chapter 2

Background

In this chapter we will show the basics of algebraic effects and handlers. We will start with the simply-typed lambda calculus (2.1) and add algebraic effects and instances to it. We end with dynamic instances and show why a type system for them is difficult to implement.

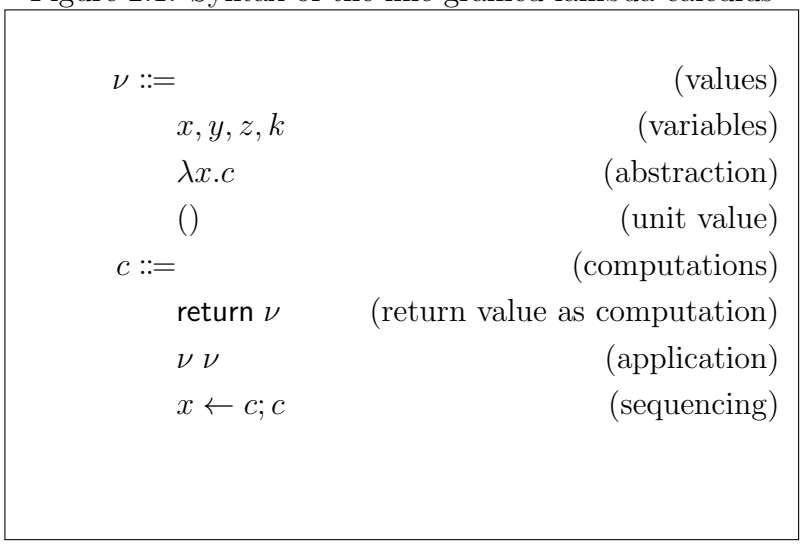
2.1 Simply-typed lambda calculus

As our base language we will take the fine-grained call-by-value simply-typed lambda calculus (FG-STLC) [8]. This system is a version of the simply-typed lambda calculus with a syntactic distinction between values and computations. Because of this distinction there is exactly one evaluation order: call-by-value. In a system with side effects the evaluation order is very important since a different order could have a different result. Having the evaluation order be apparent from the syntax is thus a good choice for a system with algebraic effects. Another way to look at FG-STLC is to see it as a syntax for the lambda calculus that constrains the program to always be in A-normal form [9].

The terms are shown in Figure 2.1. The terms are split in to values and computations. Values are pieces of data that have no effects, while computations are terms that may have effects.

Values We have x, y, z, k ranging over variables, where we will use k for

Figure 2.1: Syntax of the fine-grained lambda calculus



variables that denote continuations later on. We also have the usual abstractions though note that the body is a computation. To keep things simple we take unit as our only base value, this because adding more base values will not complicate the theory. Using the unit value we can also delay computations by wrapping them in an abstraction that takes a unit value.

Computations For any value ν we have return ν for the computation that simply returns a value without performing any effects. We have function application $(\nu\ \nu)$, where both the function and argument have to be values. Sequencing computations is done with $(x \leftarrow c; c)$. Normally in the lambda calculus the function and the argument in an application could be any term and so a choice would have to be made in what order these have to be evaluated or whether to evaluate the argument at all before substitution. In the fine-grained calculus both the function and argument in $(\nu\ \nu)$ are values so there's no choice of evaluation order. The order is made explicit by the sequencing syntax $(x \leftarrow c; c)$.

Semantics The small-step operational semantics is shown in Figure 2.2. The relation $\rightsquigarrow$ is defined on computations, where the $c \rightsquigarrow c'$ means c reduces to c' in one step. These rules are a fine-grained approach to the standard reduction rules of the simply-typed lambda calculus. In STLC-S-App we

Figure 2.2: Semantics of the fine-grained lambda calculus

$$\frac{}{(\lambda x.c)\ \nu \rightsquigarrow c[x := \nu]} \quad \text{(STLC-S-App)}$$

$$\frac{}{(x \leftarrow \mathsf{return}\ \nu; c) \rightsquigarrow c[x := \nu]} \quad \text{(STLC-S-SeqReturn)}$$

$$\frac{c_1 \rightsquigarrow c_1'}{(x \leftarrow c_1; c_2) \rightsquigarrow (x \leftarrow c_1'; c_2)} \quad \text{(STLC-S-Seq)}$$

Figure 2.3: Types of the fine-grained simply-typed lambda calculus

τ	::=	(value types)
	$()$	(unit type)
	$\tau \to \underline{\tau}$	(type of functions)
$\underline{\tau}$	::=	(computation types)
	τ	(value type)

apply a lambda abstraction to a value argument, by substituting the value for the variable x in the body of the abstraction. In STLC-S-SeqReturn we sequence a computation that just returns a value in another computation by substituting the value for the variable x in the computation. Lastly, in STLC-S-Seq we can reduce a sequence of two computations, c_1 and c_2 by reducing the first, c_1.
We define $\rightsquigarrow^*$ as the transitive-reflexive closure of $\rightsquigarrow$.

Types Next we give the *types* in Figure 2.3. Similar to the terms we split the syntax into value and computation types. Values are typed by value types and computations are typed by computation types. A value type is either the unit type $()$ or a function type with a value type τ as argument type and a computation type $\underline{\tau}$ as return type.
For the simply-typed lambda calculus a computation type is simply a value

Figure 2.4: Typing rules of the fine-grained simply-typed lambda calculus

$$\frac{\Gamma[x] = \tau}{\Gamma \vdash x : \tau} \quad \text{(STLC-T-Var)}$$

$$\frac{}{\Gamma \vdash () : ()} \quad \text{(STLC-T-Unit)}$$

$$\frac{\Gamma, x : \tau_1 \vdash c : \underline{\tau}_2}{\Gamma \vdash \lambda x.c : \tau_1 \to \underline{\tau}_2} \quad \text{(STLC-T-Abs)}$$

$$\frac{\Gamma \vdash \nu : \tau}{\Gamma \vdash \mathsf{return}\ \nu : \underline{\tau}} \quad \text{(STLC-T-Return)}$$

$$\frac{\Gamma \vdash \nu_1 : \tau_1 \to \underline{\tau}_2 \qquad \Gamma \vdash \nu_2 : \tau_1}{\Gamma \vdash \nu_1\ \nu_2 : \underline{\tau}_2} \quad \text{(STLC-T-App)}$$

$$\frac{\Gamma \vdash c_1 : \underline{\tau}_1 \qquad \Gamma, x : \tau_1 \vdash c_2 : \underline{\tau}_2}{\Gamma \vdash (x \leftarrow c_1; c_2) : \underline{\tau}_2} \quad \text{(STLC-T-Seq)}$$

type, but when we add algebraic effects computation types will become more meaningful by recording the effects a computation may use.

Typing rules Finally we give the typing rules in Figure 2.4. We have a typing judgment for values $\Gamma \vdash \nu : \tau$ and a typing judgment for computations $\Gamma \vdash c : \underline{\tau}$. In both these judgments the context Γ assigns value types to variables.
The rules for variables (STLC-T-Var), unit (STLC-T-Unit), abstractions (STLC-T-Abs) and applications (STLC-T-App) are the standard typing rules of the simply-typed lambda calculus. For $\mathsf{return}\ \nu$ (STLC-T-Return) we simply check the type of ν. For the sequencing of two computations $(x \leftarrow c_1; c_2)$ (STLC-T-Seq) we first check the type of c_1 and then check c_2 with the type of c_1 added to the context for x.

Examples To show the explicit order of evaluation we will translate the following program from the simply-typed lambda calculus into its fine-grained version:

$$f\ c_1\ c_2$$

Here we have a choice of whether to first evaluate c_1 or c_2 and whether to evaluate $(f\ c_2)$ before evaluating c_2. In the fine-grained system the choice of evaluation order is made explicit by the syntax. This means we can write down three variants for the above program, each having a different evaluation order. In the presence of effects all three may have different results.

1. c_1 before c_2, c_2 before $(f\ c_1)$

$$x' \leftarrow c_1; y' \leftarrow c_2; g \leftarrow (f\ x'); (g\ y')$$

2. c_2 before c_1, c_2 before $(f\ c_1)$

$$y' \leftarrow c_2; x' \leftarrow c_1; g \leftarrow (f\ x'); (g\ y')$$

3. c_1 before c_2, c_2 before $(f\ c_1)$

$$x' \leftarrow c_1; g \leftarrow (f\ x'); y' \leftarrow c_2; (g\ y')$$

To give a more concrete example, let's say we're working in a programming language based on the call-by-value lambda calculus that has arbitrary side-effects. Given a function print that takes an integer and prints it to the screen, we can define the following function printRange that prints a range of integers:

```
-- given print : Int -> ()
printRange : Int -> Int -> ()
printRange a b =
  if a > b then
    ()
  else
    (\a b -> ()) (print a) (printRange (a + 1) b)
```

Knowing the evaluation order is very important when evaluating the call `(printRange 1 10)`.
In the expression `(\a b -> ()) (print a) (printRange (a + 1) b)` the arguments can be either evaluated left-to-right or right-to-left. This makes a big difference in the output of the program, in left-to-right order the numbers 1 to 10 will be printed in increasing order while using a right-to-left evaluation strategy will print the numbers 10 to 1 in decreasing order.
From the syntax of the language we are not able to deduce with evaluation order will be used, even worse it may be left undefined in the language definition.
Translating this to a language that uses a fine-grain style syntax results in:

```
-- given print : Int -> ()
printRange : Int -> Int -> ()
printRange a b =
  if a > b then
    ()
  else
    _ <- print a;
    printRange (a + 1) b
```

Here from the syntax it is made clear that `print a` should be evaluated before `printRange (a + 1) b`, meaning a left-to-right evaluation order.
Alternatively we could translate the first program as:

```
-- given print : Int -> ()
printRange : Int -> Int -> ()
printRange a b =
  if a > b then
    ()
  else
    _ <- printRange (a + 1) b;
    print a
```

Making clear we want a right-to-left evaluation order, printing the numbers in decreasing order.

Theorems We define a computation c to be a value if c is of the form $\mathsf{return}\ \nu$

for some value ν.

$$\mathsf{value}(c) \text{ if } \exists \nu.c = \mathsf{return}\ \nu$$

We can prove the following type soundness theorem for the fine-grained simply typed lambda calculus.

Theorem 1 (Type soundness)**.**

$$\mathsf{if}\ \cdot \vdash c : \underline{\tau} \wedge c \rightsquigarrow^* c'\ \mathsf{then}\ \mathsf{value}(c')$$

This states that given a well-typed computation c and taking as many steps as possible then the resulting computation c' will be of the form $\mathsf{return}\ \nu$ for some value ν. In other words the term will not get "stuck". We can proof this theorem by the usual progress and preservation lemmas:

Lemma 1 (Progress)**.**

$$\mathsf{if}\ \cdot \vdash c : \underline{\tau}\ \mathsf{then}\ \mathsf{value}(c) \vee (\exists c'.c \rightsquigarrow c')$$

Lemma 2 (Preservation)**.**

$$\mathsf{if}\ \cdot \vdash c : \underline{\tau} \wedge c \rightsquigarrow c'\ \mathsf{then}\ \cdot \vdash c' : \underline{\tau}$$

Where the progress lemma states that given a well-typed computation c then either c is a value or c can take a step. The preservation lemma states that given a well-type computation c and if c can take a step to c' then c' is also well-typed. We can prove both these by induction on the typing derivations.

2.2 Algebraic effects

2.2.1 Intro

Explain:

- What are algebraic effects and handlers
- Why algebraic effects
  - easy to use
  - can express often used monads
  - composable
  - always commuting
  - modular (split between computations and handlers)

Algebraic effects and handlers are a way of treating computational effects that is modular and compositional.

With algebraic effects impure behavior is modeled using operations. For example a mutable store has get and put operations, exceptions have a throw operation and console input/output has read and print operations. Handlers of algebraic effects generalize handlers of exceptions by not only catching called operations but also adding the ability to resume where the operation was called. While not all monads can be written in terms of algebraic effects, for example the continuation monad, in practice most useful computation effects can be modeled this way.
For example we can model stateful computations that mutate an integer by defining the following algebraic effect signature:

$$\mathbf{State} := \{\mathbf{get} : () \to \mathbf{Int}, \mathbf{put} : \mathbf{Int} \to ()\}$$

State is an effect that has two operations **get** and **put**. **get** takes unit has its parameter type and returns an integer value, **put** takes an integer value and returns unit.
We can then use the **State** operations in a program:

$$inc\ () := x \leftarrow \textbf{get}\ ();\ \textbf{put}\ (x + 1)$$

The program **inc** uses the **get** and **put**, but these operations are abstract. Handlers are used to give the abstract effects in a computation semantics.

Handlers Algebraic effect handlers can be seen as a generalization of exception handlers where the programmer also has access to a continuation that continues from the point of where a operation was called.

For example the following handler gives the **get** and **put** the usual function-passing style state semantics:

$$state\ := \textbf{handler}\ \{\ \textbf{return}\ v \to \lambda s \to v, \textbf{get}\ ()\ k \to \lambda s \to k\ s\ s, \textbf{put}\ s\ k \to \lambda s' \to k\ ()\ s, \}$$

We are able to give different interpretations of a computation by using different handlers. We could for example think of a transaction state interpretation where changed to the state are only applied at the end if the computation succeeds.

Examples

```
effect Flip {
  flip : () -> Bool
}

program = \() ->
  b <- flip ();
  if b then
    flip ()
  else
    false

effect State {
  get : () -> Int
  put : Int -> ()
}
```

```
postInc = \() ->
  n <- get ();
  put (n + 1);
  return n
```

2.2.2 Syntax

2.2.3 Semantics

2.2.4 Type system

2.2.5 Discussion/limitations

2.3 Static instances

Should I even mention static instances?

2.3.1 Intro

Explain:

- Show problems with wanting to use multiple state instances
- What are static instances
- Show that static instances partially solve the problem

2.3.2 Syntax

2.3.3 Semantics

2.3.4 Type system

2.3.5 Examples

Show state with multiple static instances (references).

2.4 Dynamic instances (untyped)

2.4.1 Intro

Explain:

- Show that static instances require pre-defining all instances on the top-level.
- Static instances not sufficient to implement references.
- Show that dynamic instances are required to truly implement references.
- Show more uses of dynamic instances (file system stuff, local exceptions)
- No type system yet.

2.4.2 Syntax

2.4.3 Semantics

2.4.4 Examples

Show untyped examples.

- Local exceptions
- ML-style references

2.4.5 Type system (discussion, problems)

Show difficulty of implementing a type system for this.

Chapter 3

Type and effect system

3.1 Syntax

We assume there is set of effect names $E = \{\varepsilon_1, ..., \varepsilon_n\}$. Each effect has a set of operation names $O_\varepsilon = \{op_1, ..., op_n\}$. Every operation name only corresponds to a single effect. Each operation has a parameter type τ^1_{op} and a return type τ^2_{op}. We have scope variables s modeled by some countable infinite set. And we have locations l modeled by some countable infinite set. Annotations r are sets of pairs of an effect name with a scope variable: $\{E_1@s_1, ..., E_n@s_n\}$.

Figure 3.1: Syntax

$s ::=$		(scopes)
	s_{var}	(scope variable)
	s_{loc}	(scope location)
$\tau ::=$		(value types)
	$\mathsf{Inst}\ s\ \varepsilon$	(instance type)
	$\tau \to \underline{\tau}$	(type of functions)
	$\forall s_{var}.\underline{\tau}$	(universally quantified type over scope s)
$\underline{\tau} ::=$		(computation types)
	$\tau\ !\ r$	(annotated type)
$\nu ::=$		(values)
	x, y, z, k	(variables)
	$\mathsf{inst}(s, l)$	(instance values (for semantics))
	$\lambda x.c$	(abstraction)
	$\Lambda s_{var}.c$	(scope abstraction)
$c ::=$		(computations)
	$\mathsf{return}\ \nu$	(return value as computation)
	$\nu\ \nu$	(application)
	$\nu\ [s]$	(scope application)
	$x \leftarrow c;\ c$	(sequencing)
	$\nu \# op(\nu)$	(operation call)
	$\mathsf{new}\ \varepsilon @ s\ \{h; \mathsf{finally}\ x \to c\}\ \mathsf{as}\ x\ \mathsf{in}\ c$	(instance creation)
	$\mathsf{handle}(\nu)$	(handle scoped computation)
	$\mathsf{handle}^{s_{loc}}(c)$	(handle computation (for semantics))
	$\mathsf{handle}^{l}\{h\}(c)$	(handle instance (for semantics))
$h ::=$		(handlers)
	$op\ x\ k \to c; h$	(operation case)
	$\mathsf{return}\ x \to c$	(return/finally case)

Figure 3.2: Subtyping

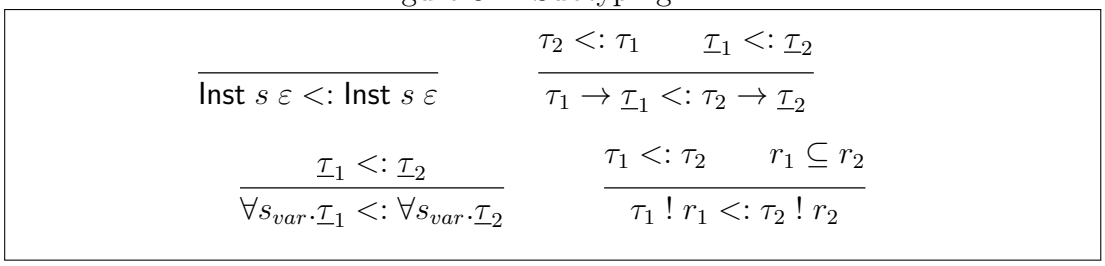


Figure 3.3: Well-formedness

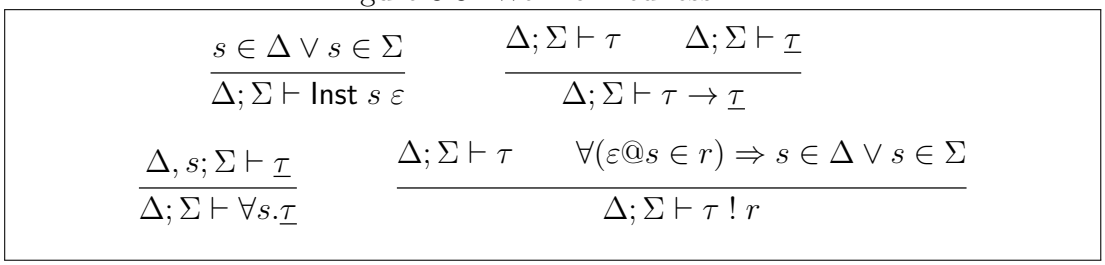


Figure 3.4: Value typing rules

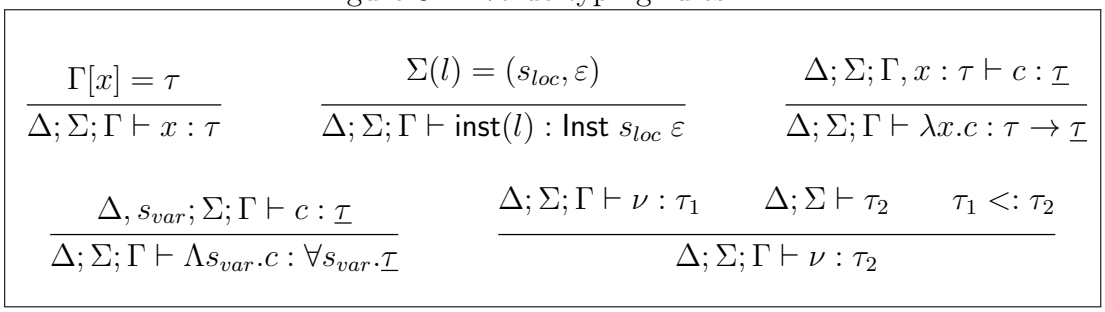


Figure 3.5: Computation typing rules

$$\frac{\Delta; \Sigma; \Gamma \vdash \nu : \tau}{\Delta; \Sigma; \Gamma \vdash \mathsf{return}\ \nu : \tau\ !\ \varnothing} \qquad \frac{\Delta; \Sigma; \Gamma \vdash \nu_1 : \tau \to \underline{\tau} \qquad \Delta; \Sigma; \Gamma \vdash \nu_2 : \tau}{\Delta; \Sigma; \Gamma \vdash \nu_1\ \nu_2 : \underline{\tau}}$$

$$\frac{s \in \Delta \vee s \in \Sigma \qquad \Delta; \Sigma; \Gamma \vdash \nu : \forall s'_{var}.\underline{\tau}}{\Delta; \Sigma; \Gamma \vdash \nu\ [s] : \underline{\tau}[s'_{var} := s]}$$

$$\frac{\Delta; \Sigma; \Gamma \vdash c_1 : \tau_1\ !\ r \qquad \Delta; \Sigma; \Gamma, x : \tau_1 \vdash c_2 : \tau_2\ !\ r}{\Delta; \Sigma; \Gamma \vdash (x \leftarrow c_1;\ c_2) : \tau_2\ !\ r}$$

$$\frac{\Delta; \Sigma; \Gamma \vdash \nu_1 : \mathsf{Inst}\ s\ \varepsilon \qquad op \in O_\varepsilon \qquad \Delta; \Sigma; \Gamma \vdash \nu_2 : \tau^1_{op}}{\Delta; \Sigma; \Gamma \vdash \nu_1 \# op(\nu_2) : \tau^2_{op}\ !\ \{\varepsilon @ s\}}$$

$$\frac{\begin{array}{c} s \in \Delta \vee s \in \Sigma \qquad op \in O_\varepsilon \iff op \in h \qquad \Delta; \Sigma; \Gamma, x : \mathsf{Inst}\ s\ \varepsilon \vdash c : \tau_1\ !\ r \\ \Delta; \Sigma; \Gamma \vdash^{\tau_1} h : \tau_2\ !\ r \qquad \varepsilon @ s \in r \qquad \Delta; \Sigma; \Gamma, y : \tau_2 \vdash c' : \tau_3\ !\ r \end{array}}{\Delta; \Sigma; \Gamma \vdash \mathsf{new}\ \varepsilon @ s\ \{h; \mathsf{finally}\ y \to c'\}\ \mathsf{as}\ x\ \mathsf{in}\ c : \tau_3\ !\ r}$$

$$\frac{\Delta; \Sigma; \Gamma \vdash \nu : \forall s_{var}.\tau\ !\ r \qquad s_{var} \notin \tau \qquad r' = \{\varepsilon @ s' \mid \varepsilon @ s' \in r \wedge s' \neq s_{var}\}}{\Delta; \Sigma; \Gamma \vdash \mathsf{handle}(\nu) : \tau\ !\ r'}$$

$$\frac{\begin{array}{c} s_{loc} \in \Sigma \\ \Delta; \Sigma; \Gamma \vdash c : \tau\ !\ r \qquad s_{loc} \notin \tau \qquad r' = \{\varepsilon @ s' \mid \varepsilon @ s' \in r \wedge s' \neq s_{loc}\} \end{array}}{\Delta; \Sigma; \Gamma \vdash \mathsf{handle}^{s_{loc}}(c) : \tau\ !\ r'}$$

$$\frac{\begin{array}{c} \Sigma(l) = (s_{loc}, \varepsilon) \qquad op \in O_\varepsilon \iff op \in h \\ \Delta; \Sigma; \Gamma \vdash^{\tau_1} h : \tau_2\ !\ r \qquad \Delta; \Sigma; \Gamma \vdash c : \tau_1\ !\ r \qquad \varepsilon @ s_{loc} \in r \end{array}}{\Delta; \Sigma; \Gamma \vdash \mathsf{handle}^{l}\{h\}(c) : \tau_2\ !\ r}$$

$$\frac{\Delta; \Sigma; \Gamma \vdash c : \underline{\tau}_1 \qquad \Delta; \Sigma \vdash \underline{\tau}_2 \qquad \underline{\tau}_1 <: \underline{\tau}_2}{\Delta; \Sigma; \Gamma \vdash c : \underline{\tau}_2}$$

Figure 3.6: Handler typing rules

$$\frac{\Delta; \Sigma; \Gamma, x : \tau^1_{op}, k : \tau^2_{op} \to \tau_2 \,!\, r \vdash c : \tau_2 \,!\, r \qquad \Delta; \Sigma; \Gamma \vdash^{\tau_1} h : \tau_2 \,!\, r}{\Delta; \Sigma; \Gamma \vdash^{\tau_1} (op\ x\ k \to c; h) : \tau_2 \,!\, r}$$

$$\frac{\Delta; \Sigma; \Gamma, x : \tau_1 \vdash c : \tau_2 \,!\, r}{\Delta; \Sigma; \Gamma \vdash^{\tau_1} (\mathsf{return}\ x \to c) : \tau_2 \,!\, r}$$

Figure 3.7: Semantics

$$\frac{}{(\lambda x.c)\ \nu \mid \Sigma \rightsquigarrow c[x := \nu] \mid \Sigma} \qquad \frac{}{(\Lambda s.c)\ [s'] \mid \Sigma \rightsquigarrow c[s := s'] \mid \Sigma}$$

$$\frac{c_1; \Sigma \rightsquigarrow c_1'; \Sigma'}{(x \leftarrow c_1;\ c_2) \mid \Sigma \rightsquigarrow (x \leftarrow c_1';\ c_2) \mid \Sigma'} \qquad \frac{}{(x \leftarrow (\mathsf{return}\ \nu);\ c) \mid \Sigma \rightsquigarrow c[x := \nu] \mid \Sigma}$$

$$\frac{}{(y \leftarrow (x \leftarrow c_1;\ c_2);\ c_3 \mid \Sigma \rightsquigarrow (x \leftarrow c_1;\ y \leftarrow c_2;\ c_3) \mid \Sigma}$$

$$\frac{}{(x \leftarrow (\mathsf{new}\ \varepsilon@s\ \{h; \mathsf{finally}\ z \to c_3\}\ \mathsf{as}\ y\ \mathsf{in}\ c_1);\ c_2) \mid \Sigma \rightsquigarrow \mathsf{new}\ \varepsilon@s\ \{h; \mathsf{finally}\ z \to c_3\}\ \mathsf{as}\ y\ \mathsf{in}\ (x \leftarrow c_1;\ c_2) \mid \Sigma}$$

$$\frac{s_{loc} \notin \Sigma}{\mathsf{handle}(\Lambda s_{var}.c) \mid \Sigma \rightsquigarrow \mathsf{handle}^{s_{loc}}(c[s_{var} := s_{loc}]) \mid \Sigma, s_{loc}}$$

Figure 3.8: Semantics of new handlers

$$\frac{c \mid \Sigma \rightsquigarrow c' \mid \Sigma'}{\mathsf{handle}^{s_{loc}}(c) \mid \Sigma \rightsquigarrow \mathsf{handle}^{s_{loc}}(c') \mid \Sigma'}$$

$$\frac{}{\mathsf{handle}^{s_{loc}}(\mathsf{return}\ \nu) \mid \Sigma \rightsquigarrow \mathsf{return}\ \nu \mid \Sigma}$$

$$\frac{}{\mathsf{handle}^{s_{loc}}(\nu_1 \# op(\nu_2)) \mid \Sigma \rightsquigarrow \nu_1 \# op(\nu_2) \mid \Sigma}$$

$$\frac{}{\mathsf{handle}^{s_{loc}}(x \leftarrow \nu_1 \# op(\nu_2);\ c) \mid \Sigma \rightsquigarrow (x \leftarrow \nu_1 \# op(\nu_2);\ \mathsf{handle}^{s_{loc}}(c)) \mid \Sigma}$$

$$\frac{s_{loc} \neq s'_{loc}}{\begin{array}{c}\mathsf{handle}^{s_{loc}}(\mathsf{new}\ \varepsilon@s'_{loc}\ \{h; \mathsf{finally}\ y \rightarrow c'\}\ \mathsf{as}\ x\ \mathsf{in}\ c) \mid \Sigma \rightsquigarrow \\ \mathsf{new}\ \varepsilon@s'_{loc}\ \{h; \mathsf{finally}\ y \rightarrow c'\}\ \mathsf{as}\ x\ \mathsf{in}\ \mathsf{handle}^{s_{loc}}(c) \mid \Sigma\end{array}}$$

$$\frac{l \notin \mathsf{Dom}(\Sigma)}{\begin{array}{c}\mathsf{handle}^{s_{loc}}(\mathsf{new}\ \varepsilon@s_{loc}\ \{h; \mathsf{finally}\ y \rightarrow c'\}\ \mathsf{as}\ x\ \mathsf{in}\ c) \mid \Sigma \rightsquigarrow \\ \mathsf{handle}^{s_{loc}}(y \leftarrow \mathsf{handle}^{l}\{h\}(c[x := \mathsf{inst}(l)]);\ c') \mid \Sigma, l := (s_{loc}, \varepsilon)\end{array}}$$

Figure 3.9: Semantics of instance handlers

$$\frac{c \mid \Sigma \rightsquigarrow c' \mid \Sigma'}{\mathsf{handle}^{l}\{h\}(c) \mid \Sigma \rightsquigarrow \mathsf{handle}^{l}\{h\}(c') \mid \Sigma'}$$

$$\frac{}{\begin{array}{c}\mathsf{handle}^{l}\{h\}(\mathsf{new}\ \varepsilon@s\ \{h'; \mathsf{finally}\ y \rightarrow c'\}\ \mathsf{as}\ x\ \mathsf{in}\ c) \mid \Sigma \rightsquigarrow \\ \mathsf{new}\ \varepsilon@s\ \{h'; \mathsf{finally}\ y \rightarrow c'\}\ \mathsf{as}\ x\ \mathsf{in}\ \mathsf{handle}^{l}\{h\}(c) \mid \Sigma\end{array}}$$

$$\frac{}{\mathsf{handle}^{l}\{h\}(\nu_1 \# op(\nu_2)) \mid \Sigma \rightsquigarrow \mathsf{handle}^{l}\{h\}(x \leftarrow \nu_1 \# op(\nu_2);\ \mathsf{return}\ x) \mid \Sigma}$$

$$\frac{l \neq l'}{\mathsf{handle}^{l}\{h\}(x \leftarrow \mathsf{inst}(l') \# op(\nu);\ c) \mid \Sigma \rightsquigarrow (x \leftarrow \mathsf{inst}(l') \# op(\nu);\ \mathsf{handle}^{l}\{h\}(c)) \mid \Sigma}$$

$$\frac{h[op] = (x, k, c_{op})}{\mathsf{handle}^{l}\{h\}(y \leftarrow \mathsf{inst}(l) \# op(\nu);\ c) \mid \Sigma \rightsquigarrow c_{op}[x := \nu, k := (\lambda y.\mathsf{handle}^{l}\{h\}(c))] \mid \Sigma}$$

$$\frac{}{\mathsf{handle}^{l}\{h; \mathsf{return}\ x_r \rightarrow c_r\}(\mathsf{return}\ \nu) \mid \Sigma \rightsquigarrow c_r[x_r := \nu] \mid \Sigma}$$

3.2 Subtyping

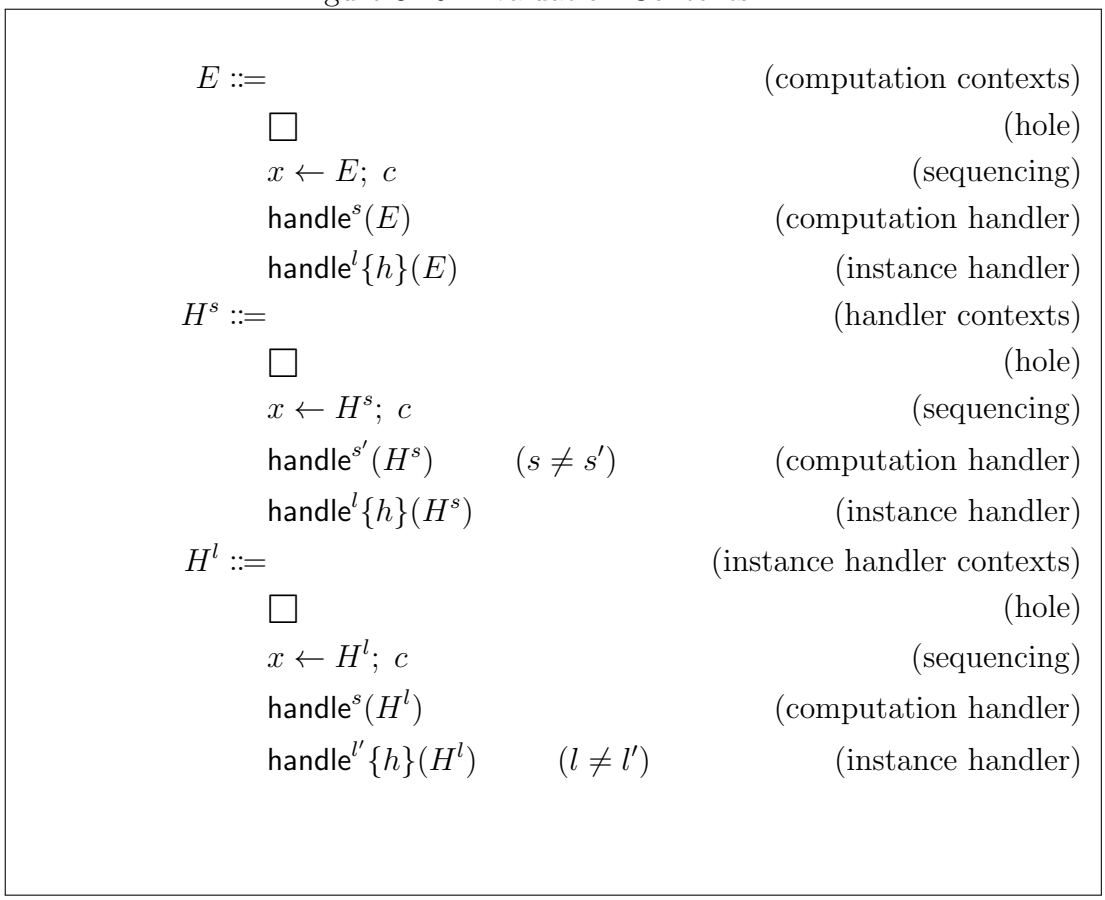
3.3 Well-formedness

3.4 Typing rules

3.5 Semantics

3.6 Evaluation Contexts

Figure 3.10: Evaluation Contexts



Chapter 4

Formalization

Chapter 5

Related work

Chapter 6

Conclusion and future work

	Eff[2][3]	Links [4]	Koka[5]	Frank[6]	Idris (effects library)[7]
Shallow handlers	No	Yes	Yes	Yes	No
Deep handlers	Yes	Yes	Yes	With recursion	Yes
Effect subtyping	Yes	No	No	No	No
Row polymorphism	No	Yes	Only for effects	No	No
Effect instances	Yes	?	Duplicated labels	No	Using labels
Dynamic effects	Yes	No	Using heaps	No	No
Indexed effects	No	No	No	No	Yes

6.1 Shallow and deep handlers

Handlers can be either shallow or deep. Let us take as an example a handler that handles a *state* effect with *get* and *set* operations. If the handler is shallow then only the first operation in the program will be handled and the result might still contain *get* and *set* operations. If the handler is deep then all the *get* and *set* operations will be handled and the result will not contain any of those operations. Shallow handlers can express deep handlers using recursion and deep handlers can encode shallow handlers with an increase in complexity. Deep handlers are easier to reason about *I think expressing deep handlers using shallow handlers with recursion might require polymorphic recursion.*
Frank has shallow handlers by default, while all the other languages have deep handlers. Links and Koka have support for shallow handlers with a *shallowhandler* construct.

In Frank recursion is needed to define the handler for the state effect, since the handlers in Frank are shallow.

```
state : S -> <State S>X -> X
state _ x = x
state s <get -> k> = state s (k s)
state _ <put s -> k> = state s (k unit)
```

Koka has deep handlers and so the handler will call itself recursively, handling all state operations.

```
val state = handler(s) {
  return x -> (x, s)
  get() -> resume(s, s)
  put(s') -> resume(s', ())
}
```

6.2 Effect subtyping and row polymorphism

A handler that only handles the *State* effect must be able to be applied to a program that has additional effects to *State*. Two ways to solve this problem are effect subtyping and row polymorphism. With effect subtyping

we say that the set of effects set_1 is a subtype of set_2 if set_2 is a subset of set_1.

$$\frac{s_2 \subseteq s_1}{s_1 \leq s_2}$$

With row polymorphism instead of having a set of effects there is a row of effects which is allowed to have a polymorphic variable that can unify with effects that are not in the row. We would like narrow a type as much as we can such that pure functions will not have any effects. With row polymorphic types this means having a closed or empty row. These rows cannot be unified with rows that have more effects so one needs to take care to add the polymorphic variable again when unifying, like Koka does.
Eff uses effect subtyping while Links and Koka employ row polymorphism *Not sure yet about Frank and Idris.*

6.3 Effect instances

One might want to use multiple instances of the same effect in a program, for example multiple *state* effects. Eff achieves this by the *new* operator, which creates a new instance of a specific effect. Operations are always called on an instance and handlers also reference the instance of the operations they are handling. In the type annotation of a program the specific instances are named allowing multiple instances of the same effect.
Idris solves this by allowing effects and operations to be labeled. These labels are then also seen in the type annotations.

In Idris labels can be used to have multiple instances of the same effect, for example in the following tree tagging function.

```
-- without labels
treeTagAux : BTree a -> { [STATE (Int, Int)] } Eff (BTree (
    Int, a))
-- with labels
treeTagAux : BTree a -> {['Tag ::: STATE Int, 'Leaves :::
    STATE Int]} Eff (BTree (Int, a))
```

Operations can then be tagged with a label.

```
treeTagAux Leaf = do
        'Leaves :- update (+1)
        pure Leaf
treeTagAux (Node l x r) = do
        l' <- treeTagAux l
        i <- 'Tag :- get
        'Tag :- put (i + 1)
        r' <- treeTagAux r
        pure (Node l' (i, x) r')
```

In Eff one has to instantiate an effect with the *new*, operations are called on this instance and they can also be arguments to an handler.

```
type 'a state = effect
  operation get: unit -> 'a
  operation set: 'a -> unit
end

let r = new state

let monad_state r = handler
  | val y -> (fun _ -> y)
  | r#get () k -> (fun s -> k s s)
  | r#set s' k -> (fun _ -> k () s')

let f = with monad_state r handle
    let x = r#get () in
    r#set (2 * x);
    r#get ()
in (f 30)
```

6.4 Dynamic effects

One effect often used in imperative programming languages is dynamic allocation of ML-style references. Eff solves this problem using a special type of effect instance that holds a *resource*. This amounts to a piece of state that can be dynamically altered as soon as a operation is called. Note that this is impure. Haskell is able to emulate ML-style references using the ST-monad where the reference are made sure not to escape the thread where they are

used by a rank-2 type. Koka annotates references and read/write operations with the heap they are allowed to use.

In Eff resources can be used to emulate ML-style references.

```
let ref x =
  new ref @ x with
    operation lookup () @ s -> (s, s)
    operation update s' @ _ -> ((), s')
  end

let (!) r = r#lookup ()
let (:=) r v = r#update v
```

In Koka references are annotated with a heap parameter.

```
fun f() { var x := ref(10); x }
f : forall<h> () -> ref<h, int>
```

Note that values cannot have an effect, so we cannot create a global reference. So Koka cannot emulate ML-style references entirely.

```
> val x = ref(1)
        ^
((4), 5): error: effects do not match
  context         : val x = ref(1)
  term            :     x
  inferred effect: <alloc<_h>|_e>
  expected effect: total
  because         : Values cannot have an effect
```

6.5 Indexed effects

Similar to indexed monad one might like to have indexed effects. For example it can be perfectly safe to change the type in the *state* effect with the *set* operation, every *get* operation after the *operation* will then return a value of this new type. This gives a more general *state* effect. Furthermore we would like a version of typestate, where operations can only be called with a certain state and operations can also change the state. For example closing a file handle can only be done if the file handle is in the *open* state, after which this

state is changed to the *closed* state. This allows for encoding state machines on the type-level, which can be checked statically reducing runtime errors.

Only the effects library Idris supports this feature.

```
data State : Effect where
  Get : { a } State a
  Put : b -> { a ==> b } State ()

STATE : Type -> EFFECT
STATE t = MkEff t State

instance Handler State m where
  handle st Get k = k st st
  handle st (Put n) k = k () n

get : { [STATE x] } Eff x
get = call Get

put : y -> { [STATE x] ==> [STATE y] } Eff ()
put val = call (Put val)
```

Note that the *Put* operation changes the type from a to b. The *put* helper function also shows this in the type signature (going from $STATE\ x$ to $STATE\ y$).

Bibliography

[1] Bauer, Andrej, and Matija Pretnar. “An effect system for algebraic effects and handlers.” International Conference on Algebra and Coalgebra in Computer Science. Springer, Berlin, Heidelberg, 2013.

[2] Bauer, Andrej, and Matija Pretnar. ”Programming with algebraic effects and handlers.” Journal of Logical and Algebraic Methods in Programming 84.1 (2015): 108-123.

[3] Bauer, Andrej, and Matija Pretnar. ”An effect system for algebraic effects and handlers.” International Conference on Algebra and Coalgebra in Computer Science. Springer, Berlin, Heidelberg, 2013.

[4] Hillerström, Daniel, and Sam Lindley. ”Liberating effects with rows and handlers.” Proceedings of the 1st International Workshop on Type-Driven Development. ACM, 2016.

[5] Leijen, Daan. ”Type directed compilation of row-typed algebraic effects.” POPL. 2017.

[6] Lindley, Sam, Conor McBride, and Craig McLaughlin. ”Do Be Do. In: POPL’2017. ACM, New York, pp. 500-514. ISBN 9781450346603, http://dx. doi. org/10.1145/3009837.3009897.”

[7] Brady, Edwin. ”Programming and Reasoning with Side-Effects in IDRIS.” (2014).

[8] Levy, PaulBlain, John Power, and Hayo Thielecke. ”Modelling environments in call-by-value programming languages.” Information and computation 185.2 (2003): 182-210.

[9] Flanagan, Cormac, et al. "The essence of compiling with continuations." ACM Sigplan Notices. Vol. 28. No. 6. ACM, 1993.