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

Algebraic effects and handlers is a structured way to introduce side-effects to a programming language. The basic idea is that side-effects can be described by sets of operations, called the interface of the effect. Similar to exceptions, where exceptions can be thrown and caught. Operations can be called and "caught" by handlers. Different from exceptions however the handler also has access to a continuation which can be used to continue the computation at the point where the operation was called.

In this chapter we will introduce Algebraic effects and handlers through examples. Starting with simple algebraic effects and handlers (2.1). After we will continue with static instances (2.2) and ending with dynamic instances (2.3). The examples are written in an hypothetical functional programming language with algebraic effects and handlers with syntax similar to the Haskell programming language.

2.1 Algebraic effects and handlers

We will start with the familiar exceptions. We define an `Exc` effect interface with a single operation `throw`.

```
effect Exc {
  throw : String -> Void
```

```
}
```

For each operation in an effect definition we specify a parameter type (on the left of the arrow) and a return type (on the right of the arrow). In the case of `throw` the parameter type is a `String` (an error message) and the return type is `Void`, a type without any inhabitants.

We can now write functions that use the `Exc` effect. For example the following function `safeDiv` which will throw an error if the right argument is 0. We assume here that `Void` is equal to any type.

```
safeDiv : Int -> Int -> Int!{Exc}
safeDiv a b =
  if b == 0 then
    throw "division by zero!"
  else
    return a / b
```

We can call this function like any other function, but no computation will actually be performed. The effect will remain abstract, we still need to give them a semantics.

```
result : Int!{Exc}
result = 10 / 0
```

In order to actually "run" the effect we will need to handle the operations of that effect. For example, for `Exc` we can write a handler that returns 0 as a default value if an exception is thrown.

```
result : Int
result = handle {
  throw err k -> return 0
  return v -> return v
} (10 / 0) -- results in 0
```

For each operation we write a corresponding case in the handler, where we have access to the argument given at operation call and a continuation, which expects a value of the return type of the operation. There is also a case for values `return`, which gets as an argument the final value of a computation and has the opportunity to modify this value or to do some final computation. In this case we simply ignore the continuation and exit the computation early with a 0, we also return any values without modification.

We can give multiple ways of handling the same effect. For example we can also handle the `Exc` effect by capturing the failure or success in a sum type `Either`.

```
data Either a b = Left a | Right b

result : Either String Int
result = handle {
  throw err k -> return (Left err)
  return v -> return (Right v)
} (10 / 0) -- results in (Left "division by zero!")
```

Here we return early with `Left err` if an error is thrown, otherwise we wrap the resulting value using the `Right` constructor.

Another effect we might be interested in is non-determinism. To model this we define the `Flip` effect interface which has a single operation `flip`, which returns a boolean when called with the unit value.

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

Using the `flip` operation and if-expression we can write non-deterministic computations that can be seen as computation trees where `flip` branches the tree off in to two subtrees. The following program `choose123` non-deterministically returns either a 1, 2 or 3.

```
choose123 : () -> Bool!{Flip}
choose123 () =
  b1 <- flip ();
  if b1 then
    return 1
  else
    b2 <- flip ();
    if b2 then
      return 2
    else
```

```
    return 3
```

Here the syntax `(x <- c1; c2)` sequences the computations `c1` and `c2` by first performing `c1` and then performing `c2`, where the return value of `c1` can accessed in `x`.

Again `choose123` does not actually perform any computation when called, because we have yet to give it a semantics. We could always return `True` when a `flip` operation is called, in the case of `choose123` this will result in the first branch being picked returning 1 as the answer.

```
result : Int
result = handle {
  flip () k -> k True
  return v -> return v
} (choose123 ()) -- returns 1
```

Another handler could try all branches returning the greatest integer of all possibilities.

```
maxresult : Int
maxresult = handle {
  flip () k ->
    vtrue <- k True;
    vfalse <- k False;
    return (max vtrue vfalse)
  return v -> return v
} (choose123 ()) -- returns 3
```

Here we first call the continuation `k` with `True` and then with `False`. The we return the maximum between those results.

We could even collect the values from all branches by returning a list.

```
allvalues : List Int
allvalues = handle {
  flip () k ->
    vtrue <- k True;
    vfalse <- k False;
    return vtrue ++ vfalse
  return v -> return [v]
```

```
} (choose123 ()) -- returns [1, 2, 3]
```

Again we call the continuation `k` twice, but we append the two results instead. For the `return` base case we simply wrap the value in a singleton list.

Algebraic effects have the nice property that they combine easily. For example by combining the `Exc` and `Flip` we can implement backtracking, where the choose the first non-failing branch from a computation. For example we can write a function which returns all even sums of the numbers 1 to 3 by reusing `choose123`.

```
evensums123 : Int!{Flip, Exc}
evensums123 () =
  n1 <- choose123 ();
  n2 <- choose123 ();
  sum <- return (n1 + n2);
  if sum % 2 == 0 then
    return sum
  else
    throw "not even!"
```

We implement backtracking in `backtrack` by handling both the `flip` and `throw` operations. For `flip` and the `return` case we do the same as in `allvalues`, calling the continuation `k` with both `True` and `False` and appending the results together. For `throw` we ignore the error message and continuation and exit early with the empty list, this means that branches that results in a failure will not actually return any values.

```
backtrack : List Int
backtrack () = handle {
  flip () k ->
    vtrue <- k True;
    vfalse <- k False;
    return vtrue ++ vfalse
  throw msg k -> return []
  return v -> return [v]
} (evensums123 ()) -- returns [2, 4, 4, 6]
```

We can also handle the effects independently of each other. For example we could implement a partial version of `backtrack` that only handles the `Flip` effect.

```
partlybacktrack : (List Int)!{Exc}
partlybacktrack () = handle {
  flip () k ->
    vtrue <- k True;
    vfalse <- k False;
    return vtrue ++ vfalse
  return v -> return [v]
} (evensums123 ())
```

Now we can factor out the `throw` handler into its own function.

```
fullbacktrack : List Int
fullbacktrack () = handle {
  throw msg k -> return []
  return v -> return v
} (partlybacktrack ()) -- returns [2, 4, 4, 6]
```

Algebraic effects always commute, meaning the effects can be handled in any order. In the backtracking example the order of the handlers does not actually matter, but in general different orders could have different results.

Lastly we introduce the `State` effect, which allows us to implement local mutable state. We restrict ourselves to a state that consists of a single integer value, but in a language with parametric polymorphism a more general state effect could be written.

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

Our state effect has two operations, `get` and `put`. The `get` operation allows us to retrieve a value from the state and with the `put` operation we can change the value in the state.

We can now implement the familiar “post increment” operation as seen in the C programming language. This function retrieves the current value of the state, increments it by 1 and returns the previously retrieved value.

```
postInc : () -> Int!{State}
postInc () =
  x <- get ();
  put (x + 1);
  return x
```

To implement the semantics of the State effect we use parameter-passing similar to how the State monad is implemented in Haskell. We will abstract the implementation of the state handler in a function runState.

```
runState : (() -> Int!{State}) -> (Int -> (Int, Int))
runState comp = handle {
  get () k -> return (\s -> (f <- k s; return f s))
  put v k -> return (\s -> (f <- k (); return f v))
  return v -> return (\s -> return (s, v))
} (comp ())
```

runState takes as it's only argument a computation that returns an integer and may use the State effect. To delay the computation we take it as a function from unit. runState returns a function that takes the initial value of the state and returns a tuple of the final state and the return value of the computation. Let us take a look at the return case first, here we return a function that takes a state value and returns a tuple of this state and the return value. For the get case we return a function that takes a state value and runs the continuation k with this value, giving access to the state at the point where the get operation was called. From this continuation we get back another function, which we call with the current state, continuing the computation without changing the state. The put case is similar to the get but we call the continuation with the unit value and we continue the computation by calling f with the value giving with the put operation call.

Using state now is as simple as calling runState.

```
stateResult : () -> (Int, Int)
stateResult () =
  f <- runState postInc; -- returns a function taking the initial state
  f 42 -- post-increments 42 returning (43, 42)
```

Using the state effect we can implement imperative algorithms such as summing a range of numbers. We first implement a recursive function sumRangeRec which uses `State` to keep a running sum. After we define `sumRange` which calls `sumRangeRec` and runs the `State` effect with 0 as the initial value.

```
sumRangeRec : Int -> Int -> Int!{State}
sumRangeRec a b =
  if a > b then
    (_, result) <- get ();
    return result
  else
    x <- get ();
    put (x + a);
    sumRangeRec (a + 1) b

sumRange : Int -> Int -> Int
sumRange a b =
  f <- runState (\() -> sumRangeRec a b);
  f 0 -- initial sum value is 0
```

2.2 Static instances

2.3 Dynamic instances

Chapter 3

Background (Theory)

In this chapter we will show the basics of algebraic effects and handlers. We will start with the simply-typed lambda calculus (3.1) and add algebraic effects (3.2) and instances (3.3, 3.4) to it. We end with dynamic instances and show why a type system for them is difficult to implement.

3.1 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 3.1. 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 3.1: Syntax of the fine-grained lambda calculus

$$
\begin{array}{llr}
\nu ::= & & \text{(values)} \\
& x, y, z, k & \text{(variables)} \\
& \lambda x.c & \text{(abstraction)} \\
& () & \text{(unit value)} \\
c ::= & & \text{(computations)} \\
& \mathsf{return}\ \nu & \text{(return value as computation)} \\
& \nu\ \nu & \text{(application)} \\
& x \leftarrow c; c & \text{(sequencing)}
\end{array}
$$

variables that denote continuations later on. Lambda abstractions are denoted as $\lambda x.c$, note that the body c of the abstraction is restricted to be a computation as opposed to the ordinary lambda calculus where the body can be any expression. 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 $\mathsf{return}\ \nu$ 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 3.2. The relation $\rightsquigarrow$ is defined on computations, where the $c \rightsquigarrow c'$ means c reduces to

Figure 3.2: Semantics of the fine-grained lambda calculus

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

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

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

Figure 3.3: Types of the fine-grained simply-typed lambda calculus

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

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 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 $\rightsquigarrow$. Meaning that c in $c \rightsquigarrow^* c$ can reach c' in zero or more steps, while c in $c \rightsquigarrow c'$ reaches c' in exactly on step.

Types Next we give the *types* in Figure 3.3. Similar to the terms we split the syntax into value and computation types. Values are typed by value types

Figure 3.4: Typing rules of the fine-grained simply-typed lambda calculus

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

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

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

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

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

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

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

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

To give a more concrete example, take 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)
```

Here we use a lambda abstraction (`(\a b -> ())`) in order to simulate sequencing. Knowing the evaluation order is very important when evaluating the call `(printRange 1 10)`. In the expression `(\a b -> ()) (print a) (printRange (a +` the arguments can be either evaluated left-to-right or right-to-left, corresponding to (1) and (2) in the list above respectively. 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. A third option is to first evaluation `(print a)` then the call `(\a b -> ()) (print a)`, resulting in `(\b -> ()) (printRange (a + 1) b)`, after which this application is reduced. This corresponds to (3) in the list above, but has the same result as (1) in this example. From the syntax of the language we are not able to deduce which evaluation order will be used, even worse it may be left undefined in the language definition.

Translating the evaluation order corresponding to (1) 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. Because the fine-grained lambda calculus has explicit sequencing syntax we do not have to use lambda abstraction (`(\a b -> ())`) for this purpose.

Alternatively a translation that corresponds to evaluation order (2) results in:

```
-- 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.

Because we have eliminated the lambda abstraction there is no translation corresponding to (3), but semantically it would be identical to the first (left-to-right) translation.

Type soundness In order to prove type soundness for the previously defined calculus we first have define what it means for a computation to be a value. We define a computation c to be a value if c is of the form $\mathsf{return}\ \nu$ for some value ν.

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

Using this definition we can state the following type soundness theorem for the fine-grained simply typed lambda calculus.

Theorem 1 (Type soundness)**.**

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

This states that given a well-typed computation c and taking some amount of steps then the resulting computation c' will be of either a value or another step can be taken. In other words the term will not get "stuck". Note that this is only true if the computation c is typed in the empty context. If the context is not empty then the computation could get stuck on free variables.

We can prove this theorem using the following 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)**.**

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

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-typed 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. Note again that the context has to be empty for the Progress lemma, again because the computation could get stuck on free variables. For the Preservation lemma the context can be anything however, since the operational semantics will not introduce any new free variables that are not already in the context.

3.2 Algebraic effects

3.2.1 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:

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

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:

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

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
```

3.2.2 Syntax

3.2.3 Semantics

3.2.4 Type system

3.2.5 Discussion/limitations

3.3 Static instances

3.3.1 Intro

Explain:

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

3.3.2 Syntax

3.3.3 Semantics

3.3.4 Type system

3.3.5 Examples

Show state with multiple static instances (references).

3.4 Dynamic instances (untyped)

3.4.1 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.

3.4.2 Syntax

3.4.3 Semantics

3.4.4 Examples

Show untyped examples.

- Local exceptions
- ML-style references

3.4.5 Type system (discussion, problems)

Show difficulty of implementing a type system for this.

Chapter 4

Type and effect system

4.1 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: $\{E_1@s_1, ..., E_n@s_n\}$.

Judgments There are three kinds of judgments: subtyping, well-formedness and typing. The subtyping judgments:

$$\tau <: \tau$$

$$\underline{\tau} <: \underline{\tau}$$

well-formedness judgments:

$$\Delta; \Sigma \vdash s$$

$$\Delta; \Sigma \vdash \tau$$

$$\Delta; \Sigma \vdash \underline{\tau}$$

Figure 4.1: 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)
$\mathsf{handle}(s_{var} \to c)$	(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 4.2: Subtyping

$$\frac{}{\mathsf{Inst}\ s\ \varepsilon <: \mathsf{Inst}\ s\ \varepsilon} \qquad \frac{\tau_2 <: \tau_1 \qquad \underline{\tau}_1 <: \underline{\tau}_2}{\tau_1 \to \underline{\tau}_1 <: \tau_2 \to \underline{\tau}_2}$$

$$\frac{\underline{\tau}_1 <: \underline{\tau}_2}{\forall s_{var}.\underline{\tau}_1 <: \forall s_{var}.\underline{\tau}_2} \qquad \frac{\tau_1 <: \tau_2 \qquad r_1 \subseteq r_2}{\tau_1\ !\ r_1 <: \tau_2\ !\ r_2}$$

Figure 4.3: Well-formedness

$$\frac{s_{var} \in \Delta}{\Delta; \Sigma \vdash s_{var}} \qquad \frac{s_{loc} \in \Sigma}{\Delta; \Sigma \vdash s_{loc}}$$

$$\frac{\Delta; \Sigma \vdash s}{\Delta; \Sigma \vdash \mathsf{Inst}\ s\ \varepsilon} \qquad \frac{\Delta; \Sigma \vdash \tau \qquad \Delta; \Sigma \vdash \underline{\tau}}{\Delta; \Sigma \vdash \tau \to \underline{\tau}}$$

$$\frac{\Delta, s_{var}; \Sigma \vdash \underline{\tau}}{\Delta; \Sigma \vdash \forall s_{var}.\underline{\tau}} \qquad \frac{\Delta; \Sigma \vdash \tau \qquad \forall(\varepsilon @ s \in r) \Rightarrow \Delta; \Sigma \vdash s}{\Delta; \Sigma \vdash \tau\ !\ r}$$

typing judgments:

$$\Delta; \Sigma; \Gamma \vdash \nu : \tau$$

$$\Delta; \Sigma; \Gamma \vdash c : \underline{\tau}$$

$$\Delta; \Sigma; \Gamma \vdash^{\tau} h : \underline{\tau}$$

Figure 4.4: Value typing rules

$$\frac{\Gamma[x] = \tau}{\Delta; \Sigma; \Gamma \vdash x : \tau} \qquad \frac{\Sigma(l) = (s_{loc}, \varepsilon)}{\Delta; \Sigma; \Gamma \vdash \mathsf{inst}(l) : \mathsf{Inst}\ s_{loc}\ \varepsilon} \qquad \frac{\Delta; \Sigma; \Gamma, x : \tau \vdash c : \underline{\tau}}{\Delta; \Sigma; \Gamma \vdash \lambda x.c : \tau \to \underline{\tau}}$$

$$\frac{\Delta, s_{var}; \Sigma; \Gamma \vdash c : \underline{\tau}}{\Delta; \Sigma; \Gamma \vdash \Lambda s_{var}.c : \forall s_{var}.\underline{\tau}} \qquad \frac{\Delta; \Sigma; \Gamma \vdash \nu : \tau_1 \qquad \Delta; \Sigma \vdash \tau_2 \qquad \tau_1 <: \tau_2}{\Delta; \Sigma; \Gamma \vdash \nu : \tau_2}$$

4.2 Subtyping

4.3 Well-formedness

4.4 Typing rules

4.5 Semantics

4.6 Evaluation Contexts

Figure 4.5: Computation typing rules

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

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

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

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

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

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

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

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

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

Figure 4.6: 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 4.7: Semantics

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

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

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

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

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

Figure 4.8: Semantics of new handlers

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

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

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

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

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

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

Figure 4.9: Semantics of instance handlers

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

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

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

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

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

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

Figure 4.10: Evaluation Contexts

$$
\begin{array}{lllr}
E ::= & & & \text{(computation contexts)} \\
& \square & & \text{(hole)} \\
& x \leftarrow E;\ c & & \text{(sequencing)} \\
& \mathsf{handle}^{s}(E) & & \text{(computation handler)} \\
& \mathsf{handle}^{l}\{h\}(E) & & \text{(instance handler)} \\
H^{s} ::= & & & \text{(handler contexts)} \\
& \square & & \text{(hole)} \\
& x \leftarrow H^{s};\ c & & \text{(sequencing)} \\
& \mathsf{handle}^{s'}(H^{s}) & (s \neq s') & \text{(computation handler)} \\
& \mathsf{handle}^{l}\{h\}(H^{s}) & & \text{(instance handler)} \\
H^{l} ::= & & & \text{(instance handler contexts)} \\
& \square & & \text{(hole)} \\
& x \leftarrow H^{l};\ c & & \text{(sequencing)} \\
& \mathsf{handle}^{s}(H^{l}) & & \text{(computation handler)} \\
& \mathsf{handle}^{l'}\{h\}(H^{l}) & (l \neq l') & \text{(instance handler)}
\end{array}
$$

Chapter 5

Formalization

Chapter 6

Related work

Chapter 7

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

7.1 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', ())
}
```

7.2 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.*

7.3 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)
```

7.4 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
```

7.5 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.