Type systems for dynamic instances for algebraic effects and handlers

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

2.1 Algebraic effects

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

theory?, Category theory, algebras, forget about equations

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.

2.2 Handlers

Exception handlers can be generalized by also supplying a continuation to the programmer. The programmer can then decide to continue at the point where the exception was thrown. These generalized exception handlers were further generalized by Plotkin and Pretnar to allow for many different effects.

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.

citations, example of interpretation, more examples? (effects)

2.3 Instances

One problem that occurs with the basic algebraic effects system described above is when we try to

2.4 Dynamic instances

2.5 Resources

2.6 Effect systems for algebraic effects

Chapter 3

Type and effect system

3.1 Description

We extend a simplified version of the type system described by Bauer and Pretnar in [1]. We simplify the system from that paper by removing static instances and always having handlers contain all operations from a single effect. We extend the system with instance type variables, universally quantified value types, existential computation types and a computation to dynamically create instances. For the effect annotations on the computation types (called the dirt in [1]) we will take sets of instance variables.

3.2 Syntax

The syntax stays the same as in [1], but we add dynamic instance creation and instance values. Instance values represent a certain index at run-time, indexed by an $n \in \mathbb{N}$. They are used to define the semantics of the system but would not appear in an user-facing language. The only way to get an instance value is by using the *new* ε computation. 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}. Annotations r are sets of instance variables.

$\tau ::=$	(value types)
i, j, k	(instance variables)
$()$	(unit type)
$\tau \to \underline{\tau}$	(type of functions)
$\underline{\tau} \Rightarrow \underline{\tau}$	(type of handlers)
$\forall(i : \varepsilon).\tau$	(universally quantified value type)
$\underline{\tau} ::=$	(computation types)
$\tau \,!\, r$	(annotated type)
$\exists(i : \varepsilon).\underline{\tau}$	(existential)
$\nu ::=$	(values)
x, y, z, k	(variables)
$()$	(unit value)
$\mathsf{inst}(n)$	(instance values)
$\lambda x.c$	(abstraction)
$\mathit{handler}(\nu)\ \{\mathit{return}\ x \to c, \mathit{op}_1(x; k) \to c, ..., \mathit{op}_n(x; k) \to c\}$	(handler)
$\Lambda(i : \varepsilon).\nu$	(instance type variable abstraction)
$\nu\ [i]$	(type application)
$c ::=$	(computations)
$\mathit{return}\ \nu$	(return value as computation)
$\nu\ \nu$	(application)
$x \leftarrow c; c$	(sequencing)
$(i, x) \leftarrow c; c$	(existential unpack)
$\mathit{with}\ \nu\ \mathit{handle}\ c$	(handler application)
$\nu \# \mathit{op}(\nu; y.c)$	(operation call)
$\mathit{new}\ \varepsilon$	(instance creation)

3.3 Subtyping rules

The subtyping rules are mostly the same as the rules described in [1]. These are as you would expect.

$$\frac{}{() <: ()} \qquad \frac{a' <: a \quad b <: b'}{a \to b <: a' \to b'}$$

$$\frac{a' <: a \quad b <: b'}{a \Rightarrow b <: a' \Rightarrow b'} \qquad \frac{a <: a' \quad e \subseteq e'}{a \,!\, e <: a' \,!\, e'}$$

We add rules for instance variables and existentials, which we compare structurally.

$$\frac{}{i <: i} \qquad \frac{a <: b}{\exists(i : \varepsilon).a <: \exists(i : \varepsilon).b}$$

For existentials we are allowed to remove one if the instance variable does not appear in the type (where $FIV(a)$ is the set of free instance variables of a). We are also allowed to swap two existentials.

$$\frac{a <: b \quad i \notin FIV(a)}{\exists(i : \varepsilon).a <: b} \qquad \frac{}{\exists(i : \varepsilon).\exists(j : \varepsilon').a <: \exists(j : \varepsilon').\exists(i : \varepsilon).a}$$

We can prove reflexivity and transitivity of the subtyping relation from these rules.

Theorem 1 (Subtyping reflexivity)**.** *for all value and computation types a, $a <: a$*

Theorem 2 (Subtyping transitivity)**.** *for all value and computation types a, b and c, if $a <: b$ and $b <: c$ then $a <: c$*

3.4 Well-formedness judgement

We have a well-formedness judgement for both value and computation types $\Delta \vdash \tau$ and $\Delta \vdash \underline{\tau}$. Where Δ stores bindings of instance variables to effects. For instance variables we simply check that they occur in Δ. For computation types we check that all the variables in the annotation occur in Δ.

$$\frac{}{\Delta \vdash ()} \qquad \frac{(i : \varepsilon) \in \Delta}{\Delta \vdash i} \qquad \frac{\Delta \vdash a \quad \Delta \vdash b}{\Delta \vdash a \to b} \qquad \frac{\Delta \vdash a \quad \Delta \vdash b}{\Delta \vdash a \Rightarrow b}$$

$$\frac{\Delta \vdash a \quad \Delta \vdash j_i}{\Delta \vdash a \,!\, \{j_0, ..., j_n\}} \qquad \frac{\Delta, i : \varepsilon \vdash a}{\Delta \vdash \forall (i : \varepsilon).a}$$

$$\frac{\Delta, i : \varepsilon \vdash a}{\Delta \vdash \exists (i : \varepsilon).a}$$

3.5 Typing rules

For the typing rules there are two judgments, $\Delta; \Gamma \vdash \nu : \tau$ for assigning types to values and $\Delta; \Gamma \vdash c : \underline{\tau}$ for assigning computation types to computations. Γ stores bindings of variables to types and Δ stores bindings of instance variables to effects ε. In the sub-typing and abstraction rules we check that the introduced types are well-formed, so that we don't introduce ill-formed types in the context.

T-SubVal **T-Var** **T-Unit** **T-Abs**

$$\frac{\Delta; \Gamma \vdash \nu : \tau_1 \quad \Delta \vdash \tau_2 \quad \tau_1 <: \tau_2}{\Delta; \Gamma \vdash \nu : \tau_2} \qquad \frac{(x : \tau) \in \Gamma}{\Delta; \Gamma \vdash x : \tau} \qquad \frac{}{\Delta; \Gamma \vdash () : ()} \qquad \frac{\Delta \vdash \tau_1 \quad \Delta; \Gamma, x : \tau_1 \vdash c : \underline{\tau_2}}{\Delta; \Gamma \vdash \lambda x.c : \tau_1 \to \underline{\tau_2}}$$

In the following rule
$h = handler(\nu)\ \{return\ x_r \to c_r, op_1(x_1; k_1) \to c_1, ..., op_n(x_n; k_n) \to c_n\}$.
T-Handler

$$\frac{\begin{array}{l}\Delta; \Gamma \vdash \nu : i \\ (i : \varepsilon) \in \Delta \\ O_\varepsilon = \{op_1, ..., op_n\} \\ \Delta \vdash \tau_1 \\ \Delta; \Gamma, x_r : \tau_1 \vdash c_r : \tau_2 \,!\, r_2 \\ \Delta; \Gamma, x_i : \tau^1_{op_i}, k_i : \tau^2_{op_i} \to \tau_2 \,!\, r_2 \vdash c_i : \tau_2 \,!\, r_2 \\ r_1 \backslash \{i\} \subseteq r_2\end{array}}{\Delta; \Gamma \vdash h : \tau_1 \,!\, r_1 \Rightarrow \tau_2 \,!\, r_2}$$

For the handlers we first check that value ν is bound to an instance variable i of effect ε. We check that the operations in the handlers are exactly the operations belonging to ε. After we check that all the cases in the handler agree on the return type and annotations. We have the condition $r_1 \backslash \{i\} \subseteq r_2$ to make sure that any extra effects before the handler remain unhandled after the handler is evaluated. We check that the types on the left and right sides of the handler type are well-formed to make sure that no instance variables are introduced that are not in the context Δ.

Explain...
T-AbsT

$$\frac{\Delta, i : \varepsilon; \Gamma \vdash \nu : \tau}{\Delta; \Gamma \vdash \Lambda(i : \varepsilon).\nu : \forall(i : \varepsilon).\tau}$$

T-AppT

$$\frac{\begin{array}{l}\Delta; \Gamma \vdash \nu : \forall(i : \varepsilon).\tau \\ (j : \varepsilon) \in \Delta\end{array}}{\Delta; \Gamma \vdash \nu\ [j] : \tau[j/i]}$$

T-SubComp

$$\frac{\begin{array}{l}\Delta; \Gamma \vdash c : \underline{\tau}_1 \\ \Delta \vdash \underline{\tau}_2 \\ \underline{\tau}_1 <: \underline{\tau}_2\end{array}}{\Delta; \Gamma \vdash c : \underline{\tau}_2}$$

T-Return

$$\frac{\Delta; \Gamma \vdash \nu : \tau}{\Delta; \Gamma \vdash return\ \nu : \tau \,!\, \varnothing}$$

T-App

$$\frac{\Delta;\Gamma \vdash \nu_1 : \tau_1 \to \underline{\tau}_2 \quad \Delta;\Gamma \vdash \nu_2 : \tau_1}{\Delta;\Gamma \vdash \nu_1\ \nu_2 : \underline{\tau}_2}$$

T-Handle

$$\frac{\Delta;\Gamma \vdash \nu : \underline{\tau}_1 \Rightarrow \underline{\tau}_2 \quad \Delta;\Gamma \vdash c : \underline{\tau}_1}{\Delta;\Gamma \vdash \mathit{with}\ \nu\ \mathit{handle}\ c : \underline{\tau}_2}$$

Instances are checked to be in Δ.

T-Instance

$$\frac{(i : \varepsilon) \in \Delta}{\Delta;\Gamma \vdash \mathsf{inst}(i) : i}$$

For the operation calls we check that the value ν_1 is bound to an instance variable i of effect ε. We check that the operation belongs to ε and that the value ν_2 is of the parameter type of the operation. We then typecheck the continuation c and make sure that the instance variable i is in the annotation on the type of c.

For the creation of instances we return an existential type to account for the newly created instance.

T-Op

$$\frac{\begin{array}{l}\Delta;\Gamma \vdash \nu_1 : i \\ (i : \varepsilon) \in \Delta \\ \mathit{op} \in O_\varepsilon \\ \Delta;\Gamma \vdash \nu_2 : \tau^1_{\mathit{op}} \\ \Delta;\Gamma, y : \tau^2_{\mathit{op}} \vdash c : \exists \overrightarrow{j}.\tau\ !\ r \\ i \in r\end{array}}{\Delta;\Gamma \vdash \nu_1 \# \mathit{op}(\nu_2; y.c) : \exists \overrightarrow{j}.\tau\ !\ r}$$

T-New

$$\frac{}{\Delta;\Gamma \vdash \mathit{new}\ \varepsilon : \exists (i : \varepsilon).i\ !\ \varnothing}$$

Write here.

T-Do

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

We can unpack an existential type and get a reference to an instance and an instance type in the continuing computation. If the instance variable occurs free in the type of the resulting computation then we have to re-wrap the existential around it. Else we can leave out the existential, since the instance variable does not occur free in the type.

T-Unpack1

$$\frac{\Delta;\Gamma \vdash c_1 : \exists(i : \varepsilon).\underline{\tau}_1 \qquad \Delta, i : \varepsilon;\Gamma, x : i \vdash c_2 : \underline{\tau}_2 \qquad i \notin \underline{\tau}_2}{\Delta;\Gamma \vdash (i, x) \leftarrow c_1; c_2 : \underline{\tau}_2}$$

T-Unpack2

$$\frac{\Delta;\Gamma \vdash c_1 : \exists(i : \varepsilon).\underline{\tau}_1 \qquad \Delta, i : \varepsilon;\Gamma, x : i \vdash c_2 : \underline{\tau}_2 \qquad i \in \underline{\tau}_2}{\Delta;\Gamma \vdash (i, x) \leftarrow c_1; c_2 : \exists(i : \varepsilon).\underline{\tau}_2}$$

3.6 Semantics

We give the small-step operational semantics for the system. These are the same as the one in [1] but with the instance creation computation added. In order to handle instance creation we update the relation to $c; n \rightsquigarrow c; n$ where c is the current computation and $n \in \mathbb{N}$ the id of the next instance that will be created.

For abstractions we have the usual beta-reduction rule.

$$\frac{}{(\lambda x.c)\ \nu\ ;\ i \rightsquigarrow c[\nu/x]\ ;\ i}$$

For instance creation we replace the call to *new* with the instance constant and we increase the instance id counter.

$$\frac{}{new\ E\ ;\ i \rightsquigarrow return\ \mathsf{inst}(i)\ ;\ i + 1}$$

For sequencing we have three rules. The first rule reduces the first computation. The second rule substitutes the value of a return computation in the

second computation. The last rule floats an operation call over the sequencing, this makes the handle computation easier, since we won't have to think about sequencing inside of a handle computation.

$$\frac{c_1 \ ; \ i \rightsquigarrow c_1' \ ; \ i'}{(x \leftarrow c_1; c_2) \ ; \ i \rightsquigarrow (x \leftarrow c_1'; c_2) \ ; \ i'}$$

$$\frac{}{(x \leftarrow \mathit{return}\, \nu; c) \ ; \ i \rightsquigarrow c[\nu/x] \ ; \ i}$$

$$\frac{}{(x \leftarrow \mathsf{inst}(j)\#\mathit{op}(\nu; y.c_1); c_2) \ ; \ i \rightsquigarrow \mathsf{inst}(j)\#\mathit{op}(\nu; y.(x \leftarrow c_1; c_2)) \ ; \ i}$$

For unpacking we can turn it in to do sequencing by just ignoring the type parameter.

$$\frac{}{(i, x) \leftarrow c_1; c_2 \ ; \ i \rightsquigarrow x \leftarrow c_1; c_2 \ ; \ i}$$

In the following rules
$h = \mathit{handler}(\mathsf{inst}(j))\ \{\mathit{return}\, x_r \to c_r, \mathit{op}_1(x_1; k_1) \to c_1, ..., \mathit{op}_n(x_n; k_n) \to c_n\}$.
We can reduce the computation that we want to handle.

$$\frac{c \ ; \ i \rightsquigarrow c' \ ; \ i'}{\mathit{with}\ h\ \mathit{handle}\ c \ ; \ i \rightsquigarrow \mathit{with}\ h\ \mathit{handle}\ c' \ ; \ i'}$$

If we are handling a return computation we simply substitute the value in the return case of the handler.

$$\frac{}{\mathit{with}\ h\ \mathit{handle}\ (\mathit{return}\, \nu) \ ; \ i \rightsquigarrow c_r[\nu/x_r] \ ; \ i}$$

If we are handling an operation call where the instance matches the instance of the handler and the operation is in the handler then we can reduce to the corresponding operation clause with the parameter value and the continuation substituted. Note that we nest the handle computation inside of the continuation, this describes deep handlers.

$$\frac{\mathit{op}_i \in \{\mathit{op}_1, ..., \mathit{op}_n\}}{\mathit{with}\ h\ \mathit{handle}\ (\mathsf{inst}(j)\#\mathit{op}_i(\nu; x.c)) \ ; \ i \rightsquigarrow c_i[\nu/x_i, (\lambda x.\mathit{with}\ h\ \mathit{handle}\ c)/k_i] \ ; \ i}$$

If the instance is different from the instance in the handler or the operation is not in the handler then we float the operation call over the handling computation.

$$\frac{op \notin \{op_1, ..., op_n\} \vee k \neq j}{with\ h\ handle\ (\mathsf{inst}(k)\#op(\nu; x.c))\ ;\ i \rightsquigarrow \mathsf{inst}(k)\#op(\nu; x.with\ h\ handle\ c)\ ;\ i}$$

3.7 Theorems

The most important theorem is the type soundness theorem, which states that any computation that typechecks will not get stuck. This is proven using two auxiliary theorems preservation and progress.
The preservation theorem states that for any program that typechecks, if we take a step using the semantics then the resulting program will still typecheck with the same type.
Theorem 3 (Preservation)**.** *If* $\Delta; \Gamma \vdash c : \underline{\tau}$ *and* $c \rightsquigarrow c'$ *then* $\Delta; \Gamma \vdash c' : \underline{\tau}$

The progress theorem states that any computation that typechecks is either able to take a step, is a value.
In order to proof this we have the auxiliary theorem for effectful computation where the evaluation can get stuck on operation calls.
Theorem 4 (Progress effectful)**.** *If* $\cdot; \cdot \vdash c : \underline{\tau}$ *then either:*

- $c \rightsquigarrow c'$ *for some* c'
- $c = \mathit{return}\, \nu$ *for some* ν
- $c = \mathsf{inst}(i)\#op(\nu; x.c)$ *for some* i, op, ν *and* c

Note that the progress theorem requires that the computation has no effects ($!\varnothing$), else the computation could get stuck on an operation call.
Theorem 5 (Progress)**.** *If* $\cdot; \cdot \vdash c : \exists \overrightarrow{i}.\tau \,!\, \varnothing$ *then either:*

- $c \rightsquigarrow c'$ *for some* c'
- $c = \mathit{return}\, \nu$ *for some* ν

Using preservation and progress we can proof type soundness (also called type safety). Here also we require that the computation has no effects.
Theorem 6 (Type soundness)**.** *If* $\cdot; \cdot \vdash c : \exists \overrightarrow{i}.\tau \,!\, \varnothing$ *and* $c, i \rightsquigarrow^* c', i'$ *(where* $c', i' \not\rightsquigarrow c'', i''$*) then* $c' = \mathit{return}\, \nu$ *for some* ν

The determinism theorem states that any computation has a single evaluation path.
Theorem 7 (Determinism)**.** *If* $c \rightsquigarrow c'$ *and* $c \rightsquigarrow c''$ *then* $c' = c''$

3.8 Examples

We will show some examples together with the types that the discussed type system will assign to them. After each example is a fitch-style type derivation. For the examples we will assume that the following effect is in the context.

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

The following function f creates a new instances and calls an operation on it, but does not return the instance itself. In the type we have an existential but we see that the instance does not appear in the value type, but only in the effect annotation. This way we know that we do not have access to the instance and so are unable to handle the effect.

```
f : () -> exists (i:Flip). Bool!{i}
f _ =
  inst <- new Flip;
  inst#flip ()
```

1 | $\vdash \_ : ()$

2 | $new\ Flip : \exists(j : Flip).i\ !\ \varnothing$

3 | $i : E \mid inst : i \vdash$

4 | $\mid x : Bool \vdash$

5 | $return\ x : Bool\ !\ \varnothing$ T-Return, 4

6 | $return\ x : Bool\ !\ \{i\}$ T-SubComp, 3, 5

7 | $inst\#flip((); x.return\ x) : Bool\ !\ \{i\}$ T-Op, 6

8 | $inst \leftarrow new\ Flip; inst\#flip((); x.return\ x) : \exists(i : Flip).Bool\ !\ \{i\}$ T-Do, 2, 7

9 | $\lambda\_.inst \leftarrow new\ Flip; inst\#flip((); x.return\ x) : () \to \exists(i : Flip).Bool\ !\ \{i\}$ T-Abs, 8

In the following function g we create a new instance, call an operation on it and then immediately handle this effect. The type of this function is pure,

since the effect is immediately handled.

```
g : () -> Bool!{}
g _ =
  inst <- new Flip;
  with handler(inst) {
    flip _ k -> k True
  } handle inst#flip ()
```

The following function is also pure because although an instance is created, this instance is never used or returned.

```
g' : () -> ()!{}
g' _ =
  inst <- new Flip;
  return ()
```

1 | | $\mid \_ : ()$

2 | | $new\ Flip : \exists(j : Flip).i\ !\ \varnothing$

3 | | | $i : E \mid inst : i$

4 | | | $() : ()$

5 | | | $return\ () : ()\ !\ \varnothing$ — T-Return, 4

6 | | $inst \leftarrow new\ Flip; return\ () : \exists(i : Flip).()\ !\ \varnothing$ — T-Do, 2, 5

7 | | $inst \leftarrow new\ Flip; return\ () : ()\ !\ \varnothing$ — T-ExistsRemove, 6

8 | $\lambda\_.inst \leftarrow new\ Flip; return\ () : () \to ()\ !\ \varnothing$ — T-Abs, 7

The following function simply wraps the creation of a *Flip* instance.

```
h' : () -> exists (i:Flip). i!{}
h' _ =
  inst <- new Flip;
  return inst
```

In h'' we use h' twice. This means that we create two distinct *Flip* instances and we also expect two instance variables bound in the type.

```
h'' : () -> exists (i:Flip) (j:Flip). (i, j)!{}
h'' _ =
  i1 <- h' ();
  i2 <- h' ();
  return (i1, i2)
```

In the following function *nested* we create a new instance and handle it, but in the computation we want to handle another instance is created and used. Only one operation call will be handled by the handler, the one called on $i1$. The operation call made on $i2$ is unhandled and so we are left with one existential quantifier.

```
nested : () -> exists (i:Flip). Bool!{i}
nested _ =
  i1 <- new Flip;
  with handler(i1) {
    flip _ k -> k True
  } handle (
    i2 <- new Flip;
    x <- i1#flip ();
    y <- i2#flip ();
    return (x && y)
  )
```

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).