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 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 Fine-grained simply-typed lambda calculus

2.1.1 Intro

Explain:

- Why fine-grained
  - effects require precise order of evaluation
  - fine-grain is explicit on order of evaluation
  - evaluation only happens in computations
- difference between values and computations
- subsume call-by-value and call-by-name (by using thunks)

2.1.2 Syntax

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

2.1.3 Semantics

$$\frac{}{(\lambda x.c)\ \nu \rightsquigarrow c[x := \nu]}$$

$$\frac{}{x \leftarrow \mathit{return}\ \nu; c \rightsquigarrow c[x := \nu]}$$

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

2.1.4 Type system

value types and computation types

$$
\begin{array}{ll}
\tau ::= & \text{(value types)} \\
\quad () & \text{(unit type)} \\
\quad \tau \rightarrow \underline{\tau} & \text{(type of functions)} \\
\underline{\tau} ::= & \text{(computation types)} \\
\quad \tau & \text{(value type)}
\end{array}
$$

$$\frac{\Gamma[x] = \tau}{\Gamma \vdash x : \tau}$$

$$\frac{}{\Gamma \vdash () : ()}$$

$$\frac{\Gamma, x : \tau_1 \vdash c : \tau_2}{\Gamma \vdash \lambda x.c : \tau_1 \to \tau_2}$$

$$\frac{\Gamma \vdash \nu : \tau}{\Gamma \vdash \mathit{return}\, \nu : \tau}$$

$$\frac{\begin{array}{l}\Gamma \vdash \nu_1 : \tau_1 \to \tau_2 \\ \Gamma \vdash \nu_2 : \tau_1\end{array}}{\Gamma \vdash \nu_1\ \nu_2 : \tau_2}$$

$$\frac{\begin{array}{l}\Gamma \vdash c_1 : \tau_1 \\ \Gamma, x : \tau_1 \vdash c_2 : \tau_2\end{array}}{\Gamma \vdash x \leftarrow c_1; c_2 : \tau_2}$$

2.1.5 Examples

Show explicit order of evaluation. Same application with different orders of evaluation.
$f\ x\ y$ translated to fine-grained:
Left-to-right call-by-value:
$x' \leftarrow x; y' \leftarrow y; g \leftarrow f\ x'; g\ y'$
Right-to-left call-by-value:
$y' \leftarrow y; x' \leftarrow x; g \leftarrow f\ x'; g\ y'$
Left-to-right call-by-value, apply f before evaluating y:
$x' \leftarrow x; g \leftarrow f\ x'; y' \leftarrow y; g\ y'$
Call-by-name:
$g \leftarrow f\ (\lambda\_.x); g\ (\lambda\_.y)$

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)

2.2.2 Syntax

2.2.3 Semantics

2.2.4 Type system

2.2.5 Examples

Show flip (non-determinism) and state examples.

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 a set of locations $l \in L$, each instance has a unique location. Annotations r are sets of instance variables.

$\tau ::=$	(value types)
$\quad i, j, k$	(instance variables)
$\quad ()$	(unit type)
$\quad \mathsf{Inst}\ i\ \varepsilon$	(instance type)
$\quad \tau \to \underline{\tau}$	(type of functions)
$\quad \forall i.\underline{\tau}$	(universally quantified type)
$\underline{\tau} ::=$	(computation types)
$\quad \tau\ !\ r$	(annotated type)
$\nu ::=$	(values)
$\quad x, y, z, k$	(variables)
$\quad ()$	(unit value)
$\quad \mathsf{inst}(l)$	(instance values)
$\quad \lambda x.c$	(abstraction)
$\quad \Lambda i.c$	(instance type variable abstraction)
$c ::=$	(computations)
$\quad \mathsf{return}\ \nu$	(return value as computation)
$\quad \nu\ \nu$	(application)
$\quad \nu\ [i]$	(type application)
$\quad x \leftarrow c;\ c$	(sequencing)
$\quad \nu \# op(\nu)$	(operation call)
$\quad \mathsf{fresh}\ i;\ c$	(fresh instance variable)
$\quad \mathsf{new}\ \varepsilon\ \mathsf{of}\ i\ \mathsf{as}\ x;\ c$	(instance creation)
$\quad \mathsf{handle}^{\nu}(c)\ \{\mathsf{return}\ x \to c, op_1(x;k) \to c, ..., op_n(x;k) \to c\}$	(handle computation)

3.2 Subtyping rules

$$\frac{}{i <: i}$$

$$\frac{}{() <: ()}$$

$$\frac{}{\mathsf{Inst}\ i\ \varepsilon <: \mathsf{Inst}\ i\ \varepsilon}$$

$$\frac{\begin{array}{c} a' <: a \\ b <: b' \end{array}}{a \to b <: a' \to b'}$$

$$\frac{a <: b}{\forall i.a <: \forall i.b}$$

$$\frac{\begin{array}{c} a <: a' \\ e \subseteq e' \end{array}}{a\ !\ e <: a'\ !\ e'}$$

3.3 Well-formedness judgement

$$\frac{i \in \Delta}{\Delta \vdash i}$$

$$\frac{}{\Delta \vdash ()}$$

$$\frac{\Delta \vdash i}{\Delta \vdash \mathsf{Inst}\ i\ \varepsilon}$$

$$\frac{\begin{array}{c} \Delta \vdash a \\ \Delta \vdash b \end{array}}{\Delta \vdash a \to b}$$

$$\frac{\Delta, i \vdash a}{\Delta \vdash \forall i.a}$$

$$\frac{\begin{array}{c} \Delta \vdash a \\ \Delta \vdash j_i \end{array}}{\Delta \vdash a\ !\ \{j_0, ..., j_n\}}$$

3.4 Typing rules

$$\frac{\begin{array}{l}\Delta; \Sigma; \Gamma \vdash \nu : \tau_1 \\ \Delta \vdash \tau_2 \\ \tau_1 <: \tau_2\end{array}}{\Delta; \Sigma; \Gamma \vdash \nu : \tau_2}$$

$$\frac{(x : \tau) \in \Gamma}{\Delta; \Sigma; \Gamma \vdash x : \tau}$$

$$\frac{}{\Delta; \Sigma; \Gamma \vdash () : ()}$$

$$\frac{\Sigma(l) = (i, \varepsilon)}{\Delta; \Sigma; \Gamma \vdash \mathsf{inst}(l) : \mathsf{Inst}\ i\ \varepsilon}$$

$$\frac{\begin{array}{l}\Delta \vdash \tau_1 \\ \Delta; \Sigma; \Gamma, x : \tau_1 \vdash c : \underline{\tau}_2\end{array}}{\Delta; \Sigma; \Gamma \vdash \lambda x.c : \tau_1 \to \underline{\tau}_2}$$

$$\frac{\Delta, i; \Sigma; \Gamma \vdash c : \underline{\tau}}{\Delta; \Sigma; \Gamma \vdash \Lambda i.c : \forall i.\underline{\tau}}$$

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

$$\frac{\Delta; \Sigma; \Gamma \vdash \nu : \tau}{\Delta; \Sigma; \Gamma \vdash \mathsf{return}\ \nu : \tau \mathbin{!} \varnothing}$$

$$\frac{\begin{array}{l}\Delta; \Sigma; \Gamma \vdash \nu_1 : \tau \to \underline{\tau} \\ \Delta; \Sigma; \Gamma \vdash \nu_2 : \tau\end{array}}{\Delta; \Sigma; \Gamma \vdash \nu_1\ \nu_2 : \underline{\tau}}$$

$$\frac{\begin{array}{l}\Delta \vdash j \\ \Delta; \Sigma; \Gamma \vdash \nu : \forall i.\underline{\tau}\end{array}}{\Delta; \Sigma; \Gamma \vdash \nu\ [j] : \underline{\tau}[i := j]}$$

$$\frac{\begin{array}{l}\Delta;\Sigma;\Gamma\vdash c_1 : \tau_1 \,!\, r\\ \Delta;\Sigma;\Gamma, x:\tau_1 \vdash c_2 : \tau_2 \,!\, r\end{array}}{\Delta;\Sigma;\Gamma\vdash x \leftarrow c_1;\; c_2 : \tau_2 \,!\, r}$$

$$\frac{\begin{array}{l}\Delta;\Sigma;\Gamma\vdash \nu_1 : \mathsf{Inst}\ i\ \varepsilon\\ op \in O_\varepsilon\\ \Delta;\Sigma;\Gamma\vdash \nu_2 : \tau^1_{op}\end{array}}{\Delta;\Sigma;\Gamma\vdash \nu_1\#op(\nu_2) : \tau^2_{op} \,!\, \{i\}}$$

$$\frac{\begin{array}{l}\Delta, i;\Sigma;\Gamma\vdash c : \underline{\tau}\\ i \notin \underline{\tau}\end{array}}{\Delta;\Sigma;\Gamma\vdash \mathsf{fresh}\ i;\; c : \underline{\tau}}$$

$$\frac{\begin{array}{l}\Delta\vdash i\\ \Delta;\Sigma;\Gamma, x : \mathsf{Inst}\ i\ \varepsilon \vdash c : \underline{\tau}\\ i \notin \underline{\tau}\end{array}}{\Delta;\Sigma;\Gamma\vdash \mathsf{new}\ \varepsilon\ \mathsf{of}\ i\ \mathsf{as}\ x;\; c : \underline{\tau}}$$

In the following rule
$h = \mathsf{handle}^{\nu}(c)\ \{\mathsf{return}\ x_r \to c_r, op_1(x_1;k_1) \to c_1, ..., op_n(x_n;k_n) \to c_n\}$.

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

3.5 Semantics

$$\frac{}{(\lambda x.c)\ \nu \mid \mu \rightsquigarrow c[x := \nu] \mid \mu}$$

$$\frac{}{(\Lambda i.c)\ j \mid \mu \rightsquigarrow c \mid \mu}$$

$$\frac{c_1 \mid \mu \rightsquigarrow c_1' \mid \mu'}{x \leftarrow c_1;\ c_2 \mid \mu \rightsquigarrow x \leftarrow c_1';\ c_2 \mid \mu'}$$

$$\frac{}{x \leftarrow (\mathsf{return}\ \nu);\ c \mid \mu \rightsquigarrow c[x := \nu] \mid \mu}$$

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

$$\frac{}{\mathsf{fresh}\ i;\ c \mid \mu \rightsquigarrow c \mid \mu}$$

$$\frac{l \notin Dom(\mu)}{\mathsf{new}\ \varepsilon\ \mathsf{of}\ i\ \mathsf{as}\ x;\ c \mid \mu \rightsquigarrow c[x := \mathsf{inst}(l)] \mid \mu, l := (i, \varepsilon)}$$

In the following rules
$h = \mathsf{return}\ x_r \to c_r, op_1(x_1; k_1) \to c_1, ..., op_n(x_n; k_n) \to c_n$.

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

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

$$\frac{}{\mathsf{handle}^{\mathsf{inst}(l)}(\mathsf{inst}(l)\#op_i(\nu))\ \{h\} \mid \mu \rightsquigarrow c_i[x_i := \nu, k_i := \lambda x_r.c_r] \mid \mu}$$

$$\frac{l \neq l'}{\mathsf{handle}^{\mathsf{inst}(l)}(\mathsf{inst}(l')\#op(\nu))\ \{h\} \mid \mu \rightsquigarrow x_r \leftarrow \mathsf{inst}(l')\#op(\nu);\ c_r \mid \mu}$$

$$\frac{}{\begin{array}{l}\mathsf{handle}^{\mathsf{inst}(l)}(x \leftarrow (\mathsf{inst}(l)\#op_i(\nu));\ c)\ \{h\} \mid \mu \rightsquigarrow \\ c_i[x_i := \nu, k_i := (\lambda x.\mathsf{handle}^{\mathsf{inst}(l)}(c)\ \{h\})] \mid \mu\end{array}}$$

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

3.6 Existential types

$$\begin{array}{lr}
\tau ::= & \text{(value types)} \\
\quad \exists i.\underline{\tau} & \text{(existentially quantified type)} \\
c ::= & \text{(computations)} \\
\quad \mathsf{pack}\ l\ \mathsf{in}\ \nu & \text{(pack instance variable)} \\
\quad \mathsf{unpack}\ \nu\ \mathsf{as}\ (i, x);\ c & \text{(unpack existential type)}
\end{array}$$

$$\frac{a <: b}{\exists i.a <: \exists i.b}$$

$$\frac{\Delta, i \vdash a}{\Delta \vdash \exists i.a}$$

$$\frac{\Delta; \Sigma; \Gamma \vdash \nu : \tau}{\Delta; \Sigma; \Gamma \vdash \mathsf{pack}\ i\ \mathsf{in}\ \nu : (\exists i.\tau)\ !\ \varnothing}$$

$$\frac{\begin{array}{l}\Delta; \Sigma; \Gamma \vdash \nu : \exists i.\tau \\ \Delta, i; \Sigma; \Gamma, x : \tau \vdash c : \underline{\tau} \\ i \notin \underline{\tau}\end{array}}{\Delta; \Sigma; \Gamma \vdash \mathsf{unpack}\ \nu\ \mathsf{as}\ (i, x);\ c : \underline{\tau}}$$

$$\frac{}{\mathsf{pack}\ i\ \mathsf{in}\ \nu \mid \mu \rightsquigarrow \mathsf{return}\ \nu \mid \mu}$$

$$\frac{}{\mathsf{unpack}\ \nu\ \mathsf{as}\ (i, x);\ c \mid \mu \rightsquigarrow c[x := \nu] \mid \mu}$$

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).