Worksheets

Daniel Hillerström

30th July 2015

Contents

Chapter 1

Introduction

Some introductory text.

1.1 Problem analysis

Used car-dealers got a notorious reputation for being dishonest. Dishonest, because, they are reluctant to disclose any unfortunate effects that their cars may have. On the other hand, if they *did* disclose all the effects then we probably would not buy them anyway as it appears too much effort to handle those effects.

Correspondingly, some programming languages do not disclose the potential run-time effects of code execution, e.g. the ML-family of languages. For example consider the signature `readFile : string -> [string]` for a function in SML, its suggestive name hints that given a file name the function reads the file and return the contents line by line. In order to read a file the function must inevitably perform a side-effecting action, namely, accessing a storage media. But this information is not conveyed in the function signature.

Other languages disclose effects, albeit with varying degree. For example the Java programming language requires programmers to be explicit about potential unhandled checked exceptions that may occur during run-time, e.g. `String[] readFile(String f) throws IOException`. But programmers can circumvent this requirement by raising unchecked exceptions. Critics argue that Java's checked exceptions suffer versionability and scalability issues [2], and therefore it is better not to have explicit `throws` declarations.

The Haskell programming language is also explicit about effects, but, in contrast to Java, it offers no escape hatch to be implicit. Haskell insists that every effectful computation is encapsulated inside an appropriate monad. In Haskell the file reading function would be typed as `readFile :: String -> IO [String]`, where the `IO`-annotation signifies that the function might perform an input/output side-effect. A consequence of enforcing effectful computations to be wrapped inside a monad is that the function signature conveys additional information about the computation which the programmer and compiler can rely on.

1.1.1 The problem with monads

Monads provide a remarkably powerful way for structuring computations, because they integrate effectful and pure computations in an elegant and flexible manner. Sadly, monads do not compose well [7], and accordingly it is difficult to give a monadic description of computations that might perform multiple effects. Consider the following attempt at modelling a coffee dispenser in Haskell:

Example 1 (Coffee dispenser using monads)**.** First we define the sum type `Dispensable` which has two labels: `Coffee` and `Tea`. They represent the two items that the coffee machine can dispense.

```
data Dispensable = Coffee | Tea deriving Show

type ItemCode  = Integer
type Inventory = [(ItemCode,Dispensable)]
inventory = [(1,Coffee),(2,Tea)]
```

The `ItemCode` type models a button on the coffee machine, and `Inventory` associates buttons with dispensable items. The `inventory` is fixed, i.e. it will not change during run-time. We can make this explicit by encapsulating the `inventory` inside a `Reader`-monad, e.g.

```
dispenser :: ItemCode → Reader Inventory (Maybe
    Dispensable)
dispenser n = do inv ← ask
                 let item = lookup n inv
                 return item
```

The type `Reader Inventory (Maybe Dispensable)` tells us that `dispenser` accesses a read-only instance of `Inventory` and maybe returns an instance of `Dispensable`. The `Maybe`-type captures the possibility of failure, e.g. if the user requests an item that is not in the inventory. The monadic operation `ask` retrieves the inventory from the `Reader`-monad and `lookup` checks whether the item `n` is in the inventory.

Although, `Maybe` is a monad we cannot use its monadic interface, because we are in the context of the `Reader`-monad. For this simple computation it is not an issue, but it would be desirable to be able to use the failure handling capabilities of the `Maybe`-monad. □

Imagine that we want log when tea or coffee is being dispensed. The `Writer`-monad provide such capabilities. It is not immediately clear how we can integrate `Writer` with our model. Ideally, we would want a monadic computation like:

```
do inv  ← ask
   item ← lookup n inv
   tell ∘ show $ item
   return item
```

Here the monadic operation `tell` writes to the medium contained in the `Writer`-monad. However, this code does not type check. A moment's thought reveals that using just monads there is no way to construct a type for that expression. The type we want is something like

```
Writer w □ Reader e □ Maybe Dispensable
```

where `w` is the type of the writable medium, `e` is the type of an environment and □ is some "type-glue" that joins the types together. This type is exactly a Monad Transformer type which we discuss in Section 1.1.2. But using regular monads it is not possible to construct this type. Let us desugar the above expression to see why:

```
ask >>= \inv →
  lookup n inv >>= \item →
    tell ∘ show $ item
      >> return item
```

The bind operator (`>>=`) is the problem as its type is

```
Monad m ⇒ m a → (a → m b) → m b
```

Essentially, this type tells us that we cannot compose monads of different types as the monad type `m` is fixed throughout the computation. Thus we see that monads lack compositionality and modularity in general.

Effect granularity

It is possible to solve the problem using regular monads. However, it comes at a cost as suggested by the type signature of the bind operator we can compose one monad with another as long as they got the same monadic type. So, we could just use one monadic type to describe all effects. It is very tempting to bake everything into an `IO`-monad as we possibly want to I/O capabilities at some point. Albeit, `IO` is a very conservative estimate on which effects our computation might perform. Consequently, we get coarse-grained effect signatures as opposed to more specific, fine-grained effect signatures.

1.1.2 Composing monads with Monad Transformers

Monad Transformers allow two monads to be combined by stacking one on top of the other. Furthermore, a Monad Transformer is itself a monad, and thus we can create arbitrarily complex compositions. Incidentally, Monad

Transformers can capture computations that may cause several different effects. The following example rewrites the coffee dispenser model from Example 1 using Monad Transformers.

Example 2 (Coffee dispenser using Monad Transformers)**.** Most monads have a Monad Transformer cousin; by convention Monad Transformers have a capital T suffix, e.g. the `Reader`-monad's transformer is named `ReaderT`.

We rewrite Example 1 to use the `WriterT`, and `ReaderT` monad instead of `Reader`:

```
dispenser1
:: ItemCode →
   WriterT String (ReaderT Inventory Maybe)
   Dispensable

dispenser1 n = do inv  ← lift ask
                  item ← lift ∘ lift $ lookup n inv
                  tell ∘ show $ item
                  return item
```

The type may look dubious. Basically, we have built a Monad Transformer stack with three monads:

- Top of the stack: `WriterT` with a writable medium of type `String`.
- Middle: `ReaderT` with read access to an environment of type `Inventory`.
- Bottom: `Maybe` provides exception handling capabilities.

Monad Transformers allow us to express something reminiscent of the monadic computation we sought in Example 1. It is worth noting that we now use `Maybe` as a monad as opposed to a ordinary value. The benefits are obvious as we get the exception handling capabilities of `Maybe` for "free".

However, it is not entirely free as we have to introduce `lift` operations. The `lift` operations are necessary in order to work with a specific effect down the transformer stack. For example in order to use `ask` we have to `lift` once as the `ReaderT` is the second type in the stack. Moreover, to use the monadic capabilities of `Maybe` we have to `lift` twice because it is at the bottom of the stack. Using `tell` requires no `lift`s in this example as `WriterT` is the top type. Consider what happens when we add yet another monad to the stack:

```
dispenser2
:: ItemCode →
   RandT StdGen (WriterT String (ReaderT Inventory
   Maybe)) Dispensable

dispenser2 n
  = do r    ← getRandomR (1,20)
```

```
inv  ← lift ∘ lift $ ask
item ← lift ∘ lift ∘ lift $ lookup' r n inv
lift ∘ tell ∘ show $ item
return item
where
  lookup' r n inv = if r > 10
                    then lookup n inv
                    else Nothing
```

Here we extended our model with randomness to capture the possiblilty of failure caused by the system rather than the user. The `RandT` monad provides random capabilities. Moreover, we added it to the top of the transformer stack. Accordingly, we now have to use an additional `lift`, in particular, we have to `lift` in order to use `tell` now. □

Example 2 demonstrates that we can compose monads at the cost of lifting. We can think of a `lift` operation as "peeling off a layer" of the transformer stack. Thus the transformer stack enforce a static ordering on effects and interactions between effect layers [8].

Furthermore, the ordering leaks into the type signature which complicates modularity. For example, we may have a function which takes as input an effectful computation with type signature, say, `WriterT w Reader e a`. Now, the actual effectful computation has to have a type signature with the *exact* same ordering of effects even though `Writer` and `Reader` commute, i.e. the following types are isomorphic:

$$\texttt{WriterT w Reader e a} \simeq \texttt{ReaderT e Writer w a}.$$

So, we would have to permute the type signature of the actual computation [6], e.g.

```
permute :: ReaderT e Writer w a → WriterT w Reader e a
```

In this case it is safe because the two monads commute. But in general monads do not commute and therefore the consequence of permuting monads can be severe as we shall see in the next section.

The ordering implies the semantics

The effect ordering hard wires the semantics and syntactical structure of computations. Consider the following example adapted from O'Sullivan et. al [3]:

Example 3 (Importance of effect ordering [3])**.** We will demonstrate that the `Writer` and `Maybe` monads do not commute. Let `A` be the type `WriterT String Maybe` and `B` be the type `MaybeT (Writer String)`. The two types differ in their ordering of effects; type `A` has `Writer` as its outermost effect, whilst `B` has `Maybe` as its outermost effect. Now consider the following small program that performs one `tell` operation and then fails:

```
problem :: MonadWriter String m ⇒ m ()
problem = do
  tell "this is where I fail"
  fail "oops"
```

We have two possible concrete type instantiations of `m`, namely, either `A` or `B`. But as we shall see the two types enforce different semantics:

```
ghci> runWriterT (problem :: A ())
Nothing
ghci> runWriterT $ runMaybeT (problem :: B ())
(Nothing, "this is where I fail")
```

When using type `A` we lose the result from the `tell` operation. Type `B` preserves the result. Hence the two monads do not commute, and as a result the ordering of effects determine the semantics of the computation. □

We have seen that while we gain monad compositionality with Monad Transformers we do not get modularity.

1.2 Problem statement

In the previous section we argued that programming with *explicit* effect is desirable, but we pointed out that it is not painless to program with explicit effects. In particular, we demonstrated that the monadic approach lacks compositionality and modularity. But we could regain compositionality using Monad Transformers, however the transformer stack imposes a statical ordering on effects which impedes modularity. Compositionality and modularity are two key properties in programming which we ideally would like to retain along with explicit effects. This observation leads us to the following problem statement:

> *How may we achieve a programming model with modular, composable and unordered effects?*

Plotkin and Pretnar's handlers for algebraic effects [9] affords a very attractive model for programming with effects. The principal idea is to decouple the semantics and syntactic structure of effectful computations, i.e. an effect is a collection of abstract operations. By abstract we mean that the operation by itself has no concrete implementation. Abstract operations compose seamlessly to form the syntactical structure of the computation, whilst handlers instantiate abstract operations with a concrete interpretation. We will discuss handlers and algebraic effects in greater detail in Section **??**.

We suppose that handlers for algebraic effects provide a desirable model for programming with effects. A substantial amount of work has already been put into handlers and effects. In Section 1.3 we discuss and evaluate related work before we propose our own solution in Section 1.4.

1.3 Related work

This section discusses and evaluates related work on programming models with handlers and effects.

1.3.1 The Eff language

The *Eff* programming language, by Bauer and Pretnar [13], has a first-class implementation of handlers for algebraic effects. The language has the look and feel of OCaml. Eff achieves unordered effects through a combination of effect polymorphism and subtyping.

1.3.2 Haskell libraries

We will discuss two implementations of handlers on top of Haskell by Kammar et. al [7] and Wu et. al [12].

Data types á la carte

Swierstra [4] demonstrates how to compose effectful programs using *free monads* in Haskell. The free monads form the basis for a framework for encoding handlers and effects which the subsequent libraries use.

Extensible effects

A few words on Kiselyov's paper? [8].

Handlers in action

Kammar et. al considers two different approaches to implement handlers on top of Haskell. One approach is based on free monads [4], whilst the other is a continuation-based approach [7].

Their handlers are encoded as type classes, thus handlers inherent the limitations of type classes. Particularly, type classes are not first-class in Haskell, so neither are the handlers. To achieve unordered effects they use type class constraints. Type classes can only be defined in top-level as Haskell do not permit local type-class definition. Consequently, every effect handler must be defined in the top-level too.

Furthermore, the order in which handlers are composed leak into the type signature, because their (open) handlers explicitly mention a parent handler [7]. Albeit, it does not cause an issue like Monad Transformer ordering issue, but it is still undesirable.

Kammar et. al hypothesises that an implementation based on row polymorphism may remedy the limitations and yield a cleaner design [7].

Handlers in scope

Wu et. al investigate how to use handlers to delimit the scope of effects [12] as using handlers for scoping has limitations. In other words, the ordering of handlers may affect the semantics.

They present two solutions embedded in Haskell [12]. The first solution extends the existing effect handler framework based on free monads with so-called *scope markers* which fits nicely into the framework. However they demonstrate that handlers along with scope markers are insufficient to capture higher-order scoped constructs properly.

Their second approach is continuation-based and provides a *higher-order syntax* that allows to embed programs with scoping constructs [12]. However it remains an open question whether their implementation is viable in other languages than Haskell.

1.3.3 Frank

The Frank programming language by McBride [11] takes the notion of effect handlers to the extreme. In Frank there are no functions, there are only handlers. Moreover, it employs an interesting evaluation order known as "call-by-push-value" (CBPV). Intuitively, CBPV is the unification of the strict call-by-value and non-strict call-by-name semantics. The choice whether to employ the strict or non-strict semantics has been made explicit to the programmer.

Frank distinguishes between computations and values as a consequence side-effects can only occur in computations. Hence there is a clear separation between segments of code where effects might occur and where effects are guaranteed never to occur.

1.3.4 Idris' Effects

Brady [6] presents the library EFFECTS for the dependently-typed, functional programming language IDRIS.

1.3.5 Koka with row polymorphic effects

Leijen's programming language Koka is an effect-based web-oriented language [10]. It supports arbitrary user-defined effects [14]. Notably, Koka uses row polymorphism to capture unordered effects however Koka's row polymorphism allow duplicate effect occurrences which stands in contrast to the approach we propose. In particular, Koka has no notion of effect handler except for exception handlers which are to some extent reminiscent of those in Java, C#, etc.

1.4 Proposed solution

Kammar et. al proposed that a row-based effect type system would yield a cleaner design [7].

1.4.1 Objectives, aim and scope

The aim is to examine the programming model achieved by using handlers with row polymorphic effects. In order to examine the model we must first implement it, thus the primary objective is to implement handlers and support for user-defined effects in Links.

Links has support for numerous web-oriented features, however we restrict the scope to a working implementation in top-level Links.

1.4.2 Contributions

The main contributions are:

- An implementation of effect handlers in Links.
- Support for row polymorphic user-defined effects in Links.
- An examination of programming with handlers and row polymorphic effects.

Chapter 2

Background

2.1 Row polymorphism

Row polymorphism is a typing discipline for records [1]. A record is an unordered collection of fields, e.g. $\langle l_1 : t_1, \ldots, l_n : t_n \rangle$ denotes a record type with n fields where l_i and t_i denote the name and type, respectively, of the ith field. Moreover, the record type is monomorphic, that is, the type is fixed. Row polymorphism, as the name suggests, makes record types polymorphic.

The following illustrates the power of row polymorphism.

Example 4. OCaml's (regular) record types are monomorphic.[1] Consider the following two record type definitions in OCaml

```
type student    = {name : string; id : string}
type supervisor = {name : string; group : string}
```

Now we can create instances of `student` and `supervisor`

```
ocaml> let daniel = {name="Daniel"; id="s1467124"};;
val daniel : person = {name="Daniel"; id="s1467124"}

ocaml> let sam = {name="Sam"; group="LFCS"};;
val sam : supervisor = {name="Sam"; group="LFCS"}
```

As expected the OCaml compiler infers the correct record types for both instances. Since both record types have the field *name* in common we might expect to define a function which prints the name field of either record type, e.g.

```
ocaml> let print_name r = r.name;;
ocaml> let () =
          print_name daniel;
          print_name sam;;
```

[1] OCaml's object types are row polymorphic.

Surprisingly this yields the following type error

```
          print_name daniel;
                     ^^^^^^
Error: This expression has type student
       but an expression was expected of type
    supervisor
```

The record daniel is not compatible with the type of print_name. Because the record types are monomorphic, the compiler has to decide on compile time which record type print_name accepts as input parameter. Apparently in this example the compiler has decided to type print_name as supervisor → string.

Now consider the same example in Links. In contrast to OCaml record types are polymorphic in Links. First we define the print_name function

```
links> fun print_name(r) { r.name };
print_name = fun : ((name:a|ρ)) ⇝ a
```

Links tells us that the function accepts a record type which has *at least* the field name. The field type is polymorphic as signified by the presence of the type variable a. The additional type variable ρ is a polymorphic row variable which can be instantiated to additional fields, hence the actual input record may contain *more* fields. Now our printing function works as expected:

```
links> fun(){ print_name(daniel);
              print_name(sam) }();
Daniel
Sam
```

Here we wrapped the applications of print_name inside a parameterless function because Links does not support expression sequencing in the top-level. □

More on ρ, unification, etc...

Chapter 3

Programming with handlers in Links

3.1 Closed handlers

A closed handler handles a fixed set of effects, that is, it effectively describes an upper bound on which kind of effects a computation may perform. In Links this bound is made explicit in the handler's type, e.g. the closed handler h

```
var h = handler(m) {
  case Op(p,_)   → p
  case Return(x) → x
}
```

has the type $(() \xrightarrow{\{Op:a \to a\}} a) \to a$ where the absence of a row variable in the effect signature implies that the computation m may not perform any other effects than Op. It is considered a type error to handle a computation whose effect signature is larger than the handler supports.

This restriction introduces slack into the type system. To illustrate the slack consider the following computation

```
fun comp() {
  do Op(true);
  if (false == true) {
    do Op2(false)
  } else {
    true
  }
}
```

The computation comp has type $() \xrightarrow{\{Op:\texttt{Bool} \to (), Op2:\texttt{Bool} \to \texttt{Bool} \mid \rho\}} \texttt{Bool}$. Obviously, Op2 never gets discharged. However, attempting to handle comp with the handler h yields a type error because Op2 is present in the effect signature of `comp`. The type system is conservative, but in general it is undecidable whether the first or second branch of a conditional expression will be taken [5].

The following sections will show increasingly interesting examples of programming with closed handlers in Links.

3.1.1 Transforming the results of computations

Handlers take computations as input. From a handler's perspective a computation is a *thunk*, i.e. a parameterless function whose type is similar to $() \xrightarrow{\{Op_i : a_i \to b_i\}} c$. The first few examples show how to transform the output of a computation using handlers. We begin with a handler that appears to be rather boring, but in fact proves very useful as we shall see later in Section **??**.

Example 5 (The force handler)**.** We dub the handler `force` as it takes a computation (thunk) as input, evaluates it and returns its result. It has type `force` : $(() \to a) \to a$ and it is defined as

```
var force = handler(m) {
    case Return(x) → x
}
```

Essentially, this handler applies the identity transformation to the result of the computation `m`. Running `force` on a few examples should yield no surprises:

```
fun fortytwo() { 42 }
links> force(fortytwo);
42 : Int

fun hello() { "Hello" }
links> force(hello);
"Hello" : String
```

The handler `force` behaves as expected for these trivial examples. But suppose we want to print "*Hello World*" to the standard output, e.g.

```
fun print_hello() { print("Hello World") }
links> force(print_hello);
Type error: ... # Omitted for brevity
```

then the Links compiler contemptuously halts with a type error! The type of `print_hello` is $() \rightsquigarrow ()$ which at first glance may appear to be compatible with the type of formal parameter `m`. But printing to standard output is effectful action, as indicated by the squiggly arrow in the signature, hence `print_hello` is an effectful computation. Since `force` does not handle any effects we get the type error. □

As Example 5 demonstrated the handler `force` could not handle the print effect caused by `print_hello`. In fact no handler in Links is able to handle `print_hello` because the print effect is a syntactic, built-in effect known as *wild*. Handlers only handles user-defined effects.

The next example demonstrates an actual transformation.

Example 6 (The listify handler)**.** The `listify` handler transforms the result of a handled computation into a singleton list. Its type is `listify` : $(() \to a) \to [a]$ and its definition is straightforward

```
var listify = handler(m) {
  case Return(x) → [x]
}
```

When handling the computations from Example 5 we see that it behaves as expected, e.g.

```
links> listify(fortytwo);
[42] : [Int]

links> listify(hello);
["Hello"] : [String]

fun list123() { [1,2,3] }
links> listify(list123);
[[1,2,3]] : [[Int]]
```

These examples also illustrate the `Return`-case serves a similar purpose to the monadic `return`-function in Haskell whose type is `return` : $a \to m\,a$ for a monad m. It "lifts" the result into an adequate type. □

In a similar fashion to the handler `listify` in Example 6 we can define handlers that increment results by 1, perform a complex calculation using the result of the computation or wholly ignore the result. The bottom line is that it must ensure its output has an adequate type. In the case for `listify` the type must be a list of whatever type the computation yielded.

3.1.2 Exception handling

Until now we have only seen some simple transformations. Let us spice things up a bit. Example 7 introduces the practical handler `maybe`. It is similar to the `Maybe`-monad in Haskell. For reference we briefly sketched the behaviour of the `Maybe`-monad in Section 1.1.1.

Example 7 (The maybe handler)**.** The `maybe` handler handles one operation `Fail` : $a \to a$ that can be used to indicate that something unexpected has happened in a computation. The handler returns `Nothing` when `Fail` is raised, and Just the result when the computation succeeds, thus its type is

$$\texttt{maybe} : (() \xrightarrow{\{\texttt{Fail}:a \to a\}} b) \to [|\texttt{Just} : b|\texttt{Nothing}|\rho|].$$

It is defined as

```
var maybe = handler(m) {
  case Fail(_,_) → Nothing
  case Return(x) → Just(x)
}
```

When a computation raises Fail the handler discards the remainder of the computation and returns Nothing immediately, e.g.

```
fun yikes() {
  var x = "Yikes!";
  do Fail();
  x
}
links> maybe(yikes);
Nothing() : [|Just:String|Nothing|ρ|]
```

and if the computation succeeds it transforms the result, e.g.

```
fun success() {
  true
}
links> maybe(success);
Just(true) : [|Just:Bool|Nothing|ρ|]
```

□

The next example demonstrates an alternative "exception handling strategy".

Example 8 (The recover handler)**.** We can define a handler recover which ignores the raised exception and resumes execution of the computation. The type of recover is

$$\texttt{recover} : (() \xrightarrow{\{\texttt{Fail}:a \to ()\}} b) \to [|\texttt{Just} : b|\rho|].$$

A slight reminder here: The label Nothing is absent from the handler's output type because Just is a polymorphic variant label and its relation to Nothing is conventional. We define recover as

```
var recover = handler(m) {
  case Fail(_,k) → k(())
  case Return(x) → Just(x)
}
```

In contrast to maybe from Example 7 the recover handler invokes the continuation k once. This invocation effectively resumes execution of the computation. Consider recover applied to the computation yikes from before

```
links> recover(yikes)
Just("Yikes!") : [|Just:String|ρ|]
```

□

Although it is seldom a sound strategy to ignore exceptions the two Examples 7 and 8 demonstrate that we can change the semantics of the computation by changing the handler.

Bibliography

[1] Didier Rémy. “Type Inference for Records in a Natural Extension of ML”. In: *Theoretical Aspects Of Object-Oriented Programming. Types, Semantics and Language Design*. Ed. by Carl A. Gunter and John C. Mitchell. MIT Press, 1993.

[2] Bill Venners and Bruce Eckel. *The Trouble with Checked Exceptions: A Conversation with Anders Hejlsberg, Part II*. `http://www.artima.com/intv/handcuffs.html`. 2003.

[3] Bryan O’Sullivan, John Goerzen and Don Stewart. *Real World Haskell*. 1st. O’Reilly Media, Inc., 2008. ISBN: 0596514980, 9780596514983.

[4] Wouter Swierstra. “Data types à la carte”. In: *Journal of Functional Programming* 18 (04 July 2008), pp. 423–436. ISSN: 1469-7653. DOI: `10.1017/S0956796808006758`. URL: `http://journals.cambridge.org/article_S0956796808006758`.

[5] Hans Hüttel. *Transitions and Trees: An Introduction to Structural Operational Semantics*. 1st. New York, NY, USA: Cambridge University Press, 2010. ISBN: 0521147093, 9780521147095.

[6] Edwin Brady. “Programming and reasoning with algebraic effects and dependent types”. In: *ACM SIGPLAN International Conference on Functional Programming, ICFP’13, Boston, MA, USA - September 25 - 27, 2013*. 2013, pp. 133–144. DOI: `10.1145/2500365.2500581`. URL: `http://doi.acm.org/10.1145/2500365.2500581`.

[7] Ohad Kammar, Sam Lindley and Nicolas Oury. “Handlers in Action”. In: *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*. ICFP ’13. Boston, Massachusetts, USA: ACM, 2013, pp. 145–158. ISBN: 978-1-4503-2326-0. DOI: `10.1145/2500365.2500590`. URL: `http://doi.acm.org/10.1145/2500365.2500590`.

[8] Oleg Kiselyov, Amr Sabry and Cameron Swords. “Extensible Effects: An Alternative to Monad Transformers.” In: *Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell*. Haskell ’13. Boston, Massachusetts, USA: ACM, 2013, pp. 59–70. ISBN: 978-1-4503-2383-3. DOI:

10.1145/2503778.2503791. URL: http://doi.acm.org/10.1145/2503778.2503791.

[9] Gordon D. Plotkin and Matija Pretnar. "Handling Algebraic Effects". In: *Logical Methods in Computer Science* 9.4 (2013). DOI: 10.2168/LMCS-9(4:23)2013. URL: http://dx.doi.org/10.2168/LMCS-9(4:23)2013.

[10] Daan Leijen. "Koka: Programming with Row Polymorphic Effect Types". In: *Mathematically Structured Functional Programming 2014*. EPTCS, 2014. URL: http://research.microsoft.com/apps/pubs/default.aspx?id=210640.

[11] Sam Lindley and Conor McBride. *Do Be Do Be Do*. http://homepages.inf.ed.ac.uk/slindley/papers/frankly-draft-march2014.pdf. Draft, March 2014. 2014.

[12] Nicolas Wu, Tom Schrijvers and Ralf Hinze. "Effect Handlers in Scope". In: *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell*. Haskell '14. Gothenburg, Sweden: ACM, 2014, pp. 1–12. ISBN: 978-1-4503-3041-1. DOI: 10.1145/2633357.2633358. URL: http://doi.acm.org/10.1145/2633357.2633358.

[13] Andrej Bauer and Matija Pretnar. "Programming with algebraic effects and handlers". In: *Journal of Logic and Algebraic Methods in Programming* 84.1 (2015), pp. 108–123. DOI: 10.1016/j.jlamp.2014.02.001. URL: http://dx.doi.org/10.1016/j.jlamp.2014.02.001.

[14] Niki Vazou and Daan Leijen. *Remarrying effects and monads*. Submitted to ICFP '15. 2015.