Handlers for Algebraic Effects in Links

Daniel Hillerström

Master of Science
Computer Science
School of Informatics
University of Edinburgh
2015

Abstract

An abstract appears here...

Acknowledgements

Thanks to…

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Daniel Hillerström*)

Contents

1 Introduction **1**
- 1.1 Problem analysis . . . 2
- 1.2 Problem statement . . . 8
- 1.3 Related work . . . 9
- 1.4 Proposed solution . . . 11

2 Background **13**
- 2.1 Handlers and algebraic effects . . . 13
- 2.2 Row polymorphism . . . 16

3 Programming with handlers in Links **19**
- 3.1 Discharging operations in Links . . . 19
- 3.2 Closed handlers . . . 20
- 3.3 Open handlers . . . 32

4 Implementation **39**
- 4.1 Early desugaring of handlers . . . 39
- 4.2 Type checking . . . 41
- 4.3 Pattern-matching compilation . . . 42
- 4.4 Interpreter . . . 43

5 Evaluation **47**
- 5.1 Handlers with row polymorphic effects . . . 47
- 5.2 Handlers in Links . . . 47
- 5.3 Performance . . . 47

6 Conclusion and future work **51**
- 6.1 Conclusion . . . 51
- 6.2 Future work . . . 51

Bibliography **53**

Chapter 1

Introduction

A recipe for the ideal programming model would include: Compositionality, modularity and explicit effects.

Compositionality lets us break a complex problem into smaller constituent problems. The complexity of a greater problem can be harnessed by composing solutions to smaller, likely easier, constituent problems. Moreover, compositionality encourage reuse of specialised components to solve future problems.

Modularity refers to the degree of coupling between components. A high degree of modularity implies low coupling between components. Low coupling can be achieved by keeping interfaces between connected components abstract. Abstract interfaces lets us exchange one concrete implementation for another implementation effortlessly.

Together modularity and compositionality form the basis for a powerful programming model. However, being explicit about effects is often neglected [13]. An effect give a static description of the possible state-changing actions that may occur during evaluation of a particular piece of code. Moreover, effects can be informative for the compiler as well as the programmer [8, 3, 13].

Plotkin and Pretnar's *handlers for algebraic effects* [17] afford a compelling programming model which unifies the compositionality, modularity and effectful programming. This thesis examines the programming model as basis for effectful programming.

1.1 Problem analysis

Programming languages vary greatly in their approach to effects. Some 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` $\rightarrow$ `[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 [21], 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 [1]. In Haskell the file reading function would be typed as `readFile :: String` $\rightarrow$ `IO [String]`, where the `IO`-annotation signifies that the function might perform an input/output side-effect. We can think of `IO` as an effect type. In fact, Wadler and Thiemann gave the theoretical foundation for interpreting any monad as an effect type [22].

1.1.1 Benefits of being explicit about effects

An effect conveys additional information about what might happen during evaluation of a computation. This information may be used by an optimising compiler transform the computation into an equivalent, more efficient computation. For example, fine-grained effects can tell us precisely when it is safe to reuse a particular piece of code [8]. Moreover, the additional information can aid in verification of the code up-front [3].

[1] Strictly speaking it is not true as any function can be defined in terms of side-effecting `error` function without being reflected in the type signature.

Finally, explicit effects provide additional documentation to the programmer about the code. As a result thereof the programmer gain better insight into what the computation actually *does* without breaking the abstraction.

1.1.2 A monadic effectful coffee dispenser

Monads are powerful abstractions for structuring computations as long as we are working inside the same monad. Because, sadly, monads are do not compose well [7], and consequently 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)**.** The coffee dispenser is effectful, that is, it reacts to user input and may fail. Furthermore, we want to be explicit about the effects that the dispenser may cause.

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` will not change during run-time. We can capture this property in the effect signature by encapsulating the `inventory` inside a `Reader`-monad. Furthermore, we use the `Maybe`-type to capture the possibility of failure, 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 tell us that in the event of an error we get `Nothing`, for instance if the user requests an item that is not in the inventory, otherwise we get `Just` the requested item. 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 1.1.3. 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.

1.1.2.1 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.

1.1.3 A better monadic effectful coffee dispenser

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 [9].

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 [3], 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.

1.1.3.1 The importance of effect ordering

The effect ordering hard wires the semantics and syntactical structure of computations. Consider the following example adapted from O'Sullivan et. al [14]:

Example 3 (Importance of effect ordering [14])**.** 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 [17] 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 2.1.

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 [1], 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 [23].

1.3.2.1 Data types á la carte

Swierstra [19] 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.

1.3.2.2 Extensible effects

A few words on Kiselyov's paper? [9].

1.3.2.3 Handlers in action

Kammar et. al considers two different approaches to implement handlers on top of Haskell. One approach is based on free monads [19], 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].

1.3.2.4 Handlers in scope

Wu et. al investigate how to use handlers to delimit the scope of effects [23] 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 [23]. 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 [23]. 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 [12] 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 [3] 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 [20]. 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

The issues with existing models for effectful programming boil down to variants of the ordering problem as discussed in Section 1.1.3. Therefore we propose *handlers for algebraic effects* with a small twist: to eliminate effect ordering we will use *row polymorphism*. We discuss row polymorphism in greater detail in Section 2.2.

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 is a web-oriented functional programming language that already has a row-based effect system in place. Because Links has built-in support for numerous web-oriented features that are not key to our treatment, we restrict the scope to a working implementation in top-level Links. We introduce the relevant aspects of Links in Section **??**.

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 Handlers and algebraic effects

Algebraic effects and handlers have their foundation in category theory [16, 17]. Plotkin and Power [15, 16] gave a categorical treatment of algebraic effects. The term “algebraic” implies that an effect ought to have an equational theory, however we consider only free algebras, that is our theories are equationless. Therefore we will not delve into the theoretical foundations of algebraic effects and handlers, rather we will take a more practical approach. Moreover, we will use the terms algebraic effect and effect interchangeably.

2.1.1 Algebraic effect

An algebraic effect is a collection of operation signatures [11]. For example, we might define an algebraic effect `Choice` for making a boolean choice with the following signature:

$$\mathtt{Choice} \stackrel{\mathsf{def}}{=} \{\mathtt{Choose} : () \to \mathtt{Bool}\}$$

Here `Choose` is a nullary operation whose return type is boolean. The effect `Choice` is the singleton set whose only member is `Choose`.

The operation `Choose` is abstract, that is it has no concrete implementation. We say that computations composed from algebraic effects are *abstract computations*. Without handlers abstract computations are meaningless as handlers faithfully interpret effects by instantiating them with concrete implementations.

2.1.2 Effect handler

Benton and Kennedy generalised exception handlers [2] (as known from SML, C#, Java, etc) to expose a continuation to the programmer in the case no exceptions occurred during the handled context. Later their work was adapted by Plotkin and Pretnar [17] to handle arbitrary effects, thus they coined the notion of handlers for algebraic effects.

Intuitively, an effect handler is a generalised function which takes an abstract computation as input, and embodies a collection of cases for pattern matching on operations that may be discharged during the evaluation of the input computation.

2.1.3 Interpreting effects as computation trees

Abstract computations are syntactic structure without a particular semantics. Handlers assign semantics to abstract computations. To further develop intuition about handlers and effects we illustrate a diagrammatic interpretation of effects as computation trees [11]. Moreover, we see how we can assign different semantics to the same abstract computation. Consider the following expression:

```
if Choose() then
  if Choose() then 2
  else 4
else
  if Choose() then 8
  else 16
```

The expression is a nested conditional expression. We can picture this expression as a tree where the nodes encode operations, edges correspond to branching, and the leaves encode concrete values. For example Figure 2.1 depicts the above expression as a computation tree. During evaluation of the expression we eventually have to interpret the root node `Choose` in Figure 2.1 and possibly its immediate subtrees. There are multiple possible interpretations. One interpretation is to always interpret `Choose` as `true` which figuratively corresponds to taking the left (true) branch. The node we arrive at is also a `Choose`-node, so again we choose the left branch arriving at a leaf that contains the concrete value 2. Hence under this interpretation the handler collapses the computation tree to the leaf 2. Dually, we could always choose false which leads to the output value 16. Fig-

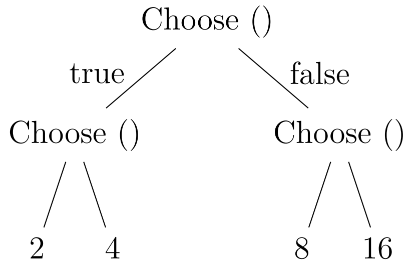
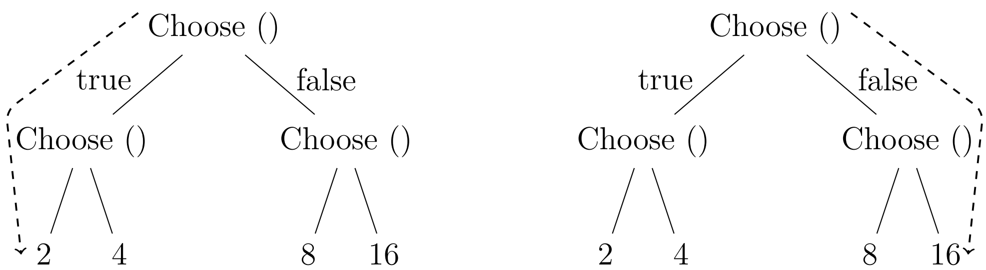


Figure 2.1: Interpretation of the conditional expression as a computation tree. The left edges correspond to taking the first branch in a conditional expression. Analogously, the right edges correspond to taking the second branch.

ures 2.2a and 2.2b illustrate the two interpretation respectively. Alternatively, we could make a random choice between true and false at each branch. Again, this interpretation leads to one single output value. Albeit, the output value would be non-deterministic under this interpretation.

Yet another interpretation is to enumerate all possible choices. For example, we can choose explore the left branch and thereafter the right branch at each node. Under this interpretation we effectively visit the entire tree. Therefore, the handler transforms the computation tree into a set of its leaves. Figure 2.3 illustrates the tree traversal. The interpretations we have discussed so far share a characteristic: Each node (operation) is handled uniformly. That is, the same strategy is applied to similar nodes. This holds in general for any handler and computation. So, we can think of handlers as kinds of fold-functions that transform computation trees [7].



(a) The "positive" interpretation: Always choose true. Output: 2.

(b) The "negative" interpretation: Always choose false. Output: 16.

Figure 2.2: Two different interpretations.

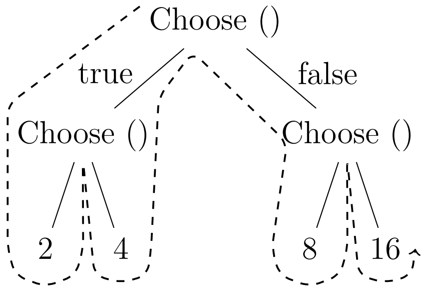


Figure 2.3: Enumerate all possible choices. Output: $\{2,4,8,16\}$.

2.2 Row polymorphism

Row polymorphism is a typing discipline for records [18]. 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;
```

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

```
print_name sam;;
```

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` $\rightarrow$ `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

Through a series of examples we will explore programming with two types of effect handlers in Links. Section 3.2 introduces *closed handlers* and in particular emphasises the high degree of modularity afforded by closed handlers. Section 3.3 introduces the slightly more generalised *open handlers* and focuses mainly on the compositionality of (open) handlers.

3.1 Discharging operations in Links

Syntactically, operations are similar to variants in Links. Every operation name starts with a capital letter, e.g. `Get`, `Put`, etc. Every operation takes an input and yields an output. The output from discharging an operation is entirely decided by effect handlers in the evaluation context. That is, alone an operation does not have any semantics.

Operations are discharged using the `do`-primitive. However discharging an operation in an unhandled context yields an evaluation error:

```
links> do Get();
*** Error: Unhandled operation: Get()
```

The typing of operations is uniform because every operation takes exactly one input. Therefore the type of an operation is on the form $a \to b$ where a and b are type variables. In order to simulate multiple parameters one can instantiate a to a record type, e.g. $\texttt{Put}((\texttt{true}, 1))$ is an operation of type $(\texttt{Bool}, \texttt{Int}) \to b$.

Similarly, one can simulate a nullary operation by passing the empty record, e.g. `Get`() has type $() \to b$.

3.2 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`

```
handler h(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.

3.2.1 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 3.3.

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.

3.2.2 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 1.1.2.

Example 7 (The maybe handler)**.** The `maybe` handler handles one operation $\texttt{Fail} : a \to a$ 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.

3.2.3 Interpreting Nim

Nim is a well-studied mathematical strategic game, and probably among the oldest of its kind [6]. In Nim two players take turns to pick between one and three sticks from heaps of sticks. Whoever takes the last stick wins. This play style is also known as *normal play*.

Nim enjoys many interesting game theoretic properties, however we will use a simplified version of Nim to demonstrate how handlers can give different interpretations of the same game. In our simplified version there is only one heap of n sticks. Moreover, there are two players: Alice and Bob, and Alice always starts. Our model is adapted from Kammar et. al [7].

We model the game as two mutual recursive abstract computations, e.g.

```
# Input n is the number of remaining sticks
fun aliceTurn(n) {
```

```
  if (n == 0) {
    Bob
  } else {
    var take = do Move((Alice,n));
    var r = n - take;
    bobTurn(r)
  }
}

# Symmetrically for Bob
fun bobTurn(n) {
  if (n == 0) {
    Alice
  } else {
    var take = do Move((Bob,n));
    var r = n - take;
    aliceTurn(r)
  }
}
```

The two computations are symmetrical. The input parameter `n` is the number of sticks left in the heap. First, Alice tests whether there are any sticks left, if there is not then she declares `Bob` the winner, otherwise she performs her move and then she gives the turn to Bob. The game has one abstract operation `Move` which has the inferred type $\texttt{Move} : ([|\texttt{Alice}|\texttt{Bob}|\rho|], \texttt{Int}) \to \texttt{Int}$, i.e. it takes two arguments

1. Who's turn it is,
2. and the number of remaining sticks.

The operation `Move` returns the number of sticks that the current player takes. Figure 3.1 depicts the shape of the computation tree representation of the game.

Tree

Figure 3.1: Nim game computation tree

The depth of the tree is potentially infinite as the depth depends entirely on the interpretation of the operation Move. The inferred type of a game is

$$\texttt{aliceTurn} : \texttt{Int} \xrightarrow{\texttt{Move}:([|\texttt{Alice}|\texttt{Bob}|\rho|],\texttt{Int})\to\texttt{Int}} [|\texttt{Alice}|\texttt{Bob}|\rho|]$$

Because it is an unary function it cannot be used as an input to any handler. We rectify the problem by using a closure, i.e. we wrap the game function inside a nullary function like `fun(){aliceTurn(n)}` where `n` is a free variable captured by the surrounding context. For conveniency, we define an auxiliary function `play` to abstract away these details. It takes as input a game handler `gh` and the number of sticks at the beginning of game `n`. Moreover, the function `play` enforces the rule that Alice always starts, e.g.

```
fun play(gh, n) {
  gh(fun() {
      aliceTurn(n)
    })
}
```

The following examples demonstrates how handlers encode the strategic behaviour of the players.

Example 9 (A naïve strategy). A very naïve strategy is to pick just *one* stick at every turn. Its implementation is straightforward

```
var naive = handler(m) {
  case Move(_,k) → k(1)
  case Return(x) → x
};
```

Here Move is handled uniformly. Independent of the parameterisation it always invokes the continuation `k` with 1 which corresponds to the player taking just 1 stick from the heap.

A moment's thought will tell us that we can easily predict the winner when using the `naive` strategy. The parity of n, the number of sticks at the beginning, determines the winner. For odd n Alice wins and vice versa for even n, e.g.

```
links> play(naive, 5);
Alice() : [|Alice|Bob|ρ|]

links> play(naive, 10);
Bob() : [|Alice|Bob|ρ|]
```

```
links> play(naive, 101);
Alice() : [|Alice|Bob|ρ|]
```

□

Example 10 (Perfect vs perfect strategy)**.** A perfect strategy makes an optimal move at each turn. An optimal move depends on the remaining number of sticks n. The perfect move can be defined as a function of n, e.g.

$$\text{perfect}(n) = \max\{n \bmod 4, 1\}$$

In our restricted Nim game a perfect strategy is a winning strategy for Alice if and only if the number of remaining sticks is *not* divisible by 4.

We implement the function `perfect` above with an addition: We pass it a continuation as second parameter

```
fun perfect(n, k) {
  k(max(mod(n,4),1))
}
```

The continuation is invoked with the optimal move. Now we can easily give a handler that assigns perfect strategies to both Alice and Bob, e.g.

```
var pvp = handler(m) {
  case Move((_,n),k) → perfect(n, k)
  case Return(x)     → x
};
```

By running some examples we see that Alice wins whenever n is not divisible by four:

```
links> play(pvp, 9);
Alice() : [|Alice|Bob|ρ|]

links> play(pvp, 18);
Alice() : [|Alice|Bob|ρ|]

links> play(pvp, 36);
Bob() : [|Alice|Bob|ρ|]
```

□

Example 11 (Mixing strategies)**.** A strategy often encountered in game theory is *mixing* which implies a player randomises its strategies in order to confuse its opponent. In similar fashion to `perfect` from Example 10 we define a function `mix` which chooses a strategy

```
fun mix(n,k) {
  var r = mod(nextInt(), 3) + 1;
  if (r > 1 && n ≥ r)) {
     k(r)
  } else {
     k(1)
  }
}
```

The function `nextInt` returns the next integer in some random sequence. The random integer is projected into the cyclic group $\mathbb{Z}_3 = \{0,1,2\}$ generated by 3. We add one to map it onto the set of valid moves $\{1,2,3\}$. If the random choice `r` is greater than the number of remaining sticks `n` then we default to take one (even though the optimal choice might be to take two).

The `mixing` strategy handler is similar to perfect-vs-perfect handler from Example 10

```
var mixing = handler(m) {
    case Move((_,n),k) → mix(n,k)
    case Return(x)     → x
};
```

Replaying the same game a few times ought eventually yield the two possible outcomes

```
links> play(mixing, 7);
Bob() : [|Alice|Bob|ρ|]

links> play(mixing, 7);
Alice() : [|Alice|Bob|ρ|]
```

□

Example 12 (Brute force strategy)**.** Examples 9-11 only invoked the continuation once per move. However, we can invoke the continuation multiple times to enumerate all possible future moves, this way we can brute force a winning strategy, if one exists. In order to brute force a winning strategy, we define a convenient utility function which computes the set of valid moves given the number of remaining sticks

```
fun validMoves(n) {
 filter(fun(m) { m ≤ n }, [1,2,3])
}
```

The function simply filters out impossible moves based on the current configuration n. Note when $n > 3$ the function `validMoves` behaves like the identity function. The function `bruteForce` computes the winning strategy for a particular player if such a strategy exists:

```
fun bruteForce(player, n, k) {
  var winners = map(k, validMoves(n));
  var hasPlayerWon = indexOf(player, winners);
  switch (hasPlayerWon) {
     case Nothing → k(1)
     case Just(i) → k(i+1)
  }
}
```

The first line inside `bruteForce` is the critical point. Here we map the continuation `k` over the possible valid moves in the current game configuration. Thus the function effectively simulates all possible future configurations yielding a list of possible winners. The auxiliary function `indexOf` looks up the position of `player` in the list of winners. The position plus one corresponds to the winning strategy because lists indexes are zero-based. If the player has a winning strategy then the (zero-based) position is returned inside a `Just`, otherwise `Nothing` is returned.

Let Alice play the brute force strategy and let Bob play the perfect strategy which is captured by the strategy handler `bfvp`

```
var bfvp = handler(m) {
  case Move((Alice,n),k) → bruteForce(Alice,n,k)
  case Move((Bob,n),k)   → perfect(n,k)
  case Return(x)         → x
};
```

Here we use deep pattern-matching to distinguish between when `Alice` and `Bob`'s moves. Obviously, the brute force strategy is inefficient as it redoes a lot of work for each move. The winning strategy that it discovers is exactly same as the perfect strategy. Albeit, `bruteForce` computes it in exponential time whilst `perfect` computes it in constant time. The following outcomes witness that `bruteForce` and `perfect` behaves identically

```
links> play(bfvp, 9);
Alice() : [|Alice|Bob|ρ|]

links> play(bfvp, 18);
Alice() : [|Alice|Bob|ρ|]

links> play(bfvp, 36);
Bob() : [|Alice|Bob|ρ|]
```

□

Although, the `bruteForce` strategy is significantly slower than `perfect` strategy in Example 12 the point of interest here is not efficiency but rather modularity. As Example 12 nicely demonstrates we can swap two observably equivalent handlers effortlessly. Clearly, this has practical applications for example one would be able to quickly create a prototypical system with slow but easy to implement components. Then later as the system scales one can change the slow components for a faster ones effortlessly.

Examples 9-12 gave different interpretations of the same game. Furthermore, the all computed the same thing, namely, the winner. We can use handlers to create data from computations. For example we can construct the game tree in a Nim game as Example 13 shows.

Example 13 (Game tree generator)**.** A node in a game tree represents a particular player's turn and an edge corresponds to a particular move. A path down the tree corresponds to a particular sequence of moves taken by the players ending in a leaf node which corresponds to the winner. Figure 3.2 shows an example game tree when starting with 3 sticks.

Our game tree is a ternary tree which we represent using a recursive variant type, e.g.

$$\texttt{MoveTree} \stackrel{\text{def}}{=} [|\texttt{Take} : (\texttt{Player}, [(\texttt{Int},\texttt{MoveTree})]) \,|\, \texttt{Winner} : \texttt{Player}|]$$

Actually, we will not use the type explicitly, but rather rely on Links' type inference. It will infer a polymorphic variant type rather than the monomorphic type given above.

We define a function `reifyMove` which takes a player, the number of sticks, and a continuation to construct a node in the game tree, e.g.

```
fun reifyMove(player, n, k) {
  var moves = map(k, validMoves(n));
  var interval = range(1,length(moves));
  Take(player, zip(interval, moves))
}
```

First, we map the continuation over the possible valid moves in the current game configuration to enumerate the subsequent game trees. The `interval` is a list of integers from one to the number of immediate subtrees. Finally, we construct a node `Take` with `player` and the possible subsequent game trees.

We determine the winner of a particular play in the `Return`-case of the handler, e.g.

```
var mtGen = handler(m) {
  case Move((player,n),k) → reifyMove(player,n,k)
  case Return(x)          → Winner(x)
};
```

The inferred type for `mtGen` witnesses that the handler indeed constructs a tree structure:

$$\texttt{mtGen} : (() \xrightarrow{\{\texttt{Move}:(a,\texttt{Int})\to\texttt{Int}\}} b) \to \mu c.[|\texttt{Take} : (a, [(\texttt{Int}, c)])|\texttt{Winner} : b|\rho|]$$

Figure 3.2 depicts the game tree generated by the handler when starting with 3 sticks.

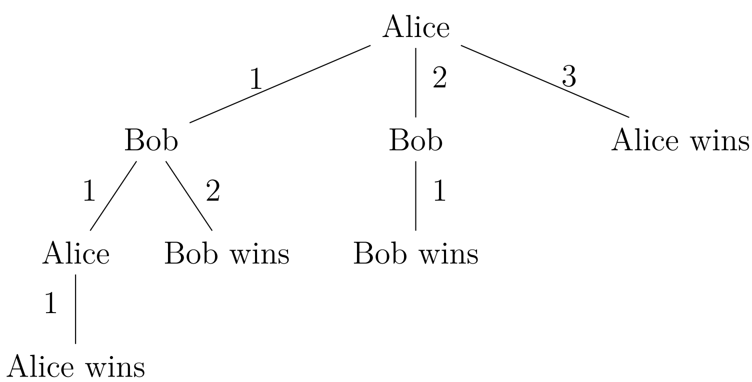


Figure 3.2: Pretty print of the game tree generated by `play(mtGen, 3)`.

□

3.3 Open handlers

Open handlers are the dual to closed handlers when we think in terms of bounds on effects. An open handler give a lower bound on the kind of effects it will handle. Through composition of open handlers we can achieve a tighter bound on the handled effects. Consequently, one can delegate responsibility to *specialised* handlers that handle a particular subset of the effects. Unhandled operations are forwarded to subsequent handlers. In other words, an open handler partially interprets an abstract computation and leaves the remainder for other handlers.

In Links the concrete syntax for open handlers is similar to that for closed handlers. To declare an open handler one simply uses the keyword `open` in the declaration, e.g.

```
open handler h(m) {
  case Opᵢ(pᵢ,kᵢ)  → bodyᵢ
  case Return(x)  → body
}
```

The inferred type for the open handler `h` is more complex than its closed counterpart:

$$\mathtt{h} : (() \xrightarrow{\{\mathtt{Op}_i : a_i \to b_i \mid \rho\}} c) \to () \xrightarrow{\{\mathtt{Op}_i : \alpha_i \mid \rho\}} d$$

Notice that the effect row of the input computation is *polymorphic* as signified by the presence of the row variable ρ. Accordingly, the input computation may perform more operations than the handler handles. The output type of an open handler looks very similar to its input type. The input as well as the output is a thunk. Moreover, their effect rows share the same polymorphic row variable ρ. But their operation signatures differ. The polymorphic variable α_i denotes that the i'th operation may be present or absent from the effect row.

Since the input type and output type of open handlers match we can compose open handlers seamlessly. The order of composition implicitly defines a stack of handlers. For example the composition of three handlers $(h_1 \circ h_2 \circ h_3)(m)$ applied to some computation m defines a stack where h_3 is the top-most element. Thus the handler stack is built outside in. The ordering inside the stack determines which handler is invoked when m discharges an operation. First the top-most handler is invoked, and if it cannot handle the discharged operation then the operation is forwarded to the second top-most handler and so forth.

Consequently, the order of composition may affect the semantics, say, h_1 and h_2 interpret the same operation differently, then, $h_1 \circ h_2$ and $h_2 \circ h_1$ potentially yield different results.

The composition of open handlers is itself an open handler, thus it will return a thunk itself. For example $(h_1 \circ h_2 \circ h_3)(m)$ yield some nullary function $() \to a$ which we must explicitly invoke to obtain the result of the computation m. To avoid this extra invocation recall the `force` handler from Section 3.2.1. We can apply the closed handler `force` to obtain the result of m directly, e.g. $(\texttt{force} \circ h_1 \circ h_2 \circ h_3)(m)$ yields a result of type a immediately.

3.3.1 An effectful coffee dispenser in Links

In Section 1.1.2 and 1.1.3 we implemented a model of a coffee dispenser in Haskell using monads (Examples 1 and 2). However, it was difficult to extend the model to include more properties like writing to a display and system failures without resorting to Monad Transformers due to regular monads' lack of compositionality.

In contrast, the modularity and compositionality afforded by (open) handlers enable us to easily implement a the a highly modular coffee dispenser model in Links. Example 14 implements the model.

Example 14 (Coffee dispenser)**.** The coffee dispenser performs two operations directly

1. `Ask`: Retrieves the inventory.

2. `Tell`: Writes a description of an item to some medium.

Indirectly, the coffee dispenser may perform the `Fail` operation when it looks up an item. Thus the type of the dispenser is

$$\texttt{dispenser} : a \xrightarrow{\{\texttt{Ask}:() \to [(a,b)], \texttt{Fail}:() \to b, \texttt{Tell}:b \to c | \rho\}} c$$

We compose the coffee dispenser from the aforementioned operations and the look-up function, e.g.

```
fun dispenser(n) {
  var inv = do Ask();
```

```
    var item = lookup(n,inv);
    do Tell(item)
  }
```

The monadic coffee dispenser model used three monads: `Reader`, `Writer` and `Maybe` to model the desired behaviour. We will implement three handlers which resemble the monads. First, let us implement `Reader`-monad as the handler `reader` whose type is

$$(() \xrightarrow{\{\texttt{Ask}:(a)\to[(\texttt{Int},[|\texttt{Coffee}|\texttt{Tea}|\rho_1|])]\,|\,\rho_2\}} b) \to () \xrightarrow{\{\texttt{Ask}:\alpha|\rho_2\}} b$$

For simplicity we hard-code the inventory into the handler

```
  open handler reader(m) {
    case Ask(_,k)  → k([(1,Coffee),(2,Tea)])
    case Return(x) → x
  }
```

When handling the operation `Ask` the handler simply invokes the continuation `k` with the inventory as parameter. Like in Example 1 we model the inventory as an association list.

Second, we implement the handler `writer` which provide capabilities to write to a medium. We let the medium be a regular string. The handler's type is

$$(() \xrightarrow{\{\texttt{Tell}:[|\texttt{Coffee}|\texttt{Tea}|]\to\texttt{String}\,|\,\rho\}} a) \to () \xrightarrow{\{\texttt{Tell}:\alpha|\rho_2\}} a$$

and its definition is

```
  open handler writer(m) {
    case Tell(Coffee,k) → k("Coffee")
    case Tell(Tea,k)    → k("Tea")
    case Return(x)      → x
  }
```

Here we use pattern-matching to convert `Coffee` and `Tea` into their respective string representations.

Finally, we implement the `lookup` function which given a key and an association list returns the element associated with the key if the key exists in the list, otherwise it discharges the `Fail`-operation to signal failure

```
  fun lookup(n, xs) {
```

```
    switch (xs) {
      case [] → do Fail()
      case ((k, e) :: xs) → if (n == k) { e }
                            else { lookup(n, xs) }
    }
  }
```

To handle failure we reuse the `maybe`-handler from Section 3.2.2 with the slight change that we make it an open handler. Now, we just have to glue all the components together

```
fun runDispenser(n) {
  force(maybe(writer(reader(fun() { dispenser(n) }))))
}
```

Note, that in this example the order in which we compose handlers is irrelevant. Running a few examples we see that it behaves similarly to the monadic version we implemented in Section 1.1.3

```
links> runDispenser(1)
Just("Coffee") : [|Just:String|Nothing|ρ|]

links> runDispenser(2)
Just("Tea") : [|Just:String|Nothing|ρ|]

links> runDispenser(3)
Nothing() : [|Just:String|Nothing|ρ|]
```

□

Observe that when we implemented the monadic version of the `dispenser` using Monad Transformers we had to pay careful attention to the ordering of effects up front because we had to lift certain operations. This issue is no longer present with handlers. In fact, we first defined `dispenser` without considering the concrete the interpretation of the operations `Ask` and `Tell` (and `Fail`). Furthermore, the effect ordering does not leak into the inferred effect row as opposed to Monad Transformers. The effect row typing is pivotal to the modular design afforded by handlers. Programmers can truly implement composable components independently as they do not have to worry about issues such as the shadow issue which is caused by having an ordering on effects.

3.3.2 Reinterpreting Nim

In Section 3.2.3 we gave various interpretations of the game Nim using closed handlers. Example 15 demonstrates how we can use the compositionality of open handlers to extend the game with an additional cheat detection mechanism without breaking a sweat.

We reuse the game model and auxiliary functions from Section 3.3.2.

Example 15 (Cheat detection in Nim)**.** First, we implement a function that given a player, the number of remaining sticks n and the number, and a continuation k determines whether the player cheats. We call this function `checkChoice`, it will perform two operations: `Move` and `Cheat`, the former simulates a particular move whilst the latter operation is used to signal that cheating has occurred. The type of the function is

$$\texttt{checkChoice} : (a, b, \texttt{Int} \xrightarrow{E} c) \xrightarrow{E} c$$

where $E \stackrel{\text{def}}{=} \{\texttt{Cheat} : (a, \texttt{Int}) \to c, \texttt{Move} : (a, b) \to \texttt{Int}|\rho\}$. The following is its implementation:

```
fun checkChoice(player,n,k) {
  var take = do Move(player,n);
  if (take < 1 || 3 < take) { # Cheater detected!
    do Cheat(player,take)
  } else {                    # Otherwise OK
    k(take)
  }
}
```

First, we simulate the player's move. If the player's choice is not in the set of valid moves $\{1, 2, 3\}$ then the function signals that cheating has occurred, otherwise the continuation `k` is invoked to actually perform the move. Now, it is straightforward to implement a handler which uses `checkChoice` to detect cheating, e.g.

```
open handler checkgame(m) {
  case Move((player,n),k) → checkChoice(player,n,k)
  case Return(x)          → x
}
```

Note that the type of `checkgame` is $(() \xrightarrow{E} c) \to () \xrightarrow{E} c$ where the effect row E is the same as above. Hence `checkgame` is itself an abstract computation. Therefore we will need two more handlers which interpret the `Cheat` and `Move` operations. We encode the cheater's strategy into the handler which handles the additional `Move` operation discharged by `checkgame`, e.g.

```
fun cheater(n,k) {
    k(n)
}

open handler aliceCheats(m) {
  case Move((Alice,n),k) → cheater(n,k)
  case Move((Bob,n),k)   → perfect(n,k)
  case Return(x)         → x
}
```

Here a cheater's strategy is simply to take all sticks in the heap and thereby win the game in one single move. In the handler `aliceCheats` we assign the cheater's strategy to Alice whilst Bob plays the perfect strategy. Thus if we play without cheat detection then Alice will always win in a single move because she always starts.

Finally, we interpret the `Cheat` operation by halting the game and reporting the cheater, e.g.

```
open handler cheatReport(m) {
  case Cheat((Alice,n),k) → error("Cheater Alice took "
   ^^ intToString(n) ^^ " sticks")
  case Cheat((Bob,n),k)   → error("Cheater Bob took " ^^
   intToString(n) ^^ " sticks")
  case Return(x)          → x
}
```

Here, we pattern match on the player to determine who cheated. The `error` function halts the game and reports to standard out who cheated along with how many sticks the player took. Now, we can put everything together and try a few examples:

```
fun checkedGame(m) {
  force(aliceCheats(cheatReport(checkgame(m))))
}

links> play(checkedGame, 36);
```

```
*** Fatal error : Cheater Alice took 36 sticks

links> play(checkedGame, 3);
Alice() : [|Alice|Bob|ρ|]
```

Alice still wins when $0 < n \leq 3$ because in this particular game configuration it is a legal move to take all sticks. Moreover, observe that the order in which we compose handlers is important in this example because `checkgame` is itself an abstract computation, therefore if we swap `aliceCheats` and `checkgame` the cheat detection mechanism never gets invoked. Accordingly, Alice would always win because she cheats. □

Like in the previous Nim game examples we changed the strategic behaviour of the players without changing the game model (`aliceTurn` and `bobTurn`), however, in addition in Example 15 we also extended the game mechanics without changing the game model.

Chapter 4

Implementation

The Links compiler is a multi-pass compiler with several distinct phases. Coarsely, we can divide the compiler into two major components the front-end and back-end. We can further subdivide the front-end into

- Parser: Transforms the input source into a syntax tree.
- Early desugar: Performs source-to-source transformations before source analysis.
- Type checker: Analyses the source, performs type inference, and ensures terms are well-typed.

The compiler has more front-end components, but these are the most relevant for our implementation. Similarly, the back-end can be further subdivided

- IR Compiler: Transforms the source into an intermediate representation used by the interpreter.
- Pattern-matching compiler: Aids the IR compiler by compiling pattern-matching constructs into the intermediate representation.

Figure 4.1 provides a high level picture of how the different relevant phases are connected. The subsequent sections discuss implementation specific details.

4.1 Early desugaring of handlers

The `handler` and `open handler` constructs are syntactic sugar. They get desugared into a legacy construct from an early implementation. The initial imple-

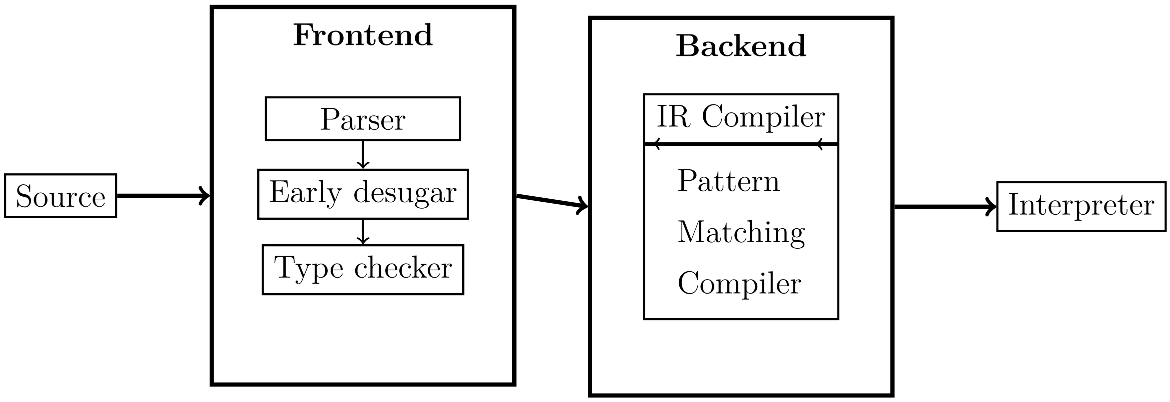


Figure 4.1: Links compiler phases overview.

mentation used a `handle`-construct for handlers. Figure 4.2 displays the conceptual transformation of `handler` to `handle`. This desugaring takes place right after the parsing phase. The early desugaring is beneficial because it allows us to take full advantage of the earlier implementation, whilst providing a more convenient syntax for handlers.

```
handler(m) {
  case Op_i(p_i,k_i) → b_i
  case Return(x) → b
}
```

$\Rightarrow$

```
fun(m) {
  handle(m) {
    case Op_i(p_i,k_i) → b_i
    case Return(x) → b
  }
}
```

Figure 4.2: The `handler`-construct gets desugared into a `handle`-construct where the computation m is abstracted over using a function.

The `open handler`-constructs get desugared in a similar fashion, but, with a small twist: The `handle`-construct gets wrapped inside a thunk. The extra layer of indirection entailed by this transformation is *the key* to make handlers composable. The crucial insight is that by transforming every open handler into a thunk compositionality follows for free because computations are modelled as thunks. Figure 4.3 shows the conceptual transformation for `open handler`-constructs.

```
open handler(m) {
  case Opᵢ(pᵢ,kᵢ) → bᵢ
  case Return(x) → b
}
```

$\Rightarrow$

```
fun(m) {
  fun() {
    handle(m) {
      case Opᵢ(pᵢ,kᵢ) → bᵢ
      case Return(x) → b
} } }
```

Figure 4.3: The `open handler`-construct gets desugared into a thunked `handle`-construct.

4.2 Type checking

The type checker implements the following typing rule for open handlers [7]:

$$\frac{\begin{array}{l} E_{in} \stackrel{\text{def}}{=} \{\mathtt{Op}_i : A_i \to B_i\}_i \uplus \rho \\ E_{out} \stackrel{\text{def}}{=} E_{forward} \uplus \rho \\ H \stackrel{\text{def}}{=} \{\mathtt{Return}(x) \mapsto M\} \uplus \{\mathtt{Op}_i(p,k) \mapsto N_i\}_i \\ (\Gamma, p : A_i, k : U_{E_{out}}(B_i \to C) \vdash_{E_{out}} N_i : C)_i \\ \Gamma, m : A \vdash_{E_{out}} M : C \end{array}}{\Gamma \vdash H : A \stackrel{E_{in}\quad E_{out}}{\Rightarrow} C} \tag{4.1}$$

The rule says, that if a computation m of type A performs effects E_{in}, and the type signatures of the operations handled by the handler H agree with E_{in}, and the return clause has type C, then H handles an effectful computation m with effects E_{in}, and may itself cause effects E_{out} and returns a computation of type M. The typing rule for closed handlers is similar, however, it leaves out the row variable ρ.

4.2.1 Implementation details

The type checker for handlers take advantage of the existing infrastructure for the `switch`-construct which also embodies a collection of `case`-expressions. Figure 4.4 displays the two constructs side-by-side.

In order to determine which operations a handler handles the type checker invokes the type checking procedure for `case`-expressions. This procedure returns a list of the patterns being matched. In the concrete case for handlers the procedure infers that the `case`-expressions pattern match on a variant type. The tags in the variant are precisely the names of the operations that the handler handles. This also reveals why operations resemble variant constructors so closely.

Internally, a variant is represented by a row. So, the handler type checker extracts the row from the inferred variant type, thereafter it applies the typing rule (4.1) to turn obtain the desired effect row.

4.3 Pattern-matching compilation

Syntactically, the `handler`-construct and `switch`-construct are similar. Figure 4.4 depicts their similarities. Notably, their semantics differ as `switch` allows

`handler(m) {`	`switch(e) {`
`case` $\mathtt{Op}_i(\mathtt{p}_i,\mathtt{k}_i) \rightarrow \mathtt{b}_i$	`case` $\mathit{Pattern}_j \rightarrow \mathtt{b}_j$
`case Return(x)` $\rightarrow$ `b`	`case other` $\rightarrow \mathtt{b}'$
`}`	`}`

Figure 4.4: The `handler`-construct resembles the `switch`-construct syntactically.

arbitrary pattern matching on an expression x and `handler` only allows pattern matching on possible operation names in some computation m. Furthermore, `switch` has a default case `other` which is not allowed in `handler`. Their syntactic similarities give rise to a similar internal representation as well. Although, the internal representation of `handler` contains extra attributes such as whether the handler is open or closed. The resemblance has certain benefits:

- Syntactical commonalities makes handlers feel like a natural integrated part in Links,
- and we can reuse the `switch` pattern-matching compilation infrastructure for `handler`.

The `switch` pattern-matching compiler supports deep pattern-matching which we want for matching on actual operation parameters, but only a handful of patterns are permitted for matching on continuation parameters. Figure 4.5 shows the legal pattern-matching on a continuation parameter. Moreover, the `Return`-case must only take one parameter. These small subtleties prevent us from using the `switch` pattern-matching compiler directly.

```
handler(m) {
  case Op_i1(_,k)         → b_i1   # Name binding
  case Op_i2(_,k as c)    → b_i2   # Aliasing
  case Op_i3(_,_)         → b_i3   # Wildcarding
  case Return(x)          → b_i4
}
```

Figure 4.5: Permissible patterns for matching on the continuation parameter.

Instead we embed the `switch` pattern-matching compiler along with a preliminary pattern-matching analyser in the `handler` pattern-matching compiler. The pattern-matching analyser checks that the patterns are legal, i.e.

- An operation-case has at least two parameters, where the last parameter is supposed to be the continuation.
- Pattern-matching on a continuation parameter is either name binding, aliasing or wildcarding.
- `Return`-case(s) only take one parameter.

If the pattern-matching analysis is successful then the `switch` pattern matching compiler is invoked to generate the code. Otherwise, a compilation error, complaining about illegal patterns, is emitted.

4.4 Interpreter

The Links compiler uses A-Normal Form (ANF) as an intermediate representation. In particular, the Links interpreter directly interprets ANF code. ANF is a relatively simple direct-style language which partitions expressions into two classes: atomic expressions and complex expressions. An expression is considered atomic if it is pure, i.e. it causes no effects and it terminates [4]. On the other hand, every complex expression must be assigned a fresh name. For example the Links expression `g(f(h(x)))` gets translated into the Links-ANF computation `({let y = h(x), let z = f(y)}, g(z))` where the first component is a list of `let`-bound intermediate computations, and the second component is a tail computation. Incidentally, it is straightforward to implement first-class control in the source language as the current continuation can be built from the Links-ANF computation. Moreover, the simplicity of ANF makes it amendable as an interpreted language.

The Links interpreter is written in continuation-passing style (CPS) which threads the current continuation directly through the program. The continuation was implemented as a stack of continuation frames which capture computations along with their contexts. Formally, a continuation frame is quadruple $F \stackrel{\text{def}}{=} (\mathcal{S},\mathcal{B},\mathcal{E},\mathcal{C})$ where

- $\mathcal{C}$ is a computation.
- $\mathcal{E}$ is an environment that binds names in $\mathcal{C}$.
- $\mathcal{B}$ is a binder for the computation.
- $\mathcal{S}$ denotes the scope of the computation.

For example the expression above gets encoded as the following continuation frame

$$(\text{scope}(\texttt{y}), \texttt{y}, \text{localise}(\texttt{y}), (\texttt{\{let z = f(y)\}, g(z)}))$$

where scope and localise are two functions, that return the scope of a binder and localises the binder in the current environment, respectively.

This particular notion of continuation is problematic for handlers because we need delimited control for continuations assigned by handlers. Therefore it is necessary to generalise the notion of continuation in the Links interpreter. Fortunately, the generalisation is conceptual simple: Lift the continuation into a stack, i.e. let it become a stack of stacks of continuation frames. In other words the generalised continuation embeds the previous continuation layout. Figure 4.6 illustrates the embedding. This scheme effectively turns every stack of continuation frames into a delimited continuation, i.e. a continuation that returns control to the caller.

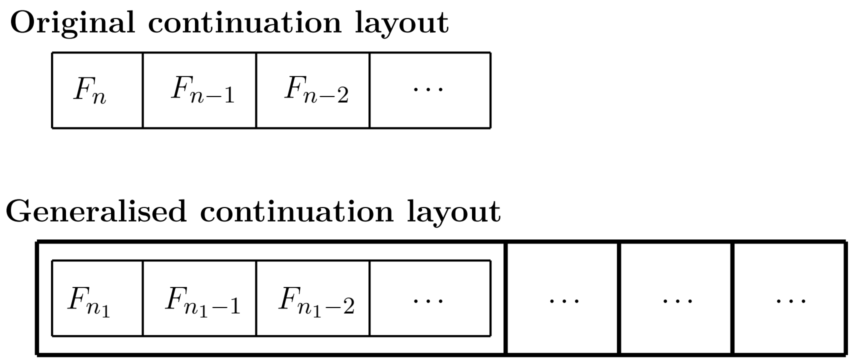


Figure 4.6: The generalised continuation embeds the previous continuation layout.

The generalised continuation is built in parallel with a stack of handlers. Whenever the interpreter encounters a handler, it pushes the handler onto the handlers' stack and allocates a new delimited continuation which is pushed onto the stack inside the generalised continuation. The top-most delimited continuation grows as the evaluation progresses. Conversely, when the top-most delimited continuation is depleted the proper `Return`-case of the top-most handler is invoked. Additionally, both elements are popped from their respective stacks. The evaluation terminates when the entire generalised continuation has been consumed.

Operation invocation follows a rather simple scheme: Upon encountering an operation the interpreter pops and invokes the top-most handler, if the handler does not handle the operation, then the second top-most handler is popped and invoked and so forth. The interpreter maintains the popped handlers in a separate temporary stack along with their corresponding delimited continuations. The temporary stack is a "slice" of the program state which is assigned to the continuation parameter when a matching case is found. If no matching case is found then an "unhandled operation" error is emitted. When the continuation is invoked the "sliced" state is merged back into the program state. This ensures that the `Return`-cases are invoked in the proper order when the handled computation finishes.

Chapter 5

Evaluation

Section 5.1 briefly summarises and evaluates *handlers for algebraic effects using row polymorphism* as an effectful programming model with respect to modularity and compositionality. Section 5.3 evaluates the relative performance cost incurred by handlers.

5.1 Handlers with row polymorphic effects

5.2 Handlers in Links

How it fits in with the rest of Links syntax etc...

5.3 Performance

Since Links is an interpreted language it does not make sense to measure the raw execution speed of handled computations as the overhead incurred by the interpreter is likely to be dominant. Instead, we will measure the relative cost incurred by using handlers.

5.3.1 Benchmarks setup

The experiments were conducted on a standard Informatics DICE Machine[1]. The following three different micro benchmarks were used:

[1]Machine name: Enna. Specifications: Intel Core i5-4570 3.20 Ghz, 8 GB Ram, Scientific Linux 6.6 (Carbon) running Linux kernel 2.6.32-504.16.2.el6.x86_64

- Stateful counting: Counting down from 10^7 to 0 using a closed state handler.
- Stateful counting with logging: Counting down from 10^7 to 0 using the state logging handler from Example ??.
- Nim game tree generation: Generation of game tree with starting configuration $n = 20$ using the handler from Example 13.

Each benchmark program has a pure counterpart. For instance pure stateful counting passes the state as a parameter to the counting function. The pure Nim game tree generator is hard-coded to produce game trees under the restrictions explained in Section 3.2.3. The source code for each pure program is listed in Appendix ??.

For each benchmark we take 10 samples. To eliminate noise caused by programs benefiting from cache locality the sampling has been interleaved. That is, first we run a benchmark once to produce one sample, then we run another benchmark to produce one sample, and so forth. This process has been repeated 10 times to produce 10 samples for each benchmark.

The built-in performance-measuring mechanism in the Links interpreter has been used to measure the execution time. The execution time only includes the run time of the program, that is it does *not* include loading up the Links interpreter or program compilation.

5.3.2 Results

Table 5.1 displays the results obtained from the experiments. The intermediate results are listed in Appendix ??.

	Handlers (ms)	Pure (ms)	Relative speed
Stateful counting	19097.10	9629.14	0.50
Stateful counting with log	161458.15	19097.10	0.12
Nim game tree generation	14406.11	814.98	0.06

Table 5.1: Results obtained from the experiments. The handlers and pure columns list the average execution time.

There is a significant increase in execution speed when using handlers. The closed state handler in the first benchmark is roughly twice as expensive as the pure tail recursive counting method. In the second state benchmark the state grows as the computation progresses, the handler version is about 8 times as expensive as the pure tail recursive method. Finally, the game tree generating handler is about 16 times as expensive as the direct, hard-coded game tree generator.

The handler game tree generator is far more general than the pure version as it can generate the game tree for any game, while the pure version only generates for the specific restricted Nim game we used in Section 3.2.3. It may not be completely fair to compare the general handler against the hand-tuned function. Therefore, we have implemented a generic pure game tree generator that makes use of higher-order functions, like the handler version, e.g. `zip`, `map`, etc. The performance results are displayed in Table 5.2. The results are surprising as the handler and generic pure version have almost equal performance.

	Time (ms)	Relative speed
Pure hard-coded	814.98	1.0
Pure generic	12441.01	0.07
Handler	14406.11	0.06

Table 5.2: Comparison of the handler version and two different pure versions of the Nim game tree generation program.

Chapter 6

Conclusion and future work

6.1 Conclusion

6.2 Future work

- Implement shallow and parameterisable handlers.
- Examine handlers in large-scale programming.
- Formal semantics for the Links interpreter.
- Enable handlers in Links web-mode.
- Applications of pure handlers.

Bibliography

[1] 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.

[2] Nick Benton and Andrew Kennedy. "Exceptional Syntax". In: *Journal of Functional Programming* 11.4 (2001), pp. 395–410. DOI: 10.1017/S0956796801004099. URL: http://dx.doi.org/10.1017/S0956796801004099.

[3] 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.

[4] Cormac Flanagan et al. "The Essence of Compiling with Continuations". In: *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*. PLDI '93. Albuquerque, New Mexico, USA: ACM, 1993, pp. 237–247. ISBN: 0-89791-598-4. DOI: 10.1145/155090.155113. URL: http://doi.acm.org/10.1145/155090.155113.

[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] Anker Helms Jørgensen. "Context and Driving Forces in the Development of the Early Computer Game Nimbi". In: *IEEE Annals of the History of Computing* 31.3 (2009), pp. 44–53. ISSN: 1058-6180.

[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] Ohad Kammar and Gordon D. Plotkin. "Algebraic foundations for effect-dependent optimisations". In: *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*. Ed. by John Field and Michael Hicks. ACM, 2012, pp. 349–360. ISBN: 978-1-4503-1083-3. DOI: 10.1145/2103656.2103698. URL: http://doi.acm.org/10.1145/2103656.2103698.

[9] 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`.

[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. "Algebraic effects and effect handlers for idioms and arrows". In: *Proceedings of the 10th ACM SIGPLAN workshop on Generic programming, WGP 2014, Gothenburg, Sweden, August 31, 2014*. Ed. by José Pedro Magalhães and Tiark Rompf. ACM, 2014, pp. 47–58. ISBN: 978-1-4503-3042-8. DOI: `10.1145/2633628.2633636`. URL: `http://doi.acm.org/10.1145/2633628.2633636`.

[12] 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.

[13] Erik Meijer. "The Curse of the Excluded Middle". In: *Queue* 12.4 (Apr. 2014), 20:20–20:29. ISSN: 1542-7730. DOI: `10.1145/2611429.2611829`. URL: `http://doi.acm.org/10.1145/2611429.2611829`.

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

[15] Gordon D. Plotkin and John Power. "Adequacy for Algebraic Effects". In: *Foundations of Software Science and Computation Structures, 4th International Conference, FOSSACS 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6, 2001, Proceedings*. Ed. by Furio Honsell and Marino Miculan. Vol. 2030. Lecture Notes in Computer Science. Springer, 2001, pp. 1–24. ISBN: 3-540-41864-4. DOI: `10.1007/3-540-45315-6_1`. URL: `http://dx.doi.org/10.1007/3-540-45315-6_1`.

[16] Gordon D. Plotkin and John Power. "Semantics for Algebraic Operations". In: *Electr. Notes Theor. Comput. Sci.* 45 (2001), pp. 332–345. DOI: `10.1016/S1571-0661(04)80970-8`. URL: `http://dx.doi.org/10.1016/S1571-0661(04)80970-8`.

[17] 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`.

[18] 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.

[19] 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`.

[20] Niki Vazou and Daan Leijen. *Remarrying effects and monads*. Submitted to ICFP '15. `http://goto.ucsd.edu/~nvazou/koka/icfp15.pdf`. 2015.

[21] 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.

[22] Philip Wadler and Peter Thiemann. "The marriage of effects and monads". In: *ACM Trans. Comput. Log.* 4.1 (2003), pp. 1–32. DOI: `10.1145/601775.601776`. URL: `http://doi.acm.org/10.1145/601775.601776`.

[23] 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`.