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

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 [12]. 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 [7, 3, 12].

Plotkin and Pretnar's *handlers for algebraic effects* [16] 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 → [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 [20], 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 → 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 [21].

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 [7]. 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 [6], 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 1.1 (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.

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 1.1 using Monad Transformers.

Example 1.2 (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 1.1 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 1.1. 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 `lifts` 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 1.2 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 [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.

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 [13]:

Example 1.3 (Importance of effect ordering [13])**.** 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 [16] 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 [6] and Wu et. al [22].

Data types á la carte

Swierstra [18] 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 [18], whilst the other is a continuation-based approach [6].

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 [6]. 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 [6].

Handlers in scope

Wu et. al investigate how to use handlers to delimit the scope of effects [22] 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 [22]. 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 [22]. 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 [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 [9]. It supports arbitrary user-defined effects [19]. 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 [15, 16]. Plotkin and Power [14, 15] 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 [10]. For example, we might define an algebraic effect `Choice` for making a boolean choice with the following signature:

$$\texttt{Choice} \stackrel{\text{def}}{=} \{\texttt{Choose} : () \to \texttt{Bool}\}$$

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 [16] 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 [10]. 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-

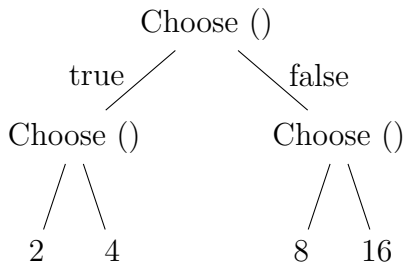
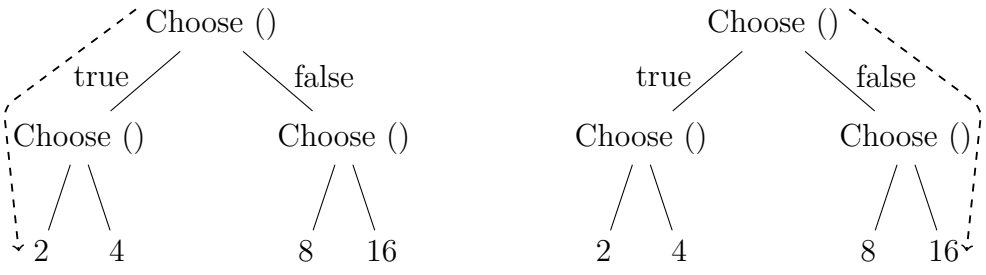


Figure 2.1: Interpretation of the conditional expression as a computation tree. The left edges correspond to taking the first branch in a conditional expression. Analogously, the right edges correspond to taking the second branch.

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 [6].



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

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

Figure 2.2: Two different interpretations.

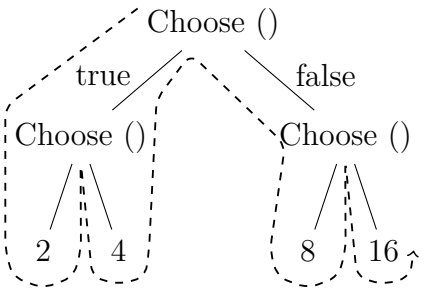


Figure 2.3: Enumerate all possible choices. Output: $\{2, 4, 8, 16\}$.

2.2 Row polymorphism

Row polymorphism is a typing discipline for row-based types such as records and variants [17]. A row is an unordered collection of labels, e.g.

$$\texttt{Student} \stackrel{\text{def}}{=} \langle \texttt{name} : \texttt{String}, \texttt{course} : \texttt{String} \rangle$$

denotes a row type with two fields: name of type `String` and course of type `String`. The record $\texttt{daniel} \stackrel{\text{def}}{=} \langle \texttt{name} = \text{"}Daniel\text{"}, \texttt{course} = \text{"}cs\text{"} \rangle$ is a possible instance of the row type `Student`.

Consider a function which returns the projection of the `name` component and the record itself, e.g. $\texttt{get\_name}(r) \stackrel{\text{def}}{=} (\pi_{\texttt{name}}(r), r)$, where $\pi_l(r)$ is the projection of label l in row r. This raises the question of how to type the function. One possible typing is

$$\texttt{get\_name} : \langle \texttt{name} : \texttt{a} \rangle \to \texttt{a} \times \langle \texttt{name} : \texttt{a} \rangle$$

The type looks innocuous, however, assuming that the type `Student` is a subtype of $\langle \texttt{name} : \texttt{String} \rangle$, then

$$\texttt{get\_name}(\texttt{daniel}) = (\text{"}Daniel\text{"}, \langle \texttt{name} = \text{"}Daniel\text{"} \rangle)$$

The output lost the `course` field! Because subtyping subsumed the field. However using row polymorphism, we can prevent the loss of information. The principal idea is to extend row types with a row variable ρ which can be instantiated with additional fields, thus we may type `get_name` as

$$\texttt{get\_name} : \langle \texttt{name} : \texttt{a} \mid \rho \rangle \to (a \times \langle \texttt{name} : \texttt{a} \mid \rho \rangle)$$

The row $\langle \texttt{name} : \texttt{a} \mid \rho \rangle$ is said to be *open* due to the presence of ρ, conversely, the row `Student` is said to be *closed*. In addition row polymorphism equip field types

with a presence flag which indicating whether a field is *present* or *absent* [17]. We will denote presence by $pre(\tau)$ and absence by $abs(\tau)$ where τ is the type of the field in question. If we attempt to apply `get_name` to `daniel` under this typing then we get `("Daniel", daniel)` as desired. Under the hood the type system has to solve the equation

$$\langle \texttt{name} : pre(\texttt{a}) \mid \rho \rangle \sim \langle \texttt{name} : pre(\texttt{String}), \texttt{course} : pre(\texttt{String}) \rangle$$

that is the two record types must be unified. Note that `get_name` reads all fields in the input record r, therefore all fields must be present in the solution. The solution is to first instantiate the type variable a with `String`, and then instantiate ρ with the additional field `course` of type $pre(\texttt{String})$. The result is a row that is structural equal to the row `daniel`. It is crucial that the left hand side row is open, otherwise the equation would have no solution.

As a final example, consider a function `graduate` which takes a record of type $\langle \texttt{name} : pre(a), \texttt{course} : \theta_1, \texttt{age} : \theta_2 \rangle$ as input and only reads the `name` field. Here the types θ_1 and θ_2 are used to denote that the row is polymorphic over the presence of the fields `course` and `age`. Since the `Student` contains field `name` of type `String` we should be able to apply `graduate` to the record `daniel` which gives rise to the following equation

$$\langle \texttt{name} : pre(\texttt{a}), \texttt{course} : \theta_1, \texttt{age} : \theta_2 \mid \rho \rangle \sim \langle \texttt{name} : pre(\texttt{String}), \texttt{course} : pre(\texttt{String}) \rangle$$

Again, we unify field by field: The case for `name` is easy, we simply instantiate the type variable a with `String`. Next, we instantiate θ_1 with $pre(\texttt{String})$ because `course` is present on the right hand side. Finally, because `age` is not present on the right hand side, and not used in the function we instantiate θ_2 with abs. This is safe because `age` is never accessed.

If `age` *was* accessed then the above equation would have no solution as the left hand side would require `age` to be present, but as it is absent from the right hand side there would be no way to instantiate it with a concrete type.

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 handler assigns semantics to abstract operations. In Links, this is reflected in the syntax as a handler embodies a collection of pattern-matching cases, which map operation names to computations, e.g.

```
handler h(m) {
  case Get(p,k)  → m_Get
  case Put(p,k)  → m_Put
  case Return(x) → m_Return
}
```

The above is an example handler `h` that handles two operations `Get` and `Put`. In the previous section we said that every operation takes exactly one argument, yet, an operation case matches on two parameters. The first parameter `p` is the operation argument, whilst the second parameter `k` exposes a delimited continuation that accepts a single parameter. Invocation of the continuation transfers control back to the handled computation `m` at the point where the said operation was discharged. Both parameters may be referenced multiple times in their respective case-computation $m_{\texttt{Get}}$ or $m_{\texttt{Put}}$. There may be a variable number of operation-cases, however, there must be at least one `Return`-case in every handler. The `Return`-case is a special case that is implicitly invoked when the handled computation `m` finishes. The purpose of `Return` is discussed in Section 3.2.2.

The formal parameter `m` is a name for the abstract computation which the handler interprets. Because Links employ a strict evaluation strategy computations are modelled as thunks, that is, the type of `m` is $() \xrightarrow{E} b$ where E is the set of operations that `m` may discharge.

3.2.1 Typing closed handlers

A closed handler handles a fixed set of effects, that is, it puts 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` above has the following type

$$(() \xrightarrow{\{\texttt{Get}:a_1 \to a_2, \texttt{Put}:a_3 \to a_4\}} b) \to c$$

where b is the return type of the computation `m`, and c is the type of $m_{\texttt{Return}}$, $m_{\texttt{Get}}$ and $m_{\texttt{Put}}$. The absence of a row variable in the effect signature implies

that the computation `m` may not perform any other effects than `Get` and `Put`. It is considered a type error to attempt to handle a computation whose effect signature is not unifiable with the effect signature of `m`.

This restriction introduces slack into the type system [5]. To illustrate the slack consider the following computation

```
fun comp() {
  s = do Get();
  if (false == true) { do Foo(); s }
  else { true }
}
```

The computation `comp` has type $() \xrightarrow{\{\texttt{Get}:()\to\texttt{Bool},\texttt{Foo}:()\to()\mid\rho\}} \texttt{Bool}$. Obviously, `Foo` never gets discharged. However, attempting to handle `comp` with the handler `h` gives rise to the following unsolvable equation

$$\{\texttt{Get} : a_1 \to a_2, \texttt{Put} : a_3 \to a_4\} \sim \{\texttt{Get} : () \to \texttt{Bool}, \texttt{Foo} : () \to () \mid \rho\}$$

There is no solution as we cannot remove `Foo` from the right hand side.

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

3.2.2 Transforming the results of computations

The first two 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 3.1 (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 $(() \to a) \to a$ and it is defined as

```
handler force(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. The `force` handler also runs side-effecting computations:

```
links> force(fun() { print("Hello World") });
"Hello World"
() : ()
```

The `force` handler's effect row implicitly contains the wild effect. Without the presence of the wild effect closed handlers would not be able to run computations that might diverge. Moreover, many of the higher-order functions in Links have non-empty effect rows. Thus disallowing the wild effect would severely limit the class of computations that a closed handler can accept as input. □

The next example demonstrates an actual transformation.

Example 3.2 (The listify handler)**.** The `listify` handler transforms the result of a handled computation into a singleton list. Its type is $(() \to a) \to [a]$ and its definition is straightforward

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

Running it on a few examples we obtain:

```
links> listify(fortytwo);
[42] : [Int]
links> listify(hello);
["Hello"] : [String]
links> listify(fun() { [1,2,3] });
[[1,2,3]] : [[Int]]
```

This example also illustrates that the `Return`-case serves a similar purpose to the monadic `return`-function in Haskell whose type is $\texttt{return} : a \to m\,a$ for a monad m. It "lifts" a value into a monadic value, similarly, the `Return`-case lifts a value into a "handled" value. □

In a similar fashion to the handler `listify` in Example 3.2 we can define handlers that increment results by 1, perform a complex calculation using the result of the computation or wholly ignore the result. However, it must ensure that the type of the output is compatible with the output type of the handler. In the case for `listify` the output must be a list of whatever the computation yielded.

3.2.3 Exception handling

Until now we have only seen some simple transformations. Let us make things more interesting. Example 3.3 introduces our first 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 3.3 (The maybe handler)**.** The `maybe` handler handles one operation $\texttt{Fail} : a_1 \to a_2$ that can be used to indicate that something unexpected has occurred in a computation. The handler returns `Nothing` when `Fail` is raised, and `Just` the result when the computation succeeds, thus its type is

$$(() \xrightarrow{\{\texttt{Fail}:a_1 \to a_2\}} b) \to \texttt{Maybe}(b).$$

It is defined as

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

When a computation discharges `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() : Maybe(String)
```

and if the computation succeeds it wraps the result inside a `Just`, e.g.

```
fun success() {
  true
}
links> maybe(success);
Just(true) : Maybe(Bool)
```

□

3.2.4 Handling choice

In Section 2.1.3 we visualised some interpretations the abstract computation

```
fun choice() {
  if (do Choose()) {
    if (do Choose()) { 2 }
    else { 4 }
  } else {
```

```
    if (do Choose()) { 8 }
    else { 16 }
}
```

which uses the operation `Choice` : $() \to$ `Bool`. For completeness we show how to implement these interpretations in Links. Example 3.4 shows the positive interpretation and Example 3.5 shows the enumeration interpretation.

Example 3.4 (The "positive" interpretation)**.** Whenever the operation `Choose` is discharged in the computation `choice` the handler has to decide whether to pick `true` or `false`. Therefore the type of the handler `positive` must be $(() \xrightarrow{\{\texttt{Choose}:()\to\texttt{Bool}\}} a) \to a$. The `positive` handler always picks `true`. To implement this behaviour, we invoke the continuation once with argument `true`. The value `true` becomes the concrete output of `do Choose` in the computation:

```
handler positive(m) {
  case Choose(_,k) → k(true)
  case Return(x)   → x
}
```

Running the handler on the computation `choice` yields the expected result:

```
links> positive(choice);
2 : Int
```

The definition of the handler `negative` from Section 2.1.3 is analogous to `positive`.

□

Example 3.5. The handler `enumerate` traverses the entire computation tree as shown in Figure 2.3. To encode this behaviour we will invoke the continuation twice: First with `true` and then with `false`. The results of both invocations have to be collected in a list. It is the job of `Return` to lift a single result into a list. Therefore the type of `enumerate` is $(() \xrightarrow{\{\texttt{Choose}:()\to\texttt{Bool}\}} a) \to [a]$. The `Return`-case lifts a single element into a singleton list. Hence the two invocations of the continuation give us two lists which we can join together to form a single list, e.g.

```
handler enumerate(m) {
  case Choose(_,k) → k(true) ++ k(false)
  case Return(x)   → [x]
}
```

Applying `enumerate` to the computation `choice` yields the result we arrived at in Section 2.1.3:

```
links> enumerate(choice);
[2, 4, 8, 16] : [Int]
```

□

3.2.5 Interpreting Nim

The in previous examples built intuition for how handlers work. In this section we will use the mathematical game Nim to demonstrate the power of modularity afforded by handlers. Nim is strategic game in which two players take turns to pick sticks from heaps on a table, and whoever takes the last stick wins. 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. Any player may only take between one and three sticks at a time. Furthermore, we name players: Alice and Bob, and Alice always starts. Our model is adapted from Kammar et. al [6].

We encode the players as the sum type $\texttt{Player} \stackrel{\text{def}}{=} [|\texttt{Alice}|\texttt{Bob}|]$ and model the game as two mutual recursive abstract computations, e.g.

Alice plays	Bob plays

`fun aliceTurn(n) {`
`  if (n == 0) { Bob }`
`  else {`
`  var take = do Move((Alice,n));`
`  var r = n - take;`
`  bobTurn(r)`
`} }`
	`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 checks 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 uses one abstract operation `Move` which has the type $\texttt{Move} : (\texttt{Player}, \texttt{Int}) \to \texttt{Int}$, i.e. it takes the current game configuration as input:

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. At this point `Move` does not have a clear semantic interpretation. We only know that its range, the integers, is infinite, so `take` may be assigned any integer value. The types of `aliceTurn` and `bobTurn` are

$$\texttt{Int} \xrightarrow{\{\texttt{Move}:(\texttt{Player},\texttt{Int})\to\texttt{Int}\}} \texttt{Player}$$

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 3.6 (A naïve strategy)**.** A very naïve strategy is to take only *one* stick at every turn. We encode this behaviour by invoking the continuation with argument 1. This assigns 1 to `take` in `aliceTurn` or `bobTurn` depending on whom discharged `Move`. The implementation of the handler is straightforward

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

The operation `Move` is handled uniformly, that is, independent current game configuration the handler always invokes the continuation `k` with 1.

A moment's thought reveals 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, 9);
Alice() : Player

links> play(naive, 18);
Bob() : Player

links> play(naive, 101);
Alice() : Player
```

□

Example 3.7 (A perfect strategy [6])**.** A perfect strategy makes an optimal move at each turn. In particular, 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.

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

Notice that this time we pattern match on `Move`'s argument to obtain the game configuration. By running some examples we witness that Alice wins whenever n is not divisible by four:

```
links> play(pvp, 9);
Alice() : Player

links> play(pvp, 18);
Alice() : Player

links> play(pvp, 36);
Bob() : Player
```

□

Example 3.8 (Mixed strategies)**.** A strategy often encountered in game theory is *mixing* which implies a player randomises its strategies in order to confuse the opponent. In similar fashion to `perfect` from Example 3.7 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 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 `mixed` strategy handler is similar to perfect-vs-perfect handler from Example 3.7

```
handler mixed(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(mixed, 9);
Bob() : Player

links> play(mixed, 9);
Alice() : Player
```

□

Example 3.9 (Brute force strategy [6])**.** Examples 3.6-3.8 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 all illegal moves for a given game configuration n. 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 moves in the current game configuration. Accordingly, the function 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`:

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

Here we take advantage of 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 happens to be the 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() : Player

links> play(bfvp, 18);
Alice() : Player

links> play(bfvp, 36);
Bob() : Player
```

□

Although, the `bruteForce` strategy is significantly slower than `perfect` strategy in Example 3.9 the point of interest here is not efficiency but rather modularity. However, remark that during Examples 3.6-3.9 the game model remained unchanged. We interpreted the game by instantiating the operation `Move` with different implementations. Moreover, Example 3.9 nicely demonstrated that we may exchange two observable equivalent implementations (handlers) effortlessly. In practical terms this implies that one would be able to exchange a slow component with a faster, improved version effortlessly. The coupling between the game model and the handlers is low as they interface through the abstract operation `Move`.

Examples 3.6-3.9 gave different interpretations of the same game. Furthermore, they all computed the same thing, namely, the winner. In particular, each handler applied the identity transformation in the `Return`-case. However, by taking full advantage of the `Return`-case we can use handlers to compute data from computations. For example we can construct the game tree for a Nim game as Example 3.10 shows.

Example 3.10 (Game tree generator [6])**.** In a game tree a node represents a particular player's turn, and outgoing edges corresponds to particular moves that

the player may perform. A path down the game 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 illustrates 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.

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

Leaves are tagged with `Winner`, while nodes are tagged with `Take`. A `Take` node embeds a tuple where the first component is current player, and the second component contains the possible subgames. 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 subgames = zip([1..length(moves)], moves);
  Take(player, subgames)
}
```

First, we map the continuation over the possible moves in the current game configuration to enumerate the subsequent game trees. We compute the subgames by zipping element-wise the two list $\{1, \ldots, |\texttt{moves}|\}$ and `moves`. Finally, we construct a node `Take` with `player` and the possible subsequent game trees.

The leaves are constructed by the `Return`-case in the handler:

```
handler gtGen(m) {
  case Move((player,n),k) → reifyMove(player,n,k)
  case Return(x)          → Winner(x)
}
```

Again, we take advantage of full pattern-matching to decompose the game configuration. The inferred type for `gtGen` witnesses that the handler indeed constructs a game tree:

$$\texttt{gtGen} : (() \xrightarrow{\{\texttt{Move:(Player,Int)}\to\texttt{Int}\}} \texttt{Player}) \to \texttt{GameTree}$$

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

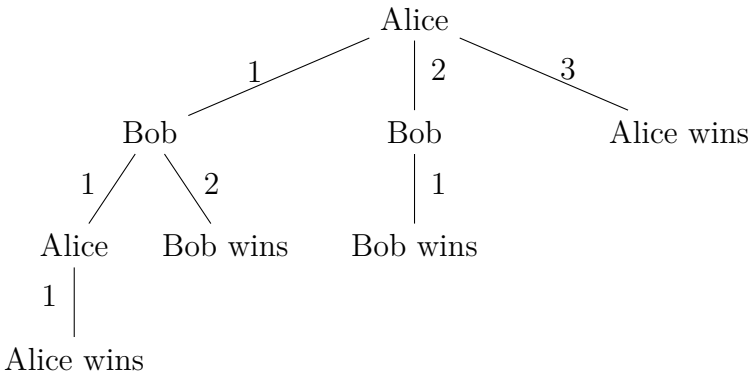


Figure 3.2: Pretty print of the game tree generated by `play(gtGen, 3)`.

□

Notice that even though the game model remains unchanged we have been able to encode strategic behaviours and generate data from the game by interpreting the game using different handlers. This emphasises the modularity afforded by handlers.

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 prepends the keyword `open` in the handler declaration, e.g.

```
open handler h1(m) {
  case Get(p,k)  → mGet
  case Put(p,k)  → mPut
  case Return(x) → mReturn
}
```

The type of the open handler $\mathtt{h}_1$ is more complex than its closed counterpart:

$$\mathtt{h} : (() \xrightarrow{\{\mathtt{Get}:a_1 \to a_2, \mathtt{Put}:a_3 \to a_4 \mid \rho\}} b) \to () \xrightarrow{\{\mathtt{Get}:\theta_1, \mathtt{Put}:\theta_2 \mid \rho\}} c$$

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 operations in the output effect row are polymorphic over their presence.

3.3.1 Composing handlers

It is no coincidence that the input type and output type of open handlers are both thunks. Both types are compatible with the notion of computation for handlers. Therefore, we can compose open handlers seamlessly. For example, imagine a handler h_2 whose signature is

$$h_2 : (() \xrightarrow{\{\texttt{Choose}:a_1' \to a_2' \mid \rho'\}} b') \to () \xrightarrow{\{\texttt{Choose}:\theta_3 \mid \rho'\}} c'$$

The composition $(h_2 \circ h_1)(m)$ gives rise to the following equation

$$\{\texttt{Choose} : a_1' \to a_2' \mid \rho'\} \sim \{\texttt{Get} : \theta_1, \texttt{Put} : \theta_2 \mid \rho\}$$

which has the following solution

$$\{\texttt{Choose} : a_1' \to a_2', \texttt{Get} : \theta_1, \texttt{Put} : \theta_2 \mid \rho'\}$$

The solution encompasses the three fields, where `Get` and `Put` remains polymorphic in their presence unless they are discharged by either handler. It is worth to emphasis that the row variable ρ' is shared by the input and output effect row. The implication is that the additional operation names are propagated throughout composition. Hence $(h_2 \circ h_1)(m)$ ultimately yields a computation with type

$$() \xrightarrow{\{\texttt{Get}:\theta_1,\texttt{Put}:\theta_2,\texttt{Choose}:\theta_3 \mid \rho'\}} b'$$

This is under the assumption that neither handler discharges any of the operations, otherwise the said operations would have to be *present* in the effect row. To see why the operations are presence polymorphic in the output effect row, consider a closed handler $h_3 : (() \to b') \to b'$ then the composition $(h_3 \circ h_2 \circ h_1)(m)$ gives rise to the equation

$$\emptyset \sim \{\texttt{Get} : \theta_1, \texttt{Put} : \theta_2, \texttt{Choose} : \theta_3 \mid \rho'\}$$

Since we cannot shrink rows, there can only be one solution which is

$$\{\texttt{Get} : abs, \texttt{Put} : abs, \texttt{Choose} : abs\}$$

If the operations were not presence polymorphic then we would not be have to compose an open handler with a closed handler.

Implicit handler stack

The order of composition implicitly defines a stack of handlers. For example the composition of three handlers $(g_1 \circ g_2 \circ g_3)(m)$ applied to some computation m defines a stack where g_3 is the top-most element. Thus the handler stack is built outside in. When an operation is discharged in m the runtime system unwinds the handler stack to find a suitable candidate to handle the operation. The composition order determines the order in which handlers are invoked. 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, g_1 and g_2 interpret the same operation differently, then, $g_1 \circ g_2$ and $g_2 \circ g_1$ potentially yield different results. In other words composition is not a commutative.

The result of an application of some open handler to some computation is itself a computation. For example $(g_1 \circ g_2 \circ g_3)(m)$ yield some nullary function $() \to a$ which we must invoke to obtain the result of the computation m. In order to avoid this extra invocation recall the `force` handler from Section 3.2.2. We may compose `force` with the open handlers to obtain the result of m directly, e.g. $(\texttt{force} \circ g_1 \circ g_2 \circ g_3)(m)$ yields a value with type a immediately.

3.3.2 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 1.1 and 1.2). 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 highly modular coffee dispenser model in Links. Example 3.11 implements the model.

Example 3.11 (Coffee dispenser)**.** The coffee dispenser performs two operations directly

1. `Ask`() → `Inventory` retrieves the inventory.
2. `Tell` : `Dispensable` → `String` 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

$$\texttt{dispenser} : \texttt{Int} \xrightarrow{\{\texttt{Ask}:()\to\texttt{Inventory},\texttt{Fail}:()\to a,\texttt{Tell}:\texttt{Dispensable}\to\texttt{String}\mid\rho\}} \texttt{String}$$

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

$$((() \xrightarrow{\{\texttt{Ask}:a\to\texttt{Inventory}\mid\rho\}} b) \to () \xrightarrow{\{\texttt{Ask}:\theta\mid\rho\}} b$$

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 1.1 we model the inventory as an association list.

Secondly, 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

$$((() \xrightarrow{\{\texttt{Tell}:\texttt{Dispensable}\to\texttt{String}\mid\rho\}} a) \to () \xrightarrow{\{\texttt{Tell}:\theta\mid\rho\}} a$$

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, the `lookup` function traverses an association list in order to find the element associated with the given key. If the key exists, then the element is returned, otherwise the `Fail` operation is discharged to signal failure. We will not show its implementation here. To handle failure we reuse the `maybe`-handler from Section 3.2.3 with the slight change that we make it an open handler. Now, we can glue 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") : Maybe(String)

links> runDispenser(2)
Just("Tea") : Maybe(String)

links> runDispenser(3)
Nothing() : Maybe(String)
```

□

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.

3.3.3 Reinterpreting Nim

In Section 3.2.5 we gave various interpretations of the game Nim using closed handlers. Example 3.12 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 3.3.3.

Example 3.12 (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 3.12 we also extended the game mechanics without changing the game model.

3.3.4 Simulating state

We can use handlers to simulate stateful computations, and thereby enabling us to abstract over how state is interpreted. To handle state we need two operations:

- `Get` : $() \to s$ that reads the current state.
- `Put` : $s \to ()$ that updates the state.

We will use the following simple stateful computation to illustrate stateful interpretations:

```
fun scomp() {
  var s = do Get(); do Put(s + 2);
  var s = do Get(); do Put(s + s);
  do Get()
}
```

First, the computation reads the current integer state, then the state is incremented by 2. The new state is then read and doubled. The computation returns the final state. Example 3.13 gives a direct interpretation of state.

Example 3.13 (State handler)**.** Since Links does not have mutable variables we have to find another way to implement state. A pure functional approach is to pass the state around as an explicit parameter. Basically, we adopt this approach to implement state, however, we will introduce an extra layer of indirection to pass the state around. We will abstract over state by encapsulating it inside a function. The function will take a concrete state s as input parameter. For the `Get`-, `Put`- and `Return`-cases the handler returns a new state-encapsulating function. The `state` handler is defined as follows

```
open handler state(m) {
  case Get(_,k)  → fun(s) { k(s)(s)  }
  case Put(p,k)  → fun(s) { k(())(p) }
  case Return(x) → fun(s) { x }
}
```

The `state` handler is partially lazy as when either `Get` and `Put` are discharged the handler returns a single parameter function. Therefore the handler basically suspends the handled computation first time an operation is discharged.

At first glance the `state` handler may seem dubious. Essentially, the handler builds a chain of functions which passes the state around. Let us break it down: The state function returned by the `Get`-case invokes the continuation with the current state which is substituted at the invocation site of `Get`. The continuation returns the next state function to which the same state is passed. The `Put`-case invokes the continuation with unit and passes the modified state `p` as input to the next state function. When a stateful computation finishes the `Return`-case lifts the result into a function, that ignores its argument. Additionally, the function ends the chain of functions as it returns a concrete state rather than a state-function. The type of the `state` handler is

$$(() \xrightarrow{\{\texttt{Get}:()\to s,\texttt{Put}:s\to()\mid\rho\}} a) \to () \xrightarrow{\{\texttt{Get}:\theta_1,\texttt{Put}:\theta_2\mid\rho\}} s \to a$$

where $s \to a$ is the type of a state function. The type variable s is the type of the initial state. In order to execute a stateful computation it is convenient to have a driver function `runState` which abstracts away these details, e.g.

```
fun runState(s0, m) {
  force(state(m))(s0)
}
```

Applying `runState` to some initial state s_0 and `scomp` we obtain:

```
links> runState(0, scomp);
2 : Int
links> runState(-2, scomp);
-2 : Int
links> runState(3, scomp);
8 : Int
```

□

The `state` handler in Example 3.13 returns the most recent state. Incidentally, taking advantage of the composition, we can give a different interpretation which track state changes as Example 3.14 demonstrates.

Example 3.14 (Stateful logging)**.** We extend the `state` handler with a logging capability, however, we will not add this capability directly to the handler. Instead we are going to use composition to construct a stateful handler which keeps track of state changes. The idea is to introduce a new operation $\texttt{LogPut} : s \to ()$ which logs some state of type s. Further, we introduce two new handlers:

- $\texttt{putLogger} : (() \xrightarrow{\{\texttt{LogPut}:s\to(),\texttt{Put}:s\to()\mid\rho_1\}} s) \to () \xrightarrow{\{\texttt{LogPut}:s\to(),\texttt{Put}:s\to()\mid\rho_1\}} s$
- $\texttt{logState} : (() \xrightarrow{\{\texttt{LogPut}:s\to()\mid\rho_2\}} s) \to () \xrightarrow{\{\texttt{LogPut}:\theta\mid\rho_2\}} (s, [s])$

The putLogger handler handles a Put operation, however, it discharge a Put-operation itself along with a LogPut-operation as signified by their presence in the output effect signature. We implement putLogger as follows

```
open handler putLogger(m) {
  case Put(p,k)  → { do LogPut(p); do Put(p); k(()) }
  case Return(x) → x
}
```

In the Put-case the handler first discharges LogPut to log the state change, and then it performs the actual state change by discharging Put. In some sense putLogger acts as a "middleman" because it relies wholly on other handlers to interpret LogPut and Put.

The logState handler builds the log, e.g.

```
open handler logState(m) {
  case LogPut(x,k) → {
        var s = k(());
        var xs = second(s);
        (first(p), (x :: xs))
      }
  case Return(x) → (x, [])
}
```

The handler returns the final state along with a list of previous state changes. In the LogPut-case the handler first invokes the continuation in order to advance the stateful computation. The continuation returns a pair which contains the final state along with a list of changes. The handler preserves the first component, but it extends the second component with the previous state x. In order to handle Get and Put we compose the above handlers with the state handler. Finally, we can reinterpret the computation scomp:

```
links> runState(0, scomp);
(2, [1, 2]) : (Int, [Int])
links> runState(-2, scomp);
(-2, [-1, -2]) : (Int, [Int])
links> runState(3, scomp);
(8, [4, 8]) : (Int, [Int])
```

□

3.3.5 A handler based parsing framework

The state handler enable us to implement abstract, stateful handlers that employ the state operations. In this section we will demonstrate how to implement a simple, but highly modular, backtracking parser as a handler that interprets parser combinators. The result is a small parser framework in Links.

To demonstrate the framework we implement a parser for the simple language PALINDROMES $\stackrel{\text{def}}{=} \{w \in \{a,b\}^* \mid w \text{ is a palindrome}\}$. The language PALINDROMES is generated by the following grammar

$$P ::= a \mid b \mid aPa \mid bPb \mid \varepsilon \tag{3.1}$$

The grammar is simple, yet it contains choice, concatenation, recursion and the empty string (ε) which are sufficient to make an interesting example.

Parser combinators

The principal idea behind parser combinators is to compose parsers from smaller, simpler parsers. The parsers are abstract computations composed using the following three operations:

- `Choose` : $() \to$ `Bool` that makes a nondeterministic choice.
- `Token` : $() \to$ `Char` that consumes a character from the input stream.
- `Fail` : $() \to ()$ that signals failure.

The parsers are implemented as functions whose types are $() \xrightarrow{E} ()$, where E is an open effect row that contains either all, some or none of the above operations. In other words, a parser is a nullary function which may cause several effects, and returns unit. From its signature it is not clear, that a parser does anything sensible. In fact, the purpose of parsers is to capture the structure of a grammar, and we leave the rest of the work to a handler.

The two simplest parsers are `empty` and `char` which accepts the empty string and one particular character, respectively. Their definitions are given below: The

Empty string parser	Single character parser

`fun empty() {`
`  ()`
`}`
	`fun char(c) {`
	`  fun() {`
	`  var t = do Token();`
	`  if (t == c) { () }`
	`  else { do Fail() }`
	`}}`

parser `empty` does nothing, it simply returns unit. The function `char` is not really a parser, but rather a *parser generator*. It takes a character `c` as input, and generates a parser that checks whether `c` is the next character in the input

stream. If `c` is the next character, then the parser returns unit otherwise it signals failure.

We require two more basic parser generators: Choice and sequence. The choice generator takes two parsers as input, and makes a nondeterministic choice between the two. The sequence generator takes two parsers, and applies them in sequence. For syntactic conveniency, we define them as binary operators in Links:

Choice parser

```
op p <|> q {
  fun() {
    if (do Choose())
  { p() }
    else { q() }
  }
}
```

Sequence parser

```
op p <*> q {
  fun() {
    p(); q()
  }
}
```

Choice (`<|>`) generates a parser which discharges `Choose` to decide whether to apply parser `p` or `q`. Sequence (`<*>`) generates a parser which applies parsers `p` and `q` in sequence.

Later, Example 3.15 demonstrates how to compose these four parsers to construct a parser for Palindromes.

Interpreting parsers

The previous section gave the building blocks for constructing parsers. In this section we will implement an abstract, stateful handler which interprets parsers. The handler has to handle the three operations `Choose`, `Token` and `Fail`. Furthermore, it will use the state operations `Get` and `Put` to manipulate the parsing state. The parsing state is a pair $\texttt{PState} \stackrel{\text{def}}{=} ([\texttt{Char}], [\texttt{Char}])$ where the first component contains parsed symbols, and the second component contains the remainder of the input stream. Furthermore, if the input string is not in the language then the `parser`-handler should produce an error, otherwise it should return the parsed string. Thus, the `parser`-handler has the rather involved type

$$(() \xrightarrow{\{\texttt{Choose}:()\to\texttt{Bool},\texttt{Fail}:a_1\to a_2,\texttt{Get}:()\to\texttt{PState},\texttt{Put}:\texttt{PState}\to(),\texttt{Token}:()\to\texttt{Char}\mid\rho\}} b)$$
$$\to () \xrightarrow{\{\texttt{Choose}:\theta_1,\texttt{Fail}:\theta_2,\texttt{Get}:()\to\texttt{PState},\texttt{Put}:\texttt{PState}\to(),\texttt{Token}:\theta_3\mid\rho\}} \texttt{Maybe}(\texttt{[Char]})$$

This type witnesses a cosmetic issue with the implementation, because effects are not explicitly named the effect signature easily blows up and therefore becomes

difficult to read. The parser-handler is more involved than the previous handlers we implemented. Therefore, we name the handler `parser` and implement it one case at a time. The easiest case `Fail` which, unconditionally, returns `Nothing`. We will not show it here, instead we begin with the case for `Token`:

```
case Token(_,k) → {
  var s = do Get();
  var stream = second(s);
  switch (stream) {
    case [] → Nothing
    case (x :: xs) → { do Put((x :: first(s), xs)); k(x) }
} }
```

First the handler retrieves the current parsing state, and then pattern matches on the current state of the input stream. In the event that the input stream is empty the handler returns `Nothing`. Otherwise, it consumes the next character `x` and cons it to the list of parsed symbols. Thereafter the continuation is invoked to return the character `x` to the `token`-parser that discharged `Token`. If the subsequent parsing is successful, then the `k` returns `Just` the result, otherwise it returns `Nothing`. The case for `Choose` follows a similar pattern:

```
case Choose(_,k) → {
  var s = do Get();
  switch (k(true)) {
    case Nothing → { do Put(s); k(false) }
    case Just(x) → Just(x)
} }
```

Again the handler retrieves the current parsing state `s`. Thereafter we pattern match on the result of picking the `true`-branch. If it leads to failure, then we restore the previous state `s` by discharging a `Put`-operation, and then subsequently try the `false`-branch. Further, `k` returns either `Just` the result or `Nothing`. If the choice led to success, then we simply return the identity. Finally, we implement the `Return`-case which is somewhat similar to the `Token`-case, e.g.

```
case Return(x)  → {
  var s = do Get();
  var stream = second(s);
  switch(stream) {
    case []    → Just(reverse(first(s)))
    case other → Nothing
} }
```

Here, we pattern match on the input stream to determine whether all input has been consumed. If the stream is empty, then we return `Just` the reversed list of parsed symbols, otherwise we return `Nothing`. As a final function we implement a convenient driver function `parse` to abstract away the details of running the

parser-handler. The parse function takes a parser p and a string source as input parameters:

```
fun parse(p, source) {
  var s = ([], explode(source));
  switch (runState(parser(p), s)) {
     case Just(r) → Just(implode(r))
     case Nothing → Nothing
} }
```

The function builds the initial state s, where the first component is the empty list, and the second component contains the input string as a list of characters. The switch-expression runs the parser with the initial state, and pattern matches on the result. If the parser was successful, then the list of characters is converted back into a string, otherwise it returns the identity.

Parsing palindromes

The previous two sections provided the basic building blocks for parsing. Now, we are ready to put them into action. Example 3.15 demonstrates how to parse the Palindromes language.

Example 3.15 (Palindromes parser)**.** We implement a parser for the grammar (3.1) using parser combinators. The structure of the resulting parser will closely resemble the structure of the grammar e.g.

```
fun p() {
  var a   = char('a'); var b = char('b');
  var apa = a <*> p <*> a;
  var bpb = b <*> p <*> b;
  var p = apa <|> bpb <|> a <|> b <|> empty;
  p()
}
```

The function p is a parser as its type is $() \xrightarrow{E} ()$. The first line in p constructs two parsers a and b which parses a single character each. Next, the parser apa parses a palindrome which starts with an 'a'. Similarly, bpb parses a palindrome starting with a 'b'. Finally, the parsers are combined to form a parser for the Palindromes grammar (3.1). Notably, the definition of the resulting parser p corresponds closely to the production rule P in grammar (3.1). Note that the parsers in p are combined in a carefully chosen order to cope with the ambiguity of the grammar. Running the parser on a few examples we obtain:

```
links> parse(p, "abba");
Just("abba") : Maybe(String)
links> parse(p, "bbbbaabaabbbb");
```

```
Just("bbbbaabaabbbb") : Maybe(String)
links> parse(p, "");
Just("") : Maybe(String) # The empty palindrome
links> parse(p, "aaabbb");
Nothing() : Maybe(String)
```

□

The entire implementation is less than 100 lines. Yet, the framework is quite general. Shifting all parsing state maintenance to a handler greatly simplifies parser combinators.

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-

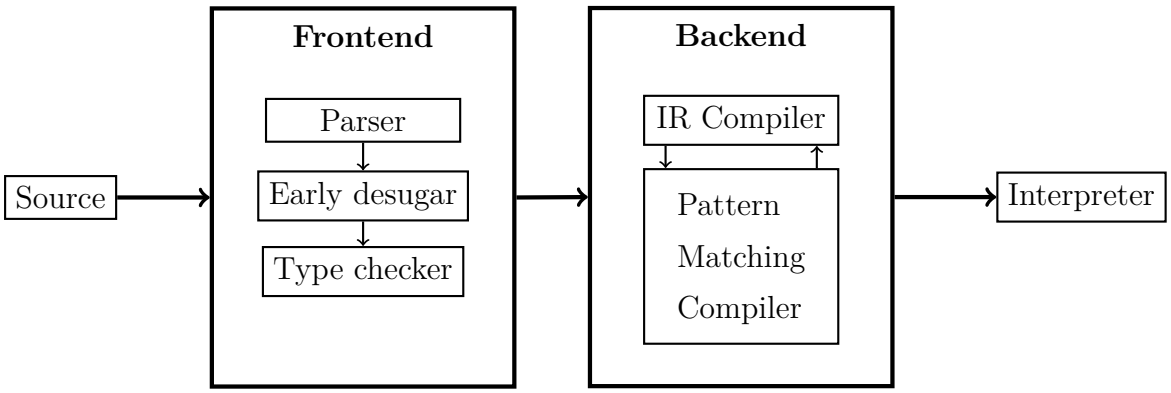


Figure 4.1: Links compiler phases overview.

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
}
```

⇒

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

⇒

```
fun(m) {
  fun() {
    handle(m) {
      case Op_i(p_i,k_i) → b_i
      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 [6]:

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

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 Op_i(p_i,k_i) → b_i            case Pattern_j → b_j
  case Return(x) → b                  case other     → 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_{i_1}(_,k)       → b_{i_1}  # Name binding
  case Op_{i_2}(_,k as c)  → b_{i_2}  # Aliasing
  case Op_{i_3}(_,_)       → b_{i_3}  # Wildcarding
  case Return(x)           → b_{i_4}
}
```

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.

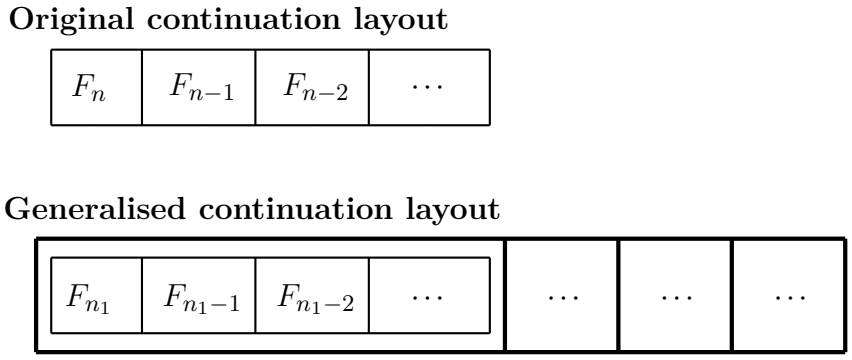


Figure 4.6: The generalised continuation embeds the previous continuation layout.

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

Through a series of examples in Chapter 3 we demonstrated that handlers for algebraic effects indeed afford a high-degree of modularity, and, the compositionality of open handlers enables us to extend the interpretation of an abstract computation effortlessly. In particular, row polymorphism makes programming with effects uniform as it effectively eliminates the ordering issue we discussed in Section 1.1.3.

Note: Handlers themselves are very simply, must require very little code.

5.2 Handlers and user-defined effects in Links

Syntax-wise handlers appear as a well-integrated part of Links because they borrow their syntax from the existent `switch`-construct. Furthermore, handlers are first-class citizens in Links, i.e. a handler can be passed as argument to a function, returned from a function or assigned a name. The first-class property is implicitly inherited from functions, because, essentially handlers desugar into functions. As a result the Links language remains coherent.

User-defined effects exploit Links' structural typing, therefore the programmer

never has to declare effects or operations in advance. The Links compiler automatically infers the type of operations. This fits well with a read-eval-print-loop style of development as employed by the Links top-level interpreter. Unfortunately, it may cause effect signatures to blow up. For example, if a computation is composed from many different operations, or the (closed) handler is recursive. In such cases the Links interpreter infers some verbose operation signatures. The issue can be solved to some extent by annotating operation invocation with an user-defined type alias. Currently, effects are implicitly given by the present operations in the effect row, it would be desirable to have effect-name aliasing, for example something like

$$\texttt{effectname State}(s) = \{\texttt{Get} : () \to s, \texttt{Put} : s \to ()\}$$

This would help condense verbose effect signatures. The current design of operations does not integrate as well with the language as handlers. This is mainly due to two things:

1. Operations use type constructor syntax, but act on values,
2. and operations have to be explicitly discharged using the `do`-primitive.

Neither type constructors nor operations are first-class in Links, but one might expect to able to pass an operation as parameter or return one. If one tries to do this, then the type checker will infer a variant type, rather than an operation, which can steer confusion.

Finally, the `do`-primitive is rather strange in Links, because it is a syntactic construct that does not compose with the rest of the language. A `do` must be immediately followed by an operation name. Moreover, it appears as a prefix operator, but "do" is not a lexicographic valid operator name in Links.

5.3 Relative 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:

- 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 3.10.

Each handler program has two pure counterparts. For the stateful counting benchmarks the first pure version is a direct, tail-recursive implementation which passes the current state as an explicit parameter between invocation. The second pure implementation encapsulates state inside a function in similar fashion to the state handler.

The first pure game tree generator program is a hard-coded, direct implementation for the specific restricted version of Nim we used in Section 3.2.5. The second pure version is more general, and will generate the game tree under any rules. Thus it resembles the handler version. The generality is achieved through use of higher-order functions such as `map`, `zip`, etc.

The benchmark programs were *not* optimised. Each benchmark was sampled ten times. 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 first stateful counting benchmark. The direct, tail-recursive implementation is twice the alternative implementations. A state change does not incur an extra cost, because it is passed as an explicit parameter between invocations. However, the handler implementation and the second pure implementation both use functions to encapsulate state,

[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

therefore each state change causes a new function allocation. The results would suggest that the penalty is high for repetitive allocation of functions.

	Time (ms)	Relative speed
Pure I, tail-recursive	9629.14	1.0
Pure II, function state	20364.6	0.47
Closed handler	14406.11	0.5

Table 5.1: Results obtained from the stateful counting benchmark.

The same seems to be evident for the second stateful counting benchmark. The results are shown in Table 5.2. The tail-recursive pure implementation is significantly faster than the two alternative implementations. It is roughly 3.5 times faster than the pure implementation that encapsulates state inside functions, and further it is about 8 times faster than the handler version. However, this time there is a big difference between the the second pure implementation and the handler implementation. The pure version is little less than twice as fast as the handler version. Possibly, the additional cost is incurred by the handler being open. Each change in state causes two additional operations to be discharged. In addition the handler stack is unwound three times. In particular, three different delimited continuations are invoked during one state change. When a continuation returns control to a handler, it implicitly passes through the return-cases of that particular handler's predecessors (in order to lift the result of the computation). So, the open handler stack incurs a large extra cost.

	Time (ms)	Relative speed
Pure I, tail-recursive	19097.1	1.0
Pure II, function state	68615.1	0.28
Open handler (stack size: 4)	161458.15	0.12

Table 5.2: Results obtained from the stateful counting with logging benchmark.

Finally, in the Nim game tree generator benchmark the hard-coded pure version is superior in terms of execution speed. The handler implementation and generic pure implementation perform, respectively, at 6% and 7% of the hard-coded version. The results are shown in table 5.3. The generic pure implementation is about 14% faster than the handler implementation.

	Time (ms)	Relative speed
Pure I, hard-coded	814.98	1.0
Pure II, generic	12441.01	0.07
Closed handler	14406.11	0.06

Table 5.3: Results obtained from the Nim game tree generation benchmark.

These results suggest that there is plenty of room for optimisations. Kammar et al. achieve better relative performance figures than us [6] with their embedding of handlers in Haskell. However their handlers desugar into monads which the Haskell compiler extensively optimises. Thus, it is likely that with optimisations with could achieve far better performance figures.

Chapter 6

Conclusion and future work

6.1 Future work

- Implement shallow and parameterisable handlers.
- Formal semantics for the Links interpreter.
- Enable handlers in Links web-mode.
- Applications of pure handlers.
- Make handlers practical (performance-wise).

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/S0956796-801004099. 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] 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.

[7] 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.

[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] 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.

[10] 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.

[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] 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.

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

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

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

[20] 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.

[21] 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.

[22] 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.