Compilation of Effect Handlers and their Applications in Concurrency

Daniel Hillerström

Master of Science by Research
CDT in Pervasive Parallelism
School of Informatics
University of Edinburgh
2016

Abstract

Algebraic effects combined with effect handlers have emerged as compelling, modular abstraction for modelling and controlling computational effects. By separating the effect signatures from their implementation, algebraic effects afford a high degree of modularity as programmers can describe effectful computations independently of their concrete interpretation.

We present a compiler for the functional programming language Links that uses the Multicore OCaml backend to provide a native implementation of effect handlers.

We present a core calculus $\lambda^{\rho}_{\text{eff}}$ with row-polymorphic effects and effect handlers based on a variation of A-normal form used in our implementation. In addition, we give an operational semantics for the calculus. Furthermore, we describe a translation from $\lambda^{\rho}_{\text{eff}}$ to a subset of the intermediate language used by OCaml.

Interestingly, concurrency can be described as an algebraic effect, whose handler amounts to a scheduler. Thus, rather than baking concurrency support into the compiler, we keep the compiler lean by implementing the message-passing concurrency model of Links using handlers. We demonstrate a faithful encoding of the concurrency model, which maintains type-safe communication by taking advantage of the effect system of Links.

Finally, we perform some experiments with the compiler using the concurrency implementation to see how it performs against the Links interpreter. Moreover, we consider how to improve the performance of the compiler.

Acknowledgements

This year has been intensive but also incredibly educational and joyful experience. I owe my gratitude to a lot of people for making this year a memorable experience.

First and foremost, I would like to thank my supervisors, Christophe Dubach and Sam Lindley, who both provided me with invaluable guidance throughout this year. Furthermore, I would also like to thank my colleagues from Office 1.07 in Informatics Forum with whom I have enjoyed a lot of fun moments.

A substantial part of this work was done during my visit to OCaml Labs, University of Cambridge. In particular, I would like to thank KC Sivaramakrishnan for demystifying the OCaml backend for me, and for many interesting discussions on applications of effect handlers. Also, I would like to thank Anil Madhavapeddy for hosting me at Pembroke College during my first visit. I would also like to thank Gemma Gordon for helping sorting our the details of my visits, and for our many cheerful conversations.

This work was supported in part by the EPSRC Centre for Doctoral Training in Pervasive Parallelism, funded by the UK Engineering and Physical Sciences Research Council (grant EP/L01503X/1) and the University of Edinburgh. This work was also supported by OCaml Labs, University of Cambridge.

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. Some of the material used in this thesis has been published in the following papers:

- Daniel Hillerström and Sam Lindley, "Liberating Effects with Rows and Handlers", to appear at Type-Driven Development (TyDe), September 2016, Nara, Japan.

- Daniel Hillerström, Sam Lindley, and KC Sivaramakrishnan, "Compiling Links Effect Handlers to the OCaml Backend", to appear at the ML Workshop, September 2016, Nara, Japan.

In addition, some of the material used in this thesis has appeared in the following research competition:

- Daniel Hillerström, "First-Class Message-Passing Concurrency with Handlers", to be presented at Student Research Competition, International Conference on Functional Programming (ICFP), September 2016, Nara, Japan.

(*Daniel Hillerström*)

Contents

Chapter 1

Introduction

Computational effects are pervasive. They arise naturally in many different shapes such as exceptions, file i/o, global state mutation, concurrency, non-determinism, and so forth. Many mainstream programming languages provide little or no support for controlling computational effects – in most cases control is limited to exception handling. Recently *algebraic effects* (Plotkin and Power, 2001, 2003) combined with *effect handlers* (Plotkin and Pretnar, 2013) have emerged as a modular abstraction for expressing and managing user-defined computational effects. By separating the expression of an effectful computation from its implementation, algebraic effects and handlers allow programmers to express effectful computations independently of their concrete interpretation (Kammar et al., 2013). In this system, concurrency arise as just another computational effect, modularly expressed as a signature of *abstract operations* whose implementations are given by one or more handlers. This allows programmers to define their own schedulers for concurrent programs (Dolan et al., 2015) rather than being stuck with a built-in, specially compiler-supported implementation.

In previous work, we extended the functional programming language Links with algebraic effects and handlers (Hillerström, 2015). Links is a full-fledged strict, single-source programming language for multi-tier web-programming. It comprise three backends:

1. a JavaScript compiler for client-side code,
2. an interpreter for the server,
3. and an SQL generator for the database.

The source language is compiled to a common intermediate representation (IR). For the client, the IR is compiled to JavaScript. For the server, the IR is interpreted using a variant of the CEK machine (Felleisen and Friedman, 1987; Hillerström and Lindley, 2016). For the database, the IR is translated into an SQL query. In addition, Links has built-in support for concurrency. The concurrency model of Links is a variant of the actor model (Hewitt and Baker, 1977) with type-safe interaction amongst processes.

In this dissertation we present an extension to the Links infrastructure: a native backend with support for effect handlers (Hillerström et al., 2016). In order to keep the compiler lean we separate the implementation of the concurrency model of Links from the compiler. Instead we implement the message-passing concurrency model using effect handlers (Hillerström, 2016). In addition to keeping the compiler lean, lifting the concurrency implementation into user-space allows us to potentially experiment with different implementations or even have several different implementations coexisting.

1.1 Motivation

Mainstream managed programming languages tend to be closely tied to a complex, monolithic runtime system. Examples are the Java family of programming languages, that run on top of Oracle's Java HotSpot VM (HotSpotVM, 2016); the .NET family of programming languages, that run on top of Microsoft's Common Language Runtime (Microsoft Corp., 2016); the Haskell programming language with the Glasgow Haskell Compiler Runtime System (GHC, 2014).

Typically runtime systems are responsible for managing concurrency amongst other things such as garbage-collection, and any language features which do not compile to code themselves. Often concurrency support is hard-wired deeply into the runtime, requiring special support from the compiler to expose concurrency primitives as libraries for the programmer (Sivaramakrishnan et al., 2016). As a consequence it is difficult to change or evolve the concurrency implementation.

If we can lift concurrency into the programming language, allowing programmers to compose their own domain-specific concurrency abstractions, then we not only simplify the compiler and runtime system, we also simplify the life of the compiler writer.

1.2 Problem statement

It poises an interesting research question whether we can implement a compiler for Links without concurrency support, and then reconstruct the message-passing concurrency model of Links using effect handlers. Thus, in this dissertation we attempt to answer the following problem:

How can we compile the handlers of Links to native code, and moreover, how can we reconstruct a faithful implementation of the message-passing concurrency model of Links with handlers?

By faithful we mean an implementation that is close to the built-in implementation of the Links interpreter as we inevitably have to give up special syntax once we reify concurrency as user-defined library.

1.2.1 Approach and proposed solution

Recently, the Multicore OCaml project (Dolan et al., 2015) has extended the industrial-strength functional programming language OCaml with so-called *linear effect handlers* as an effort to bring multicore capabilities to the language. The purpose of adding handlers is to enable programmers to express user-defined concurrent multi-threaded schedulers.

Implementing a native backend for a programming language such as Links is a non-trivial task. In order to make it viable task we plan to take advantage of Multicore OCaml backend that already provides native support for effect handlers, and that Links is written in OCaml. Thus we intend to integrate the OCaml backend into Links infrastructure, and translate the Links intermediate representation into the intermediate representation of OCaml. However, the task is somewhat complicated by the fact that the effect handlers of Links are so-called *multi-shot effect handlers*, which are more expressive than linear effect handlers. Thus, we will consider how to encode multi-shot handlers in OCaml backend.

Our work differs further from theirs as we attempt to implement process-oriented rather shared-memory concurrency. Furthermore, we consider how to encode an entire message-passing concurrency model.

1.3 Scope, aims, and objectives

Since Links is full-fledged programming language for the web, we will restrict ourselves to work only with a subset of Links, namely, the backend of Links which supports effect handlers. In other words, we aim to extend the Links infrastructure with a native backend for effect handlers. We will deliberately leave support for the web-related features of Links for future work.

Our main contributions are

- A compiler for Links, that support native compilation of multi-shot handlers (Hillerström et al., 2016). We describe a translation from the Links IR to the OCaml IR which demonstrates how to encode multi-shot handlers in the OCaml IR.
- A reconstruction of the message-passing concurrency model of Links using effect handlers (Hillerström, 2016), that maintains the type-safe communication property of the original built-in implementation.
- A formalisation of the implementation of effect handlers in Links (Hillerström and Lindley, 2016).

1.4 Thesis outline

In Chapter 2 we give a brief introduction to programming Links, effect handlers, and the concurrency model of Links. We also discuss related work.

We discuss the compiler infrastructure and the compilation strategy in Chapter 3 as well as a core calculus, $\lambda^{\rho}_{\text{eff}}$, that captures the essence of the Links IR. Furthermore, we describe a translation from the Links IR to the OCaml IR.

In Chapter 4 we demonstrate a reconstruction of the concurrency model of Links using effect handlers.

We perform some experiments with our compiler in order to measure its performance against the Links interpreter and the OCaml compiler in Chapter 5. We analyse and discuss the results of the experiments. In addition, we consider how one may improve the performance of our compiler.

Finally in Chapter 6 we conclude and discuss future work.

Chapter 2

Background and related work

In this chapter we give an introduction to programming with algebraic effects and handlers by example in Links. Although, first we introduce Links without effect handlers in Section 2.1. Then in Section 2.2 we introduce effect handlers. In Section 2.4 we discuss the built-in implementation of the concurrency model of Links. Finally, in Section 2.5 we discuss related work.

2.1 Links primer

In section provides a brief primer to the Links programming language. As an introductory example we will implement a purely functional first-in-last-out queue in the style of Okasaki (1998). The example comprises most of the Links features that we will use in Chapter 4.

We begin by defining a type constructor `Queue(a)` which classifies queues whose elements have type `a`:

```
typename Queue(a::Type) = ([a],[a]);
```

The keyword `typename` is used to define type aliases. The notation `a::Type` denotes that the type variable `a` has kind `Type`. Links has several kinds, however, we will only use the `Type` and `Row` kinds. The type constructor `Queue(a)` constructs a pair of lists whose elements have type `a`. The main idea is that the two lists, respectively, represent the front and back of the queue. We always insert elements into the back list, and remove elements from the front list.

We define a useful function which creates an empty queue:

```
sig emptyQueue : () -> Queue(a)
fun emptyQueue() { ([], []) }
```

The syntax of Links is loosely based on that of JavaScript. The `fun` keyword begins a function definition (like `function` in JavaScript). Just as in JavaScript functions are n-ary, but they can also be curried. Unlike in JavaScript, functions are statically typed and the `sig` keyword begins a type signature. The type signature reads: `emptyQueue` is a nullary function that returns a type polymorphic queue. The empty queue is represented as pair of empty lists. The notation `[]` denotes the empty list.

Next we define another useful function which puts an element into a queue:

```
sig enqueue : (Queue(a), a) -> Queue(a)
fun enqueue((xs,ys), y) { (xs, y::ys) }
```

Here we *pattern match* on the first argument of `enqueue` to bind the front list to `xs` and the back list to `ys`. We cons the element `y` onto the back list by using the list cons operator `::`.

Now that we can populate a queue we will consider how to depopulate a queue. We will define a function `dequeue` that removes the head of a given queue. The definition of `dequeue` is a bit more involved than the definition of `enqueue`, because we have to deal with the special case when the queue is empty. To deal with this case we use a standard functional programming pattern: represent the possibility of failure as a `Maybe`-type. We can define the type as a *variant*:

```
typename Maybe(a) = [|Nothing
                     |Just:a
                     |];
```

The syntax `[|...|]` denotes a variant type in Links in which components of the variant type are delimited by the pipe symbol (`|`). The `Nothing` and `Just` are data constructors. The idea is to tag a dequeued element by `Just` to inform the caller that an element was successfully removed. In case there are no elements in the queue we simply return `Nothing`. The function `dequeue` is implemented as follows:

```
sig dequeue : (Queue(a)) ~> (Maybe(a), Queue(a))
fun dequeue(q) {
  switch (q) {
    case ([], [])       -> (Nothing, q)
    case (x :: xs, ys) -> (Just(x), (xs, ys))
    case ([], ys)       -> dequeue((reverse(ys), []))
  }
}
```

The `switch(q){...}` construct pattern matches on the shape of the queue `q` through a list of clauses. The first clause considers the special case when the queue is empty. In this case we simply return `Nothing` and the unmodified queue. The

second clause considers the case when the front list is non-empty. In this case remove the head of the front list and return it inside a `Just` along with the modified queue. The final clause considers the case when the front list is empty but the back list is (possibly) non-empty. In this case we swap the front and back list. In addition, the back list gets reversed to preserve the first-in-last-out semantics of the queue.

Effect system Links has a type-and-effect system which tracks the effects that functions may perform. For example, the reader may have noticed that the arrow in the type signature of `dequeue` is squiggly (`~>`) rather than straight (`->`). The squiggly function arrow is syntactic sugar for denoting that the computation has the *wild* effect. The wild effect captures intrinsic effects such as I/O, randomness, divergence, etc. To some extent it is analogous to the *IO monad* of Haskell, though the wild effect is much stricter as without it general recursion is disallowed. The effect system of Links uses row polymorphism to provide extensible effect signature. We can elaborate the squiggly arrow `~>` to `{wild|e}->` where `{wild|e}` denotes a row with a label `wild` and an effect variable `e`. The effect variable is a row variable which can be instantiated to populate the row with additional labels (Hillerström and Lindley, 2016). The straight arrow is actually syntactic sugar for writing `{ |e}->`, that is an empty open effect row which means the function can be used in the presence of any other effect.

Type inference Often in Links we may omit type signatures altogether as Links has type inference. If we write `fun just(x) { Just(x) }` then we might expect the inferred type to be `just : (a) -> Maybe(a)`. However, Links will infer the type `just : (a) -> [|Just:a|e|]`, where `e` is a row variable that can be instantiated to contain additional labels. The reason for this behaviour is that Links employs *structural typing* as opposed to *nominal typing*. In the latter typing discipline any two terms have the same type if and only if they are constructed by the same constructor. By contrast, in the structural typing discipline any two terms have *compatible* types if they have the same structure. For example, we can unify the given type and the inferred type above

$$\texttt{Maybe(a)} \backsim \texttt{[|Just:a|e|]}$$

by instantiating the row variable `e` to `Nothing`.

2.2 Affine and multi-shot effect handlers

This section provides a short primer to effect handlers in Links. The contents of this section are largely based on Hillerström et al. (2016). An algebraic effect is given by a signature of *abstract operations*. For example *nondeterminism* is an algebraic effect that is given by a nondeterministic choice operation called `Choose`. In Links, we may use this operation to implement a coin toss:

```
sig toss : Comp({Choose:Bool |e}, Toss)
fun toss() { if (do Choose) Heads else Tails }
```

This declares an *abstract computation* `toss`, which invokes an operation `Choose` using the `do` primitive. The `sig` keyword begins a signature, which reads: `toss` is a computation with effect signature `{Choose:Bool |e}` and return value `Toss`, whose constructors are `Heads` and `Tails`. The effect signature conveys that the computation may perform the `Choose` operation. In particular, the effect row has an effect variable `e` which means the computation may be invoked in the scope of additional effects. The computation type `Comp(·,·)` is not a built-in type. It is straightforward to define in Links:

```
typename Comp(e::Row,a::Type) = () -e-> a;
```

The type constructor takes a row type `e` and a regular type `a` and constructs a thunk with effect row `e` and return type `a`.

We need a suitable effect handler in order to evaluate the computation `toss`. An effect handler instantiates a subset of the operations of an abstract computation. For example, the following handler interprets `Choose` randomly:

```
sig randomResult : (Comp({Choose:Bool |e}, a)) ->
                    Comp({Choose{_}   |e}, a)
handler randomResult {
  case Return(x)       -> x
  case Choose(resume) -> resume(random() > 0.5)
}
```

The signature conveys that the handler interprets the operation `Choose` and leaves any other operations uninterpreted. The notation `Choose{_}` denotes that the operation is polymorphic in its presence. The handler comprises two clauses:

1. the `Return`-clause specifies how to handle the return value of the computation.

2. the `Choose`-clause specifies how to handle a `Choose` operation. The parameter `resume` is the delimited continuation of the operation `Choose` in the computation.

We say that `randomResult` is a *linear handler*, because it invokes every continuation exactly once. Essentially, the handler interprets the computation `toss` as modelling a *fair* coin. Using this handler we can evaluate the computation `toss` in the Links interpreter:

```
links> randomResult(toss)();
Tails : Toss
links> randomResult(toss)();
Heads : Toss
```

Evaluating the composition produces either `Heads` or `Tails` uniformly.

Alternatively, we may define a handler for `Choose` that invokes its continuation twice to enumerate every possible outcome:

```
sig allResults : (Comp({Choose:Bool |e}, a)) ->
                  Comp({Choose{_}   |e}, [a])
handler allResults {
  case Return(x)      -> [x]
  case Choose(resume) -> resume(true) ++ resume(false)
}
```

Observe that the return value is lifted into a singleton list. The `Choose`-clause concatenates the outcomes obtained by interpreting the operation as `true` and `false`, respectively. We say that `allResults` is a *multi-shot handler*. The composition `allResults(toss)` produces a computation that yields a list of the possible outcomes, i.e.

```
links> allResults(toss)();
[Heads, Tails] : [Toss]
```

Using the `toss` computation we can model a *drunk coin toss*. In a drunken coin toss the drunkard may fail to catch the coin after flipping it. We need an additional operation `Fail : Zero` that models failure, then we can implement the drunk coin toss as follows:

```
sig drunkToss : Comp({Choose:Bool,Fail:Zero |e}, Toss)
fun drunkToss() { if (do Choose) toss()
                  else switch (do Fail) { } }
```

Here `Zero` is the empty type, and thus the `switch` pattern matching construct has no clauses.

Note that the additional operation causes the effect row to grow accordingly. Now it comprises two operations. Thus, `randomResult` is no longer sufficient in order to give a full interpretation of `drunkToss`. We need yet another handler that interprets `Fail`.

An interpretation of `Fail` amounts to defining a familiar *exception handler*.

As an example consider the following exception handler, which returns `Just` the result of the computation or returns `Nothing` if the operation `Fail` is performed:

```
sig maybeResult : (Comp({Fail:Zero |e},       a)) ->
                   Comp({Fail{_}   |e}, Maybe(a))
handler maybeResult {
  case Return(x) -> Just(x)
  case Fail(_)   -> Nothing
}
```

The type system prevents invocation of the continuation in the `Fail`-clause, because the type `Zero` has zero inhabitants. Linear and exception handlers together constitute *affine handlers*.

We can compose `maybeResult` and `randomResult` to give an interpretation of `drunkToss`. Figure 2.1 depicts a sequence diagram for evaluation of this composition. The result of the evaluation is nondeterministic:

```
links> maybeResult(randomResult(drunkToss))();
Just(Tails) : Maybe(Toss)
links> maybeResult(randomResult(drunkToss))();
Nothing : Maybe(Toss)
```

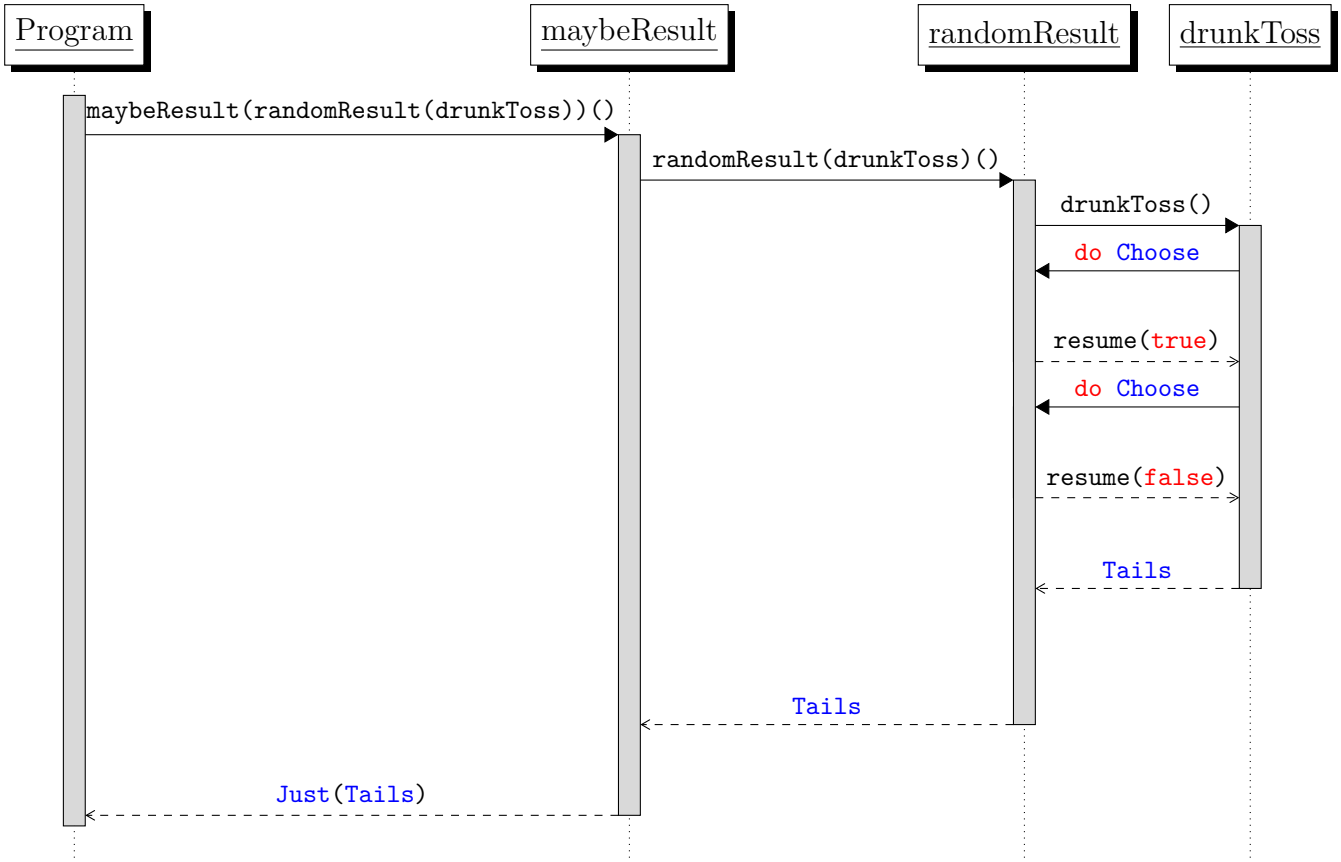

Figure 2.1: Sequence diagram for a `maybeResult(randomResult(drunkToss))`.

2.2.1 Handler pipelines

A composition of handlers effectively form a pipeline where unhandled operations flow from inside out. To capture this intuition – and to reduce the number of parentheses – we define an infix binary operator that constructs a pipeline:

```
op h -<- g { fun(m) { h(g(m)) } }
```

The `op` keyword begins the definition of an infix binary operator in Links. The operator reads: `h` after `g`. In addition, we give another operator to plug a computation into a handler pipeline:

```
op h -< m { h(m) }
```

This operator simply applies the handler `h` to the computation `m`. Furthermore, it is convenient to have a top-level function that *runs* a handler pipeline:

```
fun run(m) { m() }
```

The function forces the thunk produced by a handler pipeline. Now, we can write the composition of `maybeResult` and `randomResult` as

```
links> run -<- maybeResult -<- randomResult -< drunkToss;
Just(Tails) : Maybe(Toss)
```

For deep pipeline of handlers these two operators help make handler composition syntactically lightweight.

2.3 State and parameterised handlers

Most programs maintain some sort of state through their life time. It is possible to describe state as an effect with operations for reading (`Get : s`) and updating (`Put : s {}-> ()`) a state of type `s`. We implement them as follows:

```
sig get : () {Get:s|_}-> s
fun get() {do Get}

sig put : (s) {Put:(s) {}-> ()|_}-> ()
fun put(s) {do Put(s)}
```

Typically, we wrap the invocation of an abstract operation inside a function. This is mainly because it lets us compose effects with functions seamlessly. Moreover, sometimes we want to do more than just invoking an operation.

We use a parameterised handler to give an interpretation of state. In addition to supplying a computation to a parameterised handler, we also supply one or more parameters. In this instance we pass the state as an additional parameter `s`

```
sig evalState : (s) ->
                (Comp({Get:s ,Put:(s) {}-> () |e}, a)) ->
                 Comp({Get{_},Put{_}           |e}, a)
handler evalState(s) {
  case Get(k)    -> k(s)(s)
  case Put(s,k)  -> k(())(s)
  case Return(x) -> x
}
```

The main difference compared to an unparameterised handler is that the continuation `k` is a curried function that takes a return value followed by the handler parameters. In the `Get` clause, we return the state and also pass it unmodified to any subsequent invocations of the handler. Similarly, in the `Put` clause, we return unit and update the state.

In order to demonstrate the state handler in action consider the following example:

```
fun fortytwo() {
  var q = enqueue(get(), 42);
  put(q);
  var (x, _) = dequeue(get());
  switch (x) {
    case Just(i) -> print(intToString(i))
    case Nothing -> print("No elements.")
  }
}
```

The program retrieves a queue using the `get` operation, enqueues the element `42`, and stores the modified queue. Afterwards, the queue is retrieved again and an element is dequeued. We use the `evalState` handler to interpret the computation:

```
links> run -<- evalState(emptyQueue()) -< fortytwo;
42 : Int
```

We seed the state handler with the empty queue which is the initial state of the program.

2.4 The built-in concurrency model of Links

The concurrency model of Links is based on a typed *actor* model (Cooper et al., 2006). In an actor model processes run in (memory) isolation (Hewitt and Baker, 1977). A process can only make state changing decisions locally. In order to influence the global program state, the process must communicate with other processes through message passing. Each process is equipped with its own mailbox. Figure 2.2 sketches an abstract representation of the interaction between two processes.

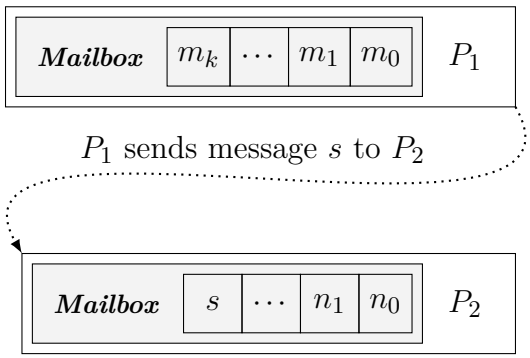

Figure 2.2: Interaction between processes.

A Links program begins in a single process of control, but this process can spawn multiple subprocesses, which in turn can spawn subprocesses of their own. multiple processes. There are four essential built-in primitives for concurrency: process *self referral*, process *spawning*, *sending* and *receiving* messages. In the following paragraphs we briefly introduce each primitive.

Self referral The built-in function `self` retrieves the process identifier from the current context. A process identifier has the type `Process({ |e })`. The type is parameterised by an effect row with an effect variable `e` which tracks the effects that the process may perform. The signature of `self` is

```
links> self;
self : () ~e~> Process ({ |e })
```

Invoking the function at the top level retrieves the identifier of the main process:

```
links> self();
0 : Process ({ |_ })
```

Evidently the main process always has identifier `0`. Subsequent processes are assigned identifiers `1`, `2`, `3`, and so forth. Although, the term `0` looks like a value of the integer type we cannot act upon it as such, because the process type is implemented as an abstract type. Therefore the type checker will prevent us from incrementing the process identifer by hand:

```
links> self() + 1;
<stdin>:1: Type error: [..]
```

Here, we have omitted the full error message for brevity, however the problem is that the addition operator `(+)` expects two arguments of type `Int`. The two types `Process({ |e })` and `Int` are incompatible. This adds a layer of safety to the concurrency model as we cannot erroneously refer to an non-existent process.

Spawning The primitive `spawn { expression }` returns a handle to a new process which begins by evaluating `expression`. For example by spawning the computation `print("Hello World")` we obtain the process identifier for the subprocess:

```
links> spawn { print("Hello World") };
Hello World
1 : Process ({ wild|_ })
```

We obtain the handle `1` whose effect row contains the `wild` effect due to the fact that the subprocess prints to the standard out.

Receiving and sending A process can receive messages using the `recv` function, e.g.

```
links> var p2 = spawn { print("Message: " ^^ recv()) };
p2 = 2 : Process ({ hear:String,wild|_ })
```

The `recv` function blocks until a message becomes available. Each mailbox is given a static type according to the messages it expects to receive. The built-in effect `hear` reflects and tracks this type. We can send a message to process `1` using the `!` primitive (pronounced "send"):

```
links> p2 ! "Hello";
Message: Hello
() : ()
```

Process `1` prints the received message `"Hello"` and terminates afterwards. The process is only capable of receiving strings but often a process will use a variant to tag the different messages it can receive. Typically, a process will dispatch on the tag of received message. As a concrete example consider this example adapted from the Links documentation (Links, 2016) where a process that gets informed about passing comets and celebrity sightings:

```
var p3 = spawn {
  fun loop() {
    var _ = switch(recv()) {
      case PassingComet(id, zenith, azimuth) -> cometSighted(
   id, zenith, azimuth)
      case CelebritySighting(name, venue)    -> celebSighted(
   name, venue)
    };
    loop()
  }
  loop()
};
```

Here we assume the existence of two functions `cometSighted` and `celebSighted` that register sightings of comets and sightings of celebrities, respectively. The

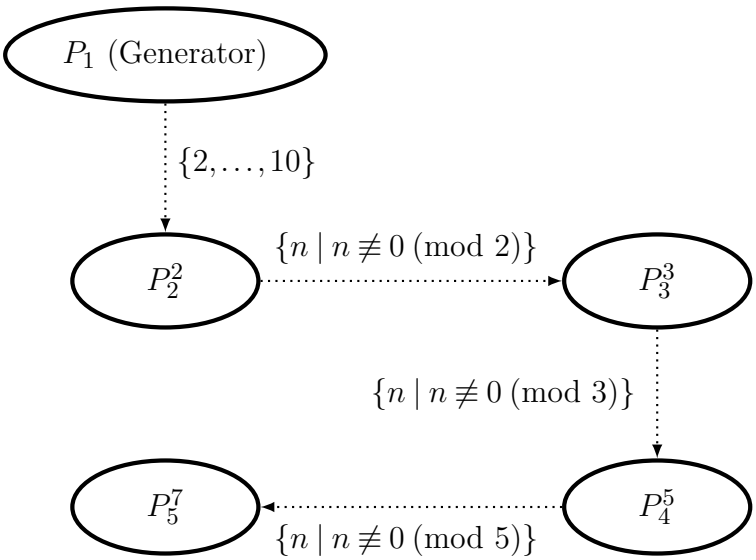

Figure 2.3: Visual representation of Sieve of Eratosthenes.

`hear` effect in the type signature of `p3` now reflects that the process expects to receive messages tagged by either `CelebritySighting` or `PassingComet`:

```
links> p3;
3 : Process ({ hear:[|CelebritySighting:(String, String)
                     |PassingComet:(Int, Float, Float)|]
             , wild|_ })
```

We can inform the process of any passing comets and celebrity sightings:

```
links> p3 ! CelebritySighting("Ewan McGregor", "Leith");
Ewan McGregor has been seen in Leith
() : ()
links> p3 ! PassingComet(42, 10.3, 180.5);
Comet no. 42 sighted (10.3, 180.5)
() : ()
```

2.4.1 Sieve of Eratosthenes example

We will now consider a larger example which will serve to demonstrate an actual concurrent application in Links. However, we shall reuse the example to demonstrate our reconstruction of the concurrency model in Chapter 4.

We shall implement a parallel version of the *Sieve of Eratosthenes* prime number finding algorithm.

The basic idea is to dynamically construct a pipeline of processes where each process holds one prime number (Andrews, 2000). Figure 2.3 visualises the sieve pipeline which finds primes between 2 and 10. In the figure the subscript of each process is its identifier and the superscript is the prime number it holds. The

job of each sieve process is to receive and perform a primality test on candidate prime by dividing the candidate number by its own prime. If the remainder after division is positive then the process forwards the candidate to its neighbour process. The initial process generates a sequence of natural numbers. These numbers are sent one by one to the first sieve process. We implement this as the function `generator(n)` where `n` is the upper bound of the sequence:

```
fun generator(n) {
  var first = spawn { sieve() };
  foreach([2..n], fun(p) { first ! Candidate(p) });
  first ! Stop
}
```

The function spawns the first sieve as a child process. The `foreach` function has type `([a], (a) ~e~> ()) ~e~> ()`, that is it takes a list and an action as arguments. The action is applied to each element in the list. The notation `[2..n]` is a shorthand for generating the sequence of integers between 2 and n. The action function sends a `Candidate`-tagged number to the first sieve process. When the entire sequence has been transmitted the generator sends the `Stop` signal.

The first message sent to a sieve process will always be its prime. Subsequent messages may either be a `Candidate` prime number or the `Stop` signal. The implementation of `sieve` is given below.

```
1  fun sieve() {
2    var myprime = fromCandidate(recv());
3    print(intToString(myprime));
4    fun loop(neighbour) {
5      switch (recv()) {
6        case Stop -> stop(neighbour)
7        case Candidate(number) ->
8        if (number `mod` myprime == 0) {
9          loop(neighbour)
10       } else {
11         var neighbour =
12           switch (neighbour) {
13             case Just(pid) -> pid
14             case Nothing   -> spawn { sieve() }
15           };
16         neighbour ! Candidate(number);
17         loop(Just(neighbour))
18       }
19     }
20   }
21   loop(Nothing)
22 }
```

We describe function line by line.

Line 2 receives the process' prime number. The `fromCandidate` simply removes

the `Candidate` tag from the number.

Line 3 simply prints the prime to standard out.

Line 4 begins the definition of the main loop of the process. The function is parameterised by a handle to its neighbouring process.

Lines 5-9 dispatch on the message tag. If the message is `Stop` then the auxiliary function `stop` (described below) propagates the stop signal to the neighbour process. If a candidate prime number is received then the process performs a primality test on the candidate number. In case the number is composite the `loop` function recurse in order to repeat the procedure.

Lines 11-17 forwards the candidate `number` to its neighbour. However before doing so the process must ensure it has a neighbour. If the process already has a neighbour then the `switch` expression simply removes the `Just` tag from the neighbour's identifier. In case it does not have a neighbour it spawns one and returns the new neighbour's identifier. Thereafter the process rewraps the candidate number and sends it to its neighbour. Finally, the `loop` function gets called recursively with the neighbour's identifier wrapped in a `Just`.

The `stop` function handles the special case of when the process has no neighbour:

```
fun stop(neighbour) {
  switch (neighbour) {
    case Nothing   -> ()
    case Just(pid) -> pid ! Stop
  }
}
```

The process silently exits if it does not have a neighbour. Otherwise the process forwards the `Stop` message before exiting. Now, we can run the example:

```
links> generator(10);
2
3
5
7
() : ()
```

As expected the primes between 2 and 10 get printed.

2.5 Related work

Since the inception of effect handlers a rather lot of implementations have appeared. Many of these implementations are attempts at encoding handlers in an existing language. Nevertheless, some new languages have been designed from the ground up with handlers in mind.

2.5.1 Implementations of effect handlers

Any signature of abstract operations can be understood as a free algebra and represented as a functor. In particular, every such functor gives rise to a free monad. Thus, free monads provide a natural basis for implementing effect handlers (Swierstra (2008) provide an account of free monads for functional programmers). Many of the library implementations of effect handlers include implementations based on free monads (Kammar et al., 2013; Kiselyov et al., 2013; Kiselyov and Ishii, 2015; Brady, 2013; Wu et al., 2014).

Kammar et al. (2013) take advantage of Haskell's aggressive fusion optimisations for an efficient Haskell library for handlers, as explained in detail by Wu and Schrijvers (2015). Saleh and Schrijvers (2016) apply a similar technique for optimising their embedding of handlers in ProLog. Kiselyov and Ishii (2015) also optimise a different Haskell library for handlers, taking advantage of prior work on optimising monadic reflection (van der Ploeg and Kiselyov, 2014).

The Idris effects library by Brady (2013) takes advantage of dependent types to provide effect handlers for a form of effects corresponding to parameterised monads (Atkey, 2009). Our work differs from these systems in that we compile effect handlers directly, rather than via library.

We are aware of three languages that are specifically designed with effect handlers in mind.

- The Eff language by Bauer and Pretnar (2015) is a strict language with Hindley-Milner type inference similar in spirit to ML, but extended with effect handlers. It has the look-and-feel of the OCaml programming language. It includes a novel feature for supporting fresh generation of effects in order to support effects such as ML-style higher-order state. Currently, the Eff is compiled to a free monad encoding in the surface syntax of OCaml.

- Frank by Lindley et al. (2016) takes the idea of effect handlers to the ex-

treme, having no primitive notion of function, only handlers. In Frank a function is but a special case of a handler. Interestingly, Frank includes the notion of *multi-handlers* which handle multiple computations at once. Frank is built on a bidirectional type system. It includes an effect type system and a novel form of effect polymorphism in which the programmer never needs to read or write any effect variables. Currently, Frank has only a prototypical interpreter.

- Shonky by McBride (2016) amounts to a dynamically-typed variant of Frank. Though it is not statically typed, handlers must be annotated with the names of the effects that they handle. The implementation of Shonky uses a generalisation of the CEK machine akin to the one used by the Links interpreter. The main differences are that Shonky uses a different IR than Links.

Although OCaml itself has no support for effect handlers, a development branch, Multicore OCaml (Dolan et al., 2015), does. Multicore OCaml does not include an effect type system, and handlers are restricted so that continuations are affine, that is, they can be invoked at most once. This design admits a particularly efficient implementation (Bruggeman et al., 1996), as continuations need never be copied, so they can simply be stored on the stack.

The programming language Koka by Leijen (2016) has recently been extended with effect handlers. The Koka compiler employs a particularly efficient compilation scheme using a type directed selective continuation-passing style (CPS) translation in order to compile effect handlers to common runtime platforms.

Chapter 3

Compiling effect handlers

Links is a strict ML-like functional language for the web (Cooper et al., 2006). The defining feature of Links is that it provides a single source language that targets all three tiers of a web application: client, server, and database. Links source code is translated into an intermediate representation (IR) based on A-normal form (Flanagan et al., 1993).

In this chapter we describe our native compiler backend for server-side Links. In addition, we give a formalisation of the intermediate language used within the Links compiler. Furthermore, we describe the translation of the intermediate representation of Links into the intermediate representation of OCaml. The material in Section 3.1 and Section 3.5 is based on Hillerström et al. (2016), while the contents of Section 3.2 are based on Hillerström and Lindley (2016).

3.1 Compilation strategy

We reuse most of the previous Links infrastructure. Though, we extend the compiler infrastructure with a native backend as shown in Figure 3.1. The Links compiler pipeline follows a rather conventional compiler pipeline design. The frontend comprises a parser, an early elaboration pass that happens before type checking. The backend contains a translation pass of the Links frontend into a small, typed intermediate language in *A-normal form* (ANF) (Flanagan et al., 1993). This pass is assisted by a pattern matching compiler. The Links interpreter implements a generalised CEK machine (Hillerström and Lindley, 2016), which interprets ANF code directly.

For the native backend our compilation strategy is to translate the Links ANF

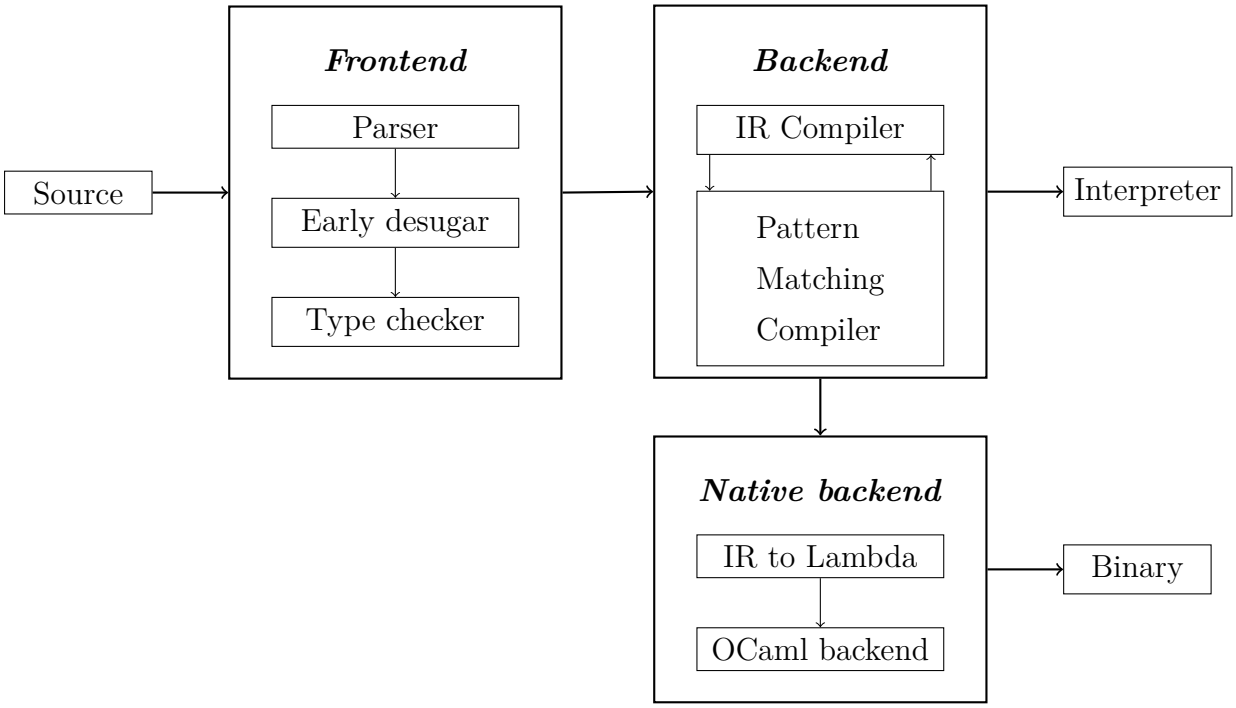

Figure 3.1: Links compiler pipeline.

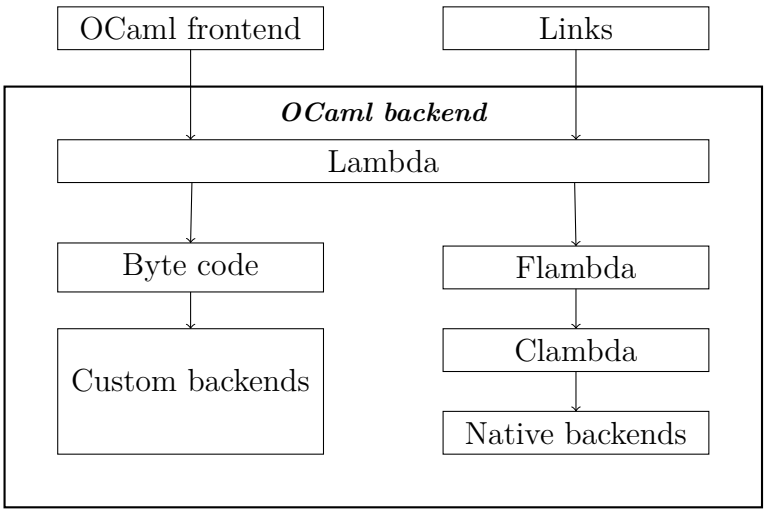

Figure 3.2: OCaml backend (Hillerström et al., 2016).

language into the OCaml *Lambda* language, which is a small, untyped lambda calculus. The OCaml backend exposes a hierarchy of intermediate representations, where the top representation is known as Lambda. As shown in the Figure 3.2, the Lambda IR offers two different compilation options: byte code and native code. Therefore by targeting Lambda rather than a lower level IR, we achieve maximum flexibility as a translation into byte code, in principle, enables us to take advantage of custom backends.

There are several semantic differences between Links and OCaml, e.g. Links employs structural typing, whilst OCaml predominantly employs nominal typing. In particular, Links employs row typing for effects, records, and variants, whereas OCaml only supports row typing for the latter. Exhibiting a faithful translation from Links to OCaml amounts to a lot of value boxing (Holmes, 2009). Thus, we target Lambda for greater flexibility and control. We effectively subvert OCaml's typechecker by targeting Lambda, however the translation is safe as Links programs are already typechecked.

3.2 A formalisation of the Links intermediate language

In this section, we present a type and effect system and a small-step operational semantics for $\lambda^{\rho}_{\text{eff}}$ (pronounced "lambda-eff-row"), a Church-style row-polymorphic call-by-value calculus for effect handlers. This core calculus captures the essence of the Links IR. We prove that the operational semantics is sound with respect to the type and effect system.

The design of $\lambda^{\rho}_{\text{eff}}$ is inspired by the λ-calculi of Kammar et al. (2013), Pretnar (2015), and Lindley and Cheney (2012). As in the work of Kammar et al. (2013), each handler can have its own effect signature. As in the work of Pretnar (2015), the underlying formalism is fine-grain call-by-value (Levy et al., 2003), which names each intermediate computation like in A-normal form (Flanagan et al., 1993), but unlike A-normal form is closed under β-reduction. As in the work of Lindley and Cheney (2012), the effect system is based on row polymorphism.

$$
\begin{array}{lrl}
\text{Value types} & A, B & ::= A \to C \mid \forall \alpha^K.C \\
& & \quad \mid \langle R \rangle \mid [R] \mid \alpha \\
\text{Computation types} & C, D & ::= A!E \\
\text{Effect types} & E & ::= \{R\} \\
\text{Row types} & R & ::= \ell : P; R \mid \rho \mid \cdot \\
\text{Presence types} & P & ::= \mathsf{Pre}(A) \mid \mathsf{Abs} \mid \theta \\
\text{Handler types} & F & ::= C \Rightarrow D \\
\text{Types} & T & ::= A \mid C \mid E \mid R \mid P \mid F \\
\text{Kinds} & K & ::= \mathsf{Type} \mid \mathsf{Row}_{\mathcal{L}} \mid \mathsf{Presence} \\
& & \quad \mid \mathsf{Comp} \mid \mathsf{Effect} \mid \mathsf{Handler} \\
\text{Label sets} & \mathcal{L} & ::= \emptyset \mid \{\ell\} \uplus \mathcal{L} \\
\text{Type environments} & \Gamma & ::= \cdot \mid \Gamma, x : A \\
\text{Kind environments} & \Delta & ::= \cdot \mid \Delta, \alpha : K
\end{array}
$$

Figure 3.3: Types, effects, kinds, and environments

3.2.1 Types

The grammars of types, kinds, label sets, and type and kind environments are given in Figure 3.3.

Value types The function type $A \to C$ represents functions that map values of type A to computations of type C. The polymorphic type $\forall \alpha^K.C$ is parameterised by a type variable α of kind K. The record type $\langle R \rangle$ represents records with fields constrained by row R. Dually, the variant type $[R]$ represents tagged sums constrained by row R. The handler type $C \Rightarrow D$ represents handlers that transform computations of type C into computations of type D.

Computation types A computation type $A!E$ is given by a value type A and an effect E, which specifies the operations that the computation may perform.

Row Types Effect types, records and variants are defined in terms of rows. A row type embodies a collection of distinct labels, each of which is annotated with a presence type. A presence type indicates whether a label is *present* with some type A ($\mathsf{Pre}(A)$), *absent* (Abs) or *polymorphic* in its presence (θ).

Row types are either *closed* or *open*. A closed row type ends in $\cdot$, whilst an

open row type ends with a *row variable* ρ. Furthermore, a closed row term can have only the labels explicitly mentioned in its type. Conversely, the row variable in an open row can be instantiated with additional labels. We identify rows up to reordering of labels. For instance, we consider the following two rows equivalent:

$$\ell_1 : P_1; \cdots ; \ell_n : P_n \equiv \ell_n : P_n; \cdots ; \ell_1 : P_1.$$

The unit and empty type are definable in terms of row types. We define the unit type as the empty, closed record, that is, $\langle\cdot\rangle$. Similarly, we define the empty type as the empty, closed variant $[\cdot]$. Usually, we usually omit the $\cdot$ for closed rows.

Handler types A handler type $C \Rightarrow D$ is given by an input computation type C and an output computation type D.

Kinds We have six kinds: Type, Comp, Effect, $\mathsf{Row}_{\mathcal{L}}$, $\mathsf{Presence}$, $\mathsf{Handler}$, which classify value types, computation types, effect types, row types, presence types, and handler types, respectively. Row kinds are annotated with a set of labels $\mathcal{L}$. The kind of a complete row is $\mathit{Row}_{\emptyset}$. More generally, the kind $\mathit{Row}_{\mathcal{L}}$ denotes a partial row that cannot mention the labels in $\mathcal{L}$.

Type variables We let α, ρ and θ range over type variables. By convention we use α for value type variables or for type variables of unspecified kind, ρ for type variables of row kind, and θ for type variables of presence kind.

Type and kind environments Type environments map term variables to their types and kind environments map type variables to their kinds.

3.2.2 Terms

The terms are given in Figure 3.4. We let x, y, z, k range over term variables. By convention, we use k to denote continuation names.

The syntax partitions terms into values, computations and handlers. Value terms comprise variables (x), lambda abstraction ($\lambda x^A.\, M$), type abstraction ($\Lambda \alpha^K.\, M$), and the introduction forms for records and variants. Records are introduced using the empty record $\langle\rangle$ and record extension $\langle \ell = V; W \rangle$, whilst variants are introduced using injection $(\ell\, V)^R$, which injects a field with label ℓ

$$
\begin{array}{lll}
\text{Values} & V, W & ::= x \mid \lambda x^{A}.\, M \mid \Lambda \alpha^{K}.\, M \\
& & \mid \ \langle\rangle \mid \langle \ell = V; W \rangle \mid (\ell\, V)^{R} \\
\\
\text{Computations} & M, N & ::= V\, W \mid V\, A \\
& & \mid \ \mathbf{let}\ \langle \ell = x; y \rangle = V\ \mathbf{in}\ N \\
& & \mid \ \mathbf{case}\, V \{ \ell\, x \mapsto M; y \mapsto N \} \mid \mathbf{absurd}^{C} V \\
& & \mid \ \mathbf{return}\, V \\
& & \mid \ \mathbf{let}\ x \leftarrow M\ \mathbf{in}\ N \\
& & \mid \ (\mathbf{do}\, \ell\, V)^{E} \\
& & \mid \ \mathbf{handle}\, M\ \mathbf{with}\, H \\
\\
\text{Handlers} & H & ::= \{ \mathbf{return}\, x \mapsto M \} \\
& & \mid \ \{ \ell\, x\, k \mapsto M \} \uplus H
\end{array}
$$

Figure 3.4: Term syntax

and value V into a row whose type is R. We include the row type annotation in order to support bottom-up type reconstruction.

All elimination forms are computation terms. Abstraction and type abstraction are eliminated using application ($V\, W$) and type application ($V\, A$) respectively. The record eliminator ($\mathbf{let}\ \langle \ell = x; y \rangle = V\ \mathbf{in}\ N$) splits a record V into x, the value associated with ℓ, and y, the rest of the record. Non-empty variants are eliminated using the case construct ($\mathbf{case}\, V\, \{\ell\, x \mapsto M; y \mapsto N\}$), which evaluates the computation M if the tag of V matches ℓ. Otherwise it falls through to y and evaluates N. The elimination form for empty variants is ($\mathbf{absurd}^{C}\, V$). A trivial computation ($\mathbf{return}\, V$) returns value V. The expression ($\mathbf{let}\ x \leftarrow M\ \mathbf{in}\ N$) evaluates M and binds the result value to x in N.

The construct $(\mathbf{do}\, \ell\, V)^{E}$ invokes an operation ℓ with value argument V. The handle construct ($\mathbf{handle}\, M\ \mathbf{with}\, H$) runs a computation M with handler definition H. A handler definition H consists of a return clause $\mathbf{return}\, x \mapsto M$ and a possibly empty set of operation clauses $\{\ell_i\, x_i\, k_i \mapsto M_i\}_i$. The return clause defines how to handle the final return value of the handled computation, which is bound to x in M. The i-th operation clause binds the operation parameter to x_i and the continuation k_i in M_i.

We write $Id(M)$ for $\mathbf{handle}\, M\ \mathbf{with}\, \{\mathbf{return}\, x \mapsto x\}$. We write $H(\mathbf{return})$

for the return clause of H and $H(\ell)$ for the set of either zero or one operation clauses in H that handle the operation ℓ. We write $dom(H)$ for the set of operations handled by H. As our calculus is Church-style, we annotate various term forms with type or kind information (term abstraction, type abstraction, injection, operations, and empty cases); we sometimes omit these annotations.

3.2.3 Static semantics

The kinding rules are given in Figure 3.5 and the typing rules are given in Figure 3.6.

The kinding judgement $\Delta \vdash T : K$ states that type T has kind K in kind environment Δ. The value typing judgement $\Delta; \Gamma \vdash V : A$ states that value term V has type A under kind environment Δ and type environment Γ. The computation typing judgement $\Delta; \Gamma \vdash M : C$ states that term M has computation type C under kind environment Δ and type environment Γ. The handler typing judgement $\Delta; \Gamma \vdash H : C \Rightarrow D$ states that handler H has type $C \Rightarrow D$ under kind environment Δ and type environment Γ. In the typing judgements, we implicitly assume that Γ, A, C, and D, are well-kinded with respect to Δ. We define the functions $FTV(\Gamma)$ to be the set of free type variables in Γ.

The kinding and typing rules are mostly straightforward. The interesting typing rules are T-Handle and the two handler rules. The T-Handle rule states that **handle** M **with** H produces a computation of type D given that the computation M has type C, and that H is a handler that transforms a computation of type C into another computation of type D.

The T-Handler rule is crucial. The effect rows on the computation type C and the output computation type D must share the same suffix R. This means that the effect row of D must explicitly mention each of the operations ℓ_i, whether that be to say that an ℓ_i is present with a given type signature, absent, or polymorphic in its presence. The row R describes the operations that are forwarded. It may include a row-variable, in which case an arbitrary number of effects may be forwarded by the handler. The typing of the return clause is straightforward. In the typing of each operation clause, the continuation returns the output computation type D. Thus, we are here defining *deep* handlers (Kammar et al., 2013) in which the handler is implicitly wrapped around the continuation, such that any subsequent operations are handled uniformly by the same handler.

$$\textsc{TyVar}\ \frac{}{\Delta, \alpha : K \vdash \alpha : K} \qquad \textsc{Comp}\ \frac{\Delta \vdash A : \mathsf{Type} \qquad \Delta \vdash E : \mathsf{Effect}}{\Delta \vdash A ! E : \mathsf{Comp}}$$

$$\textsc{Fun}\ \frac{\Delta \vdash A : \mathsf{Type} \qquad \Delta \vdash C : \mathsf{Comp}}{\Delta \vdash A \to C : \mathsf{Type}} \qquad \textsc{Forall}\ \frac{\Delta, \alpha : K \vdash C : \mathsf{Comp}}{\Delta \vdash \forall \alpha^K . C : \mathsf{Type}} \qquad \textsc{Record}\ \frac{\Delta \vdash R : \mathsf{Row}_\emptyset}{\Delta \vdash \langle R \rangle : \mathsf{Type}}$$

$$\textsc{Variant}\ \frac{\Delta \vdash R : \mathsf{Row}_\emptyset}{\Delta \vdash [R] : \mathsf{Type}} \qquad \textsc{Effect}\ \frac{\Delta \vdash R : \mathsf{Row}_\emptyset}{\Delta \vdash \{R\} : \mathsf{Effect}} \qquad \textsc{Present}\ \frac{\Delta \vdash A : \mathsf{Type}}{\Delta \vdash \mathsf{Pre}(A) : \mathsf{Presence}}$$

$$\textsc{Absent}\ \frac{}{\Delta \vdash \mathsf{Abs} : \mathsf{Presence}} \qquad \textsc{EmptyRow}\ \frac{}{\Delta \vdash \cdot : \mathsf{Row}_{\mathcal{L}}} \qquad \textsc{ExtendRow}\ \frac{\Delta \vdash P : \mathsf{Presence} \qquad \Delta \vdash R : \mathsf{Row}_{\mathcal{L} \uplus \{\ell\}}}{\Delta \vdash \ell : P ; R : \mathsf{Row}_{\mathcal{L}}}$$

$$\textsc{Handler}\ \frac{\Delta \vdash C : \mathsf{Comp} \qquad \Delta \vdash D : \mathsf{Comp}}{\Delta \vdash C \Rightarrow D : \mathsf{Handler}}$$

Figure 3.5: Kinding rules (Hillerström and Lindley, 2016).

Values

$$\textsc{T-Var}\ \frac{x : A \in \Gamma}{\Delta;\Gamma \vdash x : A} \qquad \textsc{T-Lam}\ \frac{\Delta;\Gamma, x : A \vdash M : C}{\Delta;\Gamma \vdash \lambda x^{A}.M : A \to C} \qquad \textsc{T-PolyLam}\ \frac{\Delta, \alpha : K;\Gamma \vdash M : C \qquad \alpha \notin FTV(\Gamma)}{\Delta;\Gamma \vdash \Lambda\alpha^{K}.M : \forall\alpha^{K}.C}$$

$$\textsc{T-Unit}\ \frac{}{\Delta;\Gamma \vdash \langle\rangle : \langle\rangle} \qquad \textsc{T-Extend}\ \frac{\Delta;\Gamma \vdash V : A \qquad \Delta;\Gamma \vdash W : \langle \ell : \mathsf{Abs}; R\rangle}{\Delta;\Gamma \vdash \langle \ell = V; W\rangle : \langle \ell : \mathsf{Pre}(A); R\rangle}$$

$$\textsc{T-Inject}\ \frac{\Delta;\Gamma \vdash V : A}{\Delta;\Gamma \vdash (\ell V)^{R} : [\ell : \mathsf{Pre}(A); R]}$$

Computations

$$\textsc{T-App}\ \frac{\Delta;\Gamma \vdash V : A \to C \qquad \Delta;\Gamma \vdash W : B}{\Delta;\Gamma \vdash V\,W : C} \qquad \textsc{T-PolyApp}\ \frac{\Delta;\Gamma \vdash V : \forall\alpha^{K}.C \qquad \Delta \vdash A : K}{\Delta;\Gamma \vdash V\,A : C[A/\alpha]}$$

$$\textsc{T-Split}\ \frac{\Delta;\Gamma \vdash V : \langle \ell : \mathsf{Pre}(A); R\rangle \qquad \Delta;\Gamma, x : A, y : \langle \ell : \mathsf{Abs}; R\rangle \vdash N : C}{\Delta;\Gamma \vdash \mathbf{let}\ \langle \ell = x; y\rangle = V\ \mathbf{in}\ N : C}$$

$$\textsc{T-Case}\ \frac{\Delta;\Gamma \vdash V : [\ell : \mathsf{Pre}(A); R] \qquad \Delta;\Gamma, x : A \vdash M : C \qquad \Delta;\Gamma, y : [\ell : \mathsf{Abs}; R] \vdash N : C}{\Delta;\Gamma \vdash \mathbf{case}\ V\{\ell\, x \mapsto M; y \mapsto N\} : C}$$

$$\textsc{T-Absurd}\ \frac{\Delta;\Gamma \vdash V : []}{\Delta;\Gamma \vdash \mathbf{absurd}^{C}\ V : C} \qquad \textsc{T-Return}\ \frac{\Delta;\Gamma \vdash V : A}{\Delta;\Gamma \vdash \mathbf{return}\ V : A!E}$$

$$\textsc{T-Let}\ \frac{\Delta;\Gamma \vdash M : A!E \qquad \Delta;\Gamma, x : A \vdash N : C}{\Delta;\Gamma \vdash \mathbf{let}\ x \leftarrow M\ \mathbf{in}\ N : C} \qquad \textsc{T-Do}\ \frac{\Delta;\Gamma \vdash V : A \qquad E = \{\ell : A \to B; R\}}{\Delta;\Gamma \vdash (\mathbf{do}\ \ell\ V)^{E} : B!E}$$

$$\textsc{T-Handle}\ \frac{\Delta;\Gamma \vdash M : C \qquad \Delta;\Gamma \vdash H : C \Rightarrow D}{\Delta;\Gamma \vdash \mathbf{handle}\ M\ \mathbf{with}\ H : D}$$

Handlers

$$\textsc{T-Handler}\ \frac{\begin{array}{c} C = A!\{(\ell_i : A_i \to B_i)_i; R\} \\ D = B!\{(\ell_i : P_i)_i; R\} \qquad H = \{\mathbf{return}\ x \mapsto M\} \uplus \{\ell_i\ y\ k \mapsto N_i\}_i \\ [\Delta;\Gamma, y : A_i, k : B_i \to D \vdash N_i : D]_i \qquad \Delta;\Gamma, x : A \vdash M : D \end{array}}{\Delta;\Gamma \vdash H : C \Rightarrow D}$$

Figure 3.6: Typing rules (Hillerström and Lindley, 2016).

S-App
$(\lambda x^A.\, M)V \rightsquigarrow M[V/x]$

S-TyApp
$(\Lambda\alpha^K.\, M)A \rightsquigarrow M[A/\alpha]$

S-Split
$\mathbf{let}\ \langle \ell = x; y\rangle = \langle \ell = V; W\rangle\ \mathbf{in}\ N \rightsquigarrow N[V/x, W/y]$

S-Case$_1$
$\mathbf{case}\ (\ell\, V)^R\{\ell\, x \mapsto M; y \mapsto N\} \rightsquigarrow M[V/x]$

S-Case$_2$
$\mathbf{case}\ (\ell\, V)^R\{\ell'\, x \mapsto M; y \mapsto N\} \rightsquigarrow N[(\ell\, V)^R/y], \quad \text{if } \ell \neq \ell'$

S-Let
$\mathbf{let}\ x \leftarrow \mathbf{return}\ V\ \mathbf{in}\ N \rightsquigarrow N[V/x]$

S-Lift
$M \rightsquigarrow N, \text{if } \mathcal{E}[M] \rightsquigarrow \mathcal{E}[N]$

S-Handle-Ret
$\mathbf{handle}\ (\mathbf{return}\ V)\ \mathbf{with}\ H \rightsquigarrow N[V/x], \quad \text{where } \{\mathbf{return}\ x \mapsto N\} \in H$

S-Handle-Op
$\mathbf{handle}\ \mathcal{E}[\mathbf{do}\ \ell\, V]\ \mathbf{with}\ H \rightsquigarrow N[V/x, \lambda y.\, \mathbf{handle}\ \mathcal{E}[\mathbf{return}\ y]\ \mathbf{with}\ H/k],$
where $\ell \notin BL(\mathcal{E})$ and $\{\ell\, x\, k \mapsto M\} \in H$

Evaluation contexts $\quad \mathcal{E} ::= [\,] \mid \mathbf{let}\ x \leftarrow \mathcal{E}\ \mathbf{in}\ N \mid \mathbf{handle}\ \mathcal{E}\ \mathbf{with}\ H$

Figure 3.7: Small-step operational semantics (Hillerström and Lindley, 2016).

3.2.4 Operational semantics

We give a small-step operational semantics for $\lambda^{\rho}_{\text{eff}}$. Figure 3.7 displays the operational rules. The reduction relation $\rightsquigarrow$ is defined on computation terms. The statement $M \rightsquigarrow M'$ reads: term M reduces to term M' in a single step. Most of the rules are standard. Substitution on terms is defined in the usual way. We use evaluation contexts to focus on the active expression. The interesting rules are the handler rules.

We write $BL(\mathcal{E})$ for the set of operation labels bound by $\mathcal{E}$.

$$\begin{aligned} BL([\,]) &= \emptyset \\ BL(\mathbf{let}\ x \leftarrow \mathcal{E}\ \mathbf{in}\ N) &= BL(\mathcal{E}) \\ BL(\mathbf{handle}\ \mathcal{E}\ \mathbf{with}\ H) &= BL(\mathcal{E}) \cup dom(H) \end{aligned}$$

The rule S-Handle-Ret invokes the return clause of a handler. The rule

S-Handle-op handles an operation by invoking the appropriate operation clause. The constraint $\ell \notin BL(\mathcal{E})$ ensures that no inner handler inside the evaluation context is able to handle the operation: thus a handler is able to reach past any other inner handlers that do not handle ℓ.

We write $\rightsquigarrow^+$ for the transitive closure of relation $\rightsquigarrow$. Subject reduction and type soundness for $\lambda^{\rho}_{\text{eff}}$ are standard.

Theorem 3.2.1 (Subject Reduction)**.** *If* $\Delta;\Gamma \vdash M : A!E$ *and* $M \rightsquigarrow M'$*, then* $\Delta;\Gamma \vdash M' : A!E$*.*

Proof. By induction on the given typing derivation. □

There are two ways in which a computation can terminate. It can either successfully return a value, or it can get stuck on an unhandled operation.

Definition 3.2.2. *We say that computation term* N *is normal with respect to effect* E*, if* N *is either of the form* $\textbf{\textit{return}}\, V$*, or* $\mathcal{E}[\textbf{\textit{do}}\, \ell\, W]$*, where* $\ell \in E$ *and* $\ell \notin BL(\mathcal{E})$*.*

If N is normal with respect to the empty effect $\{\cdot\}$, then N has the form $\textbf{return}\, V$.

Theorem 3.2.3 (Type Soundness)**.** *If* $\vdash M : A!E$*, then there exists* $\vdash N : A!E$*, such that* $M \rightsquigarrow^+ N \not\rightsquigarrow$*, and* N *is normal with respect to effect* E*.*

Proof. By induction on the given typing derivation. □

3.3 The Lambda intermediate language

In this section we describe OCaml IR Lambda, which is an untyped call-by-value lambda calculus. Lambda is a fairly high-level language even though it exposes low-level primitives. However a rather daunting problem with Lambda is that it does not have a formally specified semantics. As a result the behaviour of Lambda programs vary depending on the platform. As a consequence it is considerably complicated to exhibit a faithful translation into Lambda. Hopefully, this will change in the future as both Dolan (2016) and Chambert (2016) are working on defining a semantics for Lambda.

$$
\begin{array}{lrcl}
\text{Expressions} & L, I, J & ::= & x \\
& & \mid & \textbf{if } L \textbf{ then } I \textbf{ else } J \\
& & \mid & \textbf{let } x \leftarrow L \textbf{ in } J \\
& & \mid & \textbf{fun } x \mapsto L \\
& & \mid & L\, J \\
& & \mid & P \\
\\
\text{Primitives} & P & ::= & i \\
& & \mid & \langle\!\langle X \rangle\!\rangle \\
& & \mid & \textbf{field}_i\ L \\
& & \mid & \textbf{error} \\
& & \mid & \textbf{eq}(L, J) \\
& & \mid & \textbf{clone } L \\
& & \mid & \textbf{perform } L \\
& & \mid & \textbf{resume}(S, L) \\
& & \mid & \textbf{delegate}(L, J) \\
\text{Box} & & ::= & L; X \mid \cdot \\
\text{Stacks} & S & ::= & \textbf{alloc\_stack}(L, I, J) \\
\text{Integers} & i, l & \in & \mathbb{Z}
\end{array}
$$

Figure 3.8: Term syntax for Lambda.

3.3.1 Terms

The term syntax is given in Figure 3.8. The term syntax comprises: *expressions*, *primitives*, *boxes*, and *stacks*. Expressions comprise variables (x) and conditional expressions (**if** L **then** I **else** J) which evaluate one of their branches depending on the evaluation of L. Let-bindings (**let** $x \leftarrow L$ **in** J) are similar to those of $\lambda^{\rho}_{\text{eff}}$. The expression (**fun** $x \mapsto L$) is a lambda abstraction that binds the variable x in the body L. Expression application ($L\,J$) is the standard eliminator for lambda abstractions. Expressions also comprise primitives.

The primitive category comprises integers, a box introduction primitive ($\langle\!\langle X \rangle\!\rangle$) and a box elimination primitive ($\mathbf{field}_i\ L$) whose semantics is well-defined if and only if the expression L evaluates to a box in which case it projects the ith element of the given box. The **error** primitive is a special instruction which abruptly halts program execution. The generic, structural equality operator ($\mathbf{eq}(L, J)$) compares its two arguments. It returns a boolean value encoded as an integer. The operations **perform**, **resume**, **clone**, and **delegate** are all crucial to the implementation of handlers.

The $\mathbf{perform}(L)$ primitive invokes an abstract operation. An abstract operation is encoded as a two-place box, where the first component contains the operation name and the second component contains the operation argument. The $\mathbf{resume}(S, L)$ primitive evaluates the expression L with a handler S. If an abstract operation is performed in L then the run-time attempts to look up a suitable operation clause in the handler S. To forward an unhandled operation we use the $\mathbf{delegate}(L, J)$ primitive. It forwards an abstract operation name L along with the continuation, J, of the abstract operation to another enclosing **resume**-block. The $\mathbf{clone}(L)$ primitive makes a copy of an expression L. This primitive is paramount to the successful encoding of multi-shot handlers of Links as Lambda supports only linear handlers. A handler in Lambda is represented as a triple. The $\mathbf{alloc\_stack}(L, I, J)$ primitive allocates a new handler on the heap with return clause L, an exception clause I, and an operation clause J.

3.4 Translating Links into Lambda

The translation from Links to Lambda consists of three mutual recursive translation functions: $\mathcal{V}(\cdot)$ which translate a Links value into a Lambda term, $\mathcal{M}(\cdot)$

Handle translation

$$\mathcal{M}\left(\textbf{handle}\ M\ \textbf{with}\ H\right) = \textbf{resume}(\mathcal{H}\left(H\right), \mathcal{M}\left(M\right))$$

Handler stack allocation

$$\mathcal{H}\left(\{\textbf{return}\ x \mapsto M\} \uplus H\right) = \textbf{alloc\_stack}\left(\begin{array}{l} \textbf{fun}\ \mathcal{V}\left(x\right) \mapsto \mathcal{M}\left(M\right), \\ \textbf{error}, \\ \textbf{fun}\ z \mapsto \textbf{let}\ y \leftarrow \textbf{field}_1\ z\ \textbf{in} \\ \textbf{let}\ k \leftarrow \textbf{field}_2\ z\ \textbf{in}\ \mathcal{H}^{op}\left(H, y, k\right) \end{array}\right)$$

Operation clause translation

$$\begin{aligned} \mathcal{H}^{op}\left(\{\ell\, x\, k \mapsto M\} \uplus H, \langle\!\langle l; x' \rangle\!\rangle, k'\right) = {} & \textbf{if}\ \textbf{eq}(\mathcal{N}(\ell), l)\ \textbf{then} \\ & \quad \textbf{let}\ k'' \leftarrow \textbf{fun}\ y \mapsto (\textbf{clone}\ k')y \\ & \quad \textbf{in}\ \mathcal{M}\left(M\right)[k''/k, x'/x] \\ & \textbf{else}\ \mathcal{H}^{op}\left(H, z, k'\right) \\ \mathcal{H}^{op}\left(\emptyset, x, k\right) = {} & \textbf{delegate}(x, k) \end{aligned}$$

Operation invocation translation

$$\mathcal{M}\left(\textbf{do}\ \left(\ell\ V\right)^E\right) = \textbf{perform}\ \mathcal{V}\left(\left(\ell\ V\right)^E\right)$$

Operation translation

$$\mathcal{V}\left(\left(\ell\ V\right)^E\right) = \langle\!\langle \mathcal{N}(\ell); \mathcal{V}\left(V\right) \rangle\!\rangle$$

Figure 3.9: Translation of Links handlers into Lambda terms.

which translates a Links computation term into a Lambda term, and $\mathcal{H}(\cdot)$ which translates a Links handler into a Lambda term. Additionally, we require a function $\mathcal{N} : \mathcal{L} \to \mathbb{Z}$ that maps labels in Links to labels in Lambda, which happens to be just integers. Figure 3.9 shows the translation of handlers. The other syntax constructors are mostly straightforward to translate.

The (**handle** M **with** H) gets translated into a resumable context. The handler H gets translated into a Lambda handler, which is allocated using the **alloc_stack** primitive. We translate the return clause of H independently of the operation clauses. In the exception component of **alloc_stack** we place the **error** primitive. This is safe because Links does not have a distinct notion of exception as in OCaml, and thus there is no way we could ever raise an exception that would activate this component. In the operation clause component we install a function whose argument z is a box of two elements: an operation and the continuation of the operation. In the function body we dismantle the box by projecting and binding the operation to a fresh variable y. Similarly, we project and bind the continuation to k. These two variables are passed to an auxiliary translation function $\mathcal{H}^{op}(\cdot, y, k)$. This function needs to keep track of the performed operation and its continuation as it translates the operation clauses recursively.

The translation of operation clauses is rather involved. The basic idea is to translate the clauses into a chain of if-then-else expressions with each if-expression guarding a particular clause body. We check whether the label of the operation clause matches the label of the invoked operation. If they match then we create a fresh wrapper function, k'', around the continuation. This wrapper function is key to simulate multi-shot continuations with linear continuation as it clones the continuation k' before use, which effectively lets us reuse a linear continuation multiple times. Crucially, we substitute the binder of the wrapper function for the binder of the continuation in the translated body. We also substitute the binder to the operation argument. When there are no more operation clauses to translate then we insert a **delegate** which forwards the operation x and the continuation k to another enclosing handler.

3.5 Runtime representation

By using the OCaml backend we naturally inherit the OCaml run-time. OCaml implements effect handlers as heap-managed stack data structures, and as con-

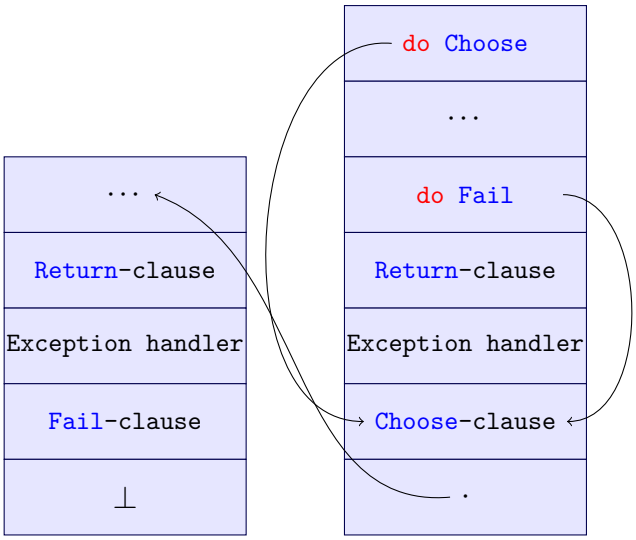

Figure 3.10: Representation of `maybeResult(randomResult(drunkToss))` at runtime.

sequence composition of handlers gives rise to n-element stacks at runtime. For example, the composition `maybeResult(randomResult(drunkToss))` is represented as a two-element stack as shown in Figure 3.10. In the figure we have explicitly unpacked the computation to exemplify how the abstract operations have immediate access to the enclosing handler's operation clause. For example, the `Choose` operation has direct access its corresponding clause in the current handler. By contrast, the `Fail` operation will have to fall-through the `Choose`-clause before finding its corresponding clause in the next handler. Thus, an invocation of an abstract operation amounts to performing a dynamic lookup to locate a suitable handler in the stack.

As our translation demonstrates, one can simulate multi-shot handlers by cloning continuations prior to invocation. The cost of cloning is linear in the size of stack. Thus it becomes rather expensive to invoke an abstract operation in a deep pipeline of handlers.

3.6 Summary

This chapter presented a native backend for the Links compiler which interfaces with the Multicore OCaml backend to provide native compilation of common language features and effect handlers. However, the translation from the Links IR to the OCaml IR is somewhat complicated by the fact that the OCaml IR

does not have a formally specified semantics. Unlike the Links IR for which we presented a formalisation. The translation shows how one may encode multi-shot handlers with linear handlers by cloning continuations prior to invocation. Purposely, the compiler does not provide any special support for concurrency.

Chapter 4

A reconstruction of the Links concurrency model with handlers

In this chapter we explore an implementation of the concurrency model of Links using effect handlers. The construction of the concurrency model spans multiple sections: in Section 4.1 we discuss a process abstraction that is somewhat faithful to the built-in process abstraction of Links in the sense that our new abstraction also tracks the effects of processes. Then in Section 4.2 we define an interface for spawning and suspending processes. In addition, we implement a basic process scheduler that we subsequently refine with more sophisticated features. In Section 4.3 we consider how to enable communication amongst processes. The two latter sections elaborate the idea outlined in Hillerström (2016).

4.1 Process abstraction

In Links each process has its own unique process identifier which makes the process uniquely identifiable. The built-in process type is an abstract type, but under the hood it is really just an integer that gets incremented each time a new process is spawned. We introduce a process handle type, `EProcess(·)`, which is a type alias for an integer:

```
typename EProcess(e::Row) = Int;
```

The name of the type is short for *effectful process* because the type is not just a simple alias, it also tracks effects. It is parameterised by a *phantom* row `e` – it is not mentioned in the definition of `EProcess` – which tracks the effects of processes. We use a small trick to get effect tracking to work as desired. Though, the trick

is quite obvious: let the type and effect system do the work for us. By making essential use of a helper function to create process handles, we can obtain effect tracking mostly for free:

```
sig makeProcessHandle : (() -e-> (), Int) -> EProcess({ |e})
fun makeProcessHandle(_, pid) { pid }
```

The cleverness is in the type signature. First, note that the function accepts as its first argument a nullary function that has effect row `e` and returns unit. It is intended that this nullary function is the computation that the process will execute. Observe that `makeProcessHandle` wholly ignores its first argument. It uses only its second argument trivially. The type system captures the effect row on the first argument and uses it to construct a process type with an open row that mentions `e`, i.e. `EProcess({ |e})`. Because the process type mentions `e` it must, by row polymorphism, have the same effects as the ignored input function. Let us try it out:

```
links> makeProcess(fun() { print("I am wild!") }, 1);
1 : EProcess ({ |wild|_ })
links> makeProcess(fun() { if (do Choose) print("True")
......                     else print("False") }, 2);
2 : EProcess ({ |Choose:Bool,wild|_ })
```

4.2 Spawning, suspending, and scheduling processes

Cooperative routines (coroutine) provide an abstraction for expressing user-defined concurrency. A coroutine is just a regular function that is evaluated by the program thread of control. Thus it is rather cheap to spawn a coroutine. However, the program thread can at most execute one coroutine at a time. The key to introduce concurrency is to have some control operators that transfer control from one coroutine to another. The caveat is that coroutines cannot provide preemptive concurrency.

4.2.1 Spawn and Yield

The interface for spawning and suspending processes consists of two operations. The operation `Spawn` takes a nullary function with an effect signature `e` as its argument and returns a process of type `EProcess({ |e})`, i.e.

```
sig pspawn : (() -e-> ()) {Spawn:(() -e-> ()) {}-> EProcess({
    |e}) |_}-> EProcess({ |e})
fun pspawn(f) { do Spawn(f) }
```

The function name `pspawn` is short for *process spawn*. The function is quite simple. Most of the work is concerned with getting the type signature right. The signature enforces any interpretation of `pspawn` to at least return a process handle. In a similar fashion we define an operation for suspending a computation:

```
sig yield : () {Yield:() |_}-> ()
fun yield() { do Yield }
```

This function is completely trivial. Next, we need to give a concrete interpretation of operations `Spawn` and `Yield`.

4.2.2 A basic round-robin scheduler

An interpretation of `Spawn` and `Yield` amounts to scheduling processes. Our programs will be running in the a single thread of control and therefore it is the responsibility of the scheduler to share execution time among processes.

We will begin by considering a simple fair, round-robin scheduler in the style of Bauer and Pretnar (2015). The main idea is to maintain a process queue with first-in-first-out (FIFO) semantics. There are two obvious scheduling policies to choose from when a process invokes the `Spawn` operation. The scheduler either enqueues the parent process and runs the child process immediately or enqueues the child process and resumes the parent process. We adopt the former policy. Using this policy an invocation of `Yield` simply enqueues the yielding process and dequeues the next process to run. The scheduler is implemented as follows:

```
1  handler basicRoundrobin {
2    case Spawn(f,resume) ->
3      var child = makeProcessHandle(f, 0);
4      enqueueProcess(fun() { resume(child) });
5      basicRoundrobin(f)()
6    case Yield(resume) ->
7      enqueueProcess(fun() { resume(()) });
8      dequeueProcess()()
9    case Return(_) ->
10     dequeueProcess()()
11 }
```

We describe the handler line by line.

Lines 2-5 handle the `Spawn` operation by first creating an process handle of type `EProcess(·)` for the new `child` process. Note that for now we carelessly assign

the identifier `0` to every process. Later we will rectify this. The handle is returned to the parent via the resumption function, `resume`. However, the evaluation of the resumption function is delayed as we store it inside a thunk. This effectively amounts to suspending the process. The thunk goes into the process queue. Afterwards the scheduler transfers control to the forked computation `f`. It is crucial that the scheduler is invoked recursively in order to handle any effects that `f` may perform.

Lines 6-8 suspend the yielding process and transfers control to the next process in the queue. The `dequeueProcess` returns a thunk which starts evaluating immediately.

Lines 9-10 handle the case when a process terminates. This leaves room for another process to run hence we dequeue the next process.

The process queue is inherently stateful as it changes every time a process is spawned or suspended. The auxiliary functions `enqueueProcess` and `dequeueProcess` use the abstract state operations `Get` and `Put` that was introduced in Section 2.3 to maintain the queue. The underlying queue data structure is the one we implemented in Section 2.1. The `enqueueProcess` simply retrieves and updates the queue, i.e.

```
sig enqueueProcess :
  (() -e-> ()) {Get: Queue(() -e-> ())
                ,Put:(Queue(() -e-> ())) {}-> ()|_}~> ()
fun enqueueProcess(f) {
  var q = enqueue(get(), f);
  put(q)
}
```

Similarly, `dequeueProcess` retrieves and removes a process from the queue. However if the queue is empty then it returns the trivial process. This way the scheduler does not need to handle any special cases. The implementation is straightforward:

```
sig dequeueProcess :
  () {Get: Queue(() -e-> ())
     ,Put:(Queue(() -e-> ())) {}-> ()|_}~> (() -e-> ())
fun dequeueProcess() {
  switch (dequeue(get())) {
    case (Just(p), q) -> put(q); p
    case (Nothing, q) -> fun() { () } # The trivial process
  }
}
```

Now all the necessary infrastructure is in place to spawn simple non-interacting processes. The following small example spawns a family of non-interacting processes:

```
fun spawnSiblings(n)() {
  if (n == 0) ()
  else {
    var p1 = pspawn(spawnSiblings(n-1));
    print("Spawned " ^^ showProcess(p1));
    var p2 = pspawn(spawnSiblings(n-1));
    print("Spawned " ^^ showProcess(p2))
  }
}
```

To run this example we plug our handlers together, i.e.

```
run -<- evalState(emptyQueue())
    -<- basicRoundrobin -< spawnSiblings(2);
```

Initially the process queue is empty hence why we seed the state handler with the empty queue. The program compiles and runs:

```
$ ./links -c basic_concur_model.links -o basic_model
$ ./basic_model
Spawned P#0
Spawned P#0
Spawned P#0
Spawned P#0
Spawned P#0
Spawned P#0
```

Under the `basicRoundrobin` scheduler a parent and its child process are indistinguishable because the scheduler assigns both of them the `0` identifier.

4.2.3 Unique processes

The scheduler `basicRoundrobin` neatly demonstrates that the essence of the scheduling is fairly simple: variations on enqueuing and dequeuing of processes. In order to have interacting processes we must to be able to uniquely identify processes.

In order to obtain a robust unique identifier generation scheme we have to detach the identifier generation from the state of the executing process. We can achieve this by turning identifier generation into an abstract operation then we can give it a stateful interpretation. Thus we introduce an operation `FreshName` that generates a new name (or identifier) of type `a`:

```
sig freshName : () {FreshName:a|_}-> a
fun freshName() {do FreshName}
```

To interpret the operation we implement a parameterised handler, `pidgenerator` that generates an increasing sequence of integers:

```
sig pidgenerator : (Int) ->
                   (Comp({FreshName:Int |e}, b)) ->
                    Comp({FreshName{_} |e}, b)
handler pidgenerator(nextPid) {
  case Return(x)     -> x
  case FreshName(k) -> k(nextPid)(nextPid+1)
}
```

Observe that the handler essentially interprets the `FreshName` operation as performing both state operations `Get` and `Put`.

Now, we have to update our round-robin scheduler to use the name generation operation. However, simply using `freshName` in the `Spawn`-clause will not be enough as the initial process is spawned through the handler. Thus the initial process would be nameless. The workaround is to define a function `upRoundrobin` ("up" for *unique processes*) that embeds a scheduler. Prior to invoking the scheduler the function generates a name for initial process:

```
fun upRoundrobin(m)() {
  var root = makeProcessHandle(m, freshName());
  handler scheduler {
    case Spawn(f,resume) ->
      var child = makeProcessHandle(f, freshName());
      enqueueProcess(fun() { resume(child) });
      scheduler(f)()
    case Yield(resume) ->
      enqueueProcess(fun() { resume(()) });
      dequeueProcess()()
    case Return(_) ->
      dequeueProcess()()
  }
  run(scheduler(m))
}
```

The implementation of `scheduler` is similar to `basicRoundrobin`. The important differences from the previous scheduler `roundrobin` are highlighted. The first invocation of `freshName` generates a name for the initial process which is the input computation `m`. The scheduler gets applied to the initial process in order to handle its effects. We end up with the following pipeline of handlers:

```
run -<- pidgenerator(0)
    -<- evalState(emptyQueue()) -<- upRoundrobin
    -<  spawnSiblings(2);
```

Compiling and running the program yields:

```
$ ./links -c up_concur_model.links -o up_model
$ ./up_model
Spawned P#1
Spawned P#2
Spawned P#3
Spawned P#4
```

```
Spawned P#5
Spawned P#6
```

As we see every process gets a unique name.

4.2.4 Self-referral

Lastly, we consider a final refinement of the round-robin scheduler: process self-referral. We introduce a new operation `Myself`:

```
sig myself : () {Myself:EProcess({ |e}) |_}-> EProcess({ |e})
fun myself() {do Myself}
```

The idea is that an invocation of `Myself` should return the handle of the calling process. We may implement this functionality through a small extension to our scheduler. The main idea is to let the scheduler keep track of the executing process. One possible way to enable this tracking is to parameterise the scheduler by the executing process:

```
1  fun roundrobin(m)() {
2    var root = makeProcessHandle(m, freshName());
3    handler scheduler(activeProcess) {
4      case Spawn(f,resume) ->
5        var childPid = freshName();
6        var child = makeProcessHandle(f, childPid);
7        enqueueProcess(fun() { resume(child)(activeProcess) });
8        scheduler(child)(f)()
9      case Yield(resume)   ->
10       enqueueProcess(fun() { resume(())(activeProcess) });
11       dequeueProcess()()
12     case Myself(resume)  ->
13       resume(activeProcess)(activeProcess)
14     case Return(_)       ->
15       dequeueProcess()()
16   }
17   run(scheduler(root)(m))
18 }
```

We describe the changes line by line.

Line 3 begins the definition of `scheduler` which takes a parameter `activeProcess` that is a handle of the current executing process.

Lines 7-8 suspend the parent process. As usual the `resume` function returns the child handle. In addition it also sets the executing process. Once `resume` gets invoked control gets transferred back to the parent process, hence the parent process becomes the active process. The `scheduler` is invoked with `child` as the active process.

Line 10 suspends the executing process and wakes up the next process to run.

Lines 12-13 returns a handle to the executing process. This clause, unlike the two previous clauses, does not alter the state of the executing process.

To illustrate the extension in action consider a variation of the `spawnSiblings` example:

```
fun spawnFamily(n)() {
  var self = myself();
  print("Spawned " ^^ showProcess(self));
  if (n == 0) { () }
  else {
    var _ = pspawn(spawnFamily(n-1));
    var _ = pspawn(spawnFamily(n-1));
    ()
  }
}
```

In this example each process announces that it has been spawned rather than its parent. Thus, we do not care about the process names returned by the calls to `pspawn`. We wire everything together as follows:

```
run -<- pidgenerator
    -<- evalState(emptyQueue()) -<- roundrobin
    -<  spawnFamily(2);
```

Compiling and running the program produce the desired result:

```
$ ./links -c rr_concur_model -o rr_model
$ ./rr_model
Spawned P#0
Spawned P#1
Spawned P#2
Spawned P#3
Spawned P#4
Spawned P#5
Spawned P#6
```

Every process gets handled uniformly by this scheduler.

4.3 Handling communication

In this section we will augment the implementation of the concurrency model with primitives for interaction between processes. Our concurrency model is an instance of a message-passing model, thus we need at least a primitive for sending messages and another for receiving them.

4.3.1 Sending and receiving

The built-in concurrency model of Links implements both blocking send and receive. We encode these as two abstract operations `Send` and `Recv`. The former is a binary operation which takes the process name of the target process and the message to send. We wrap it inside a function `psend` (for *process* send):

```
sig psend :
  (EProcess({ |e}), a) {Send:(EProcess({ |e}), a) {}-> (),
   Yield:()|_}-> ()
fun psend(proc, msg) { do Send(proc, msg); yield() }
```

Besides sending a message the function, in a rather ad-hoc fashion, also yields. However, it seems like a reasonable heuristic to yield after sending a message under the assumption that another process will be able to receive the message and make progress.

To obtain the look-and-feel of the bang operator, `!`, we define an infix operator that looks similar:

```
op proc !! msg { psend(proc, msg) }
```

Since the bang operator built deeply into the parsing level of Links we cannot override its definition in programming language level. Thus `!!` is the closest we can get.

There are several interesting designs to choose from for implementing a blocking receive. We may do the straightforward thing of "just" performing an abstract operation `Recv` leaving it for a handler to implement the blocking semantics. Alternatively, we may use type system to provide a hint to handlers of `Recv` that it is not important whether the operation is blocking or not, but rather whether a message was received or not. It is easy implement a blocking receive in terms of a non-blocking receive, e.g.

```
sig precv : () {Recv: (EProcess({ |e})) {}-> Maybe(a)
                ,Myself:EProcess({ |e}),Yield:()|_}~> a
fun precv() {
  fun loop(proc) {
    var msg = do Recv(proc);
    switch (msg) {
      case Nothing   -> yield(); loop(proc)
      case Just(msg) -> msg
    }
  }
  loop(myself())
}
```

This code implements a blocking receive. The `loop` function takes a process handle as its input and keeps looping until a message has been received. To obtain the

process handle we invoke `Myself` operation. The `loop`-body invokes the operation `Recv` which returns a `Maybe`-value. If there are no messages then the function invokes itself in order to retry. Prior to the recursive call the function yields, again this is a rather ad-hoc heuristic, but as with send it seems reasonable to transfer control when there are no messages available.

4.3.2 Facilitating communication

In the sieve example we do not know the precise number of processes, and hence the number of mailboxes, that we need ahead of time[1]. Furthermore, a mailbox is essentially a FIFO queue of messages that is private to some process. Thus we can represent a single mailbox as a pair of a process identifier and a queue. Many such pairs can be conveniently represented as a dictionary type that maps process identifiers to mailboxes. We omit the implementation details of the dictionary for brevity, however, we do present its interface:

- A type `Dictionary(k,a)` that classifies mappings from keys of type `k` to elements of type `a`.
- A function `dictEmpty : () -> Dictionary(k,a)` that returns an empty dictionary.
- A function `dictLookup : (k, Dictionary(k,a)) ~> Maybe(a)` that given a key and a dictionary returns `Just` the element associated with the key or `Nothing` if the key is not present in dictionary.
- Another function `dictModify : (k, a, Dictionary(k,a)) ~> Dictionary(k,a)` which adds a binding from a key of type `k` to an element of type `a` to a given dictionary. If the key is already bound, then that binding is overwritten.

This data structure enables us implement an auxiliary function `enqueueMessage` that appends a given message, `msg`, to the message queue belonging to the process with the given identifier, `pid`. In order to maintain the collection of mailboxes we will make use of yet another state interpretation, i.e.

[1] Although, in this particular example we could approximate the number of primes, $\pi(n)$, less than n (and thereby the number of processes) using the asymptotic law of distribution of prime numbers: $\pi(n) \approx \frac{n}{\log(n)}$.

```
sig enqueueMessage :
  (Int, a) {Get: Dictionary(Int,Queue(a))
           ,Put:(Dictionary(Int,Queue(a))) {}-> ()|_}~> ()
fun enqueueMessage(pid, msg) {
  var dict = get();
  var q = switch (dictLookup(pid, dict)) {
            case Nothing -> enqueue(emptyQueue(), msg)
            case Just(q) -> enqueue(q, msg)
          };
  put(dictModify(pid, q, dict))
}
```

First we retrieve the dictionary using the state operation `get`. Afterwards, we attempt to look up the message queue associated with `pid`. If no such queue exists, then we create a new empty queue and append the message `msg` onto it. Thus, we delay construction of a mailbox until we have some data to put inside of it. Conversely, if a queue already exists then we simply append the new message onto that queue. Finally, we bind `pid` to the new queue `q` in the dictionary.

The dual operation `dequeueMessage` dequeues a message from a mailbox. We let the function return a `Maybe`-value to account for when a mailbox is empty:

```
sig dequeueMessage :
  (Int) {Get: Dictionary(Int,Queue(a))
        ,Put:(Dictionary(Int,Queue(a))) {}-> ()|_}~> Maybe(a)
fun dequeueMessage(pid) {
  switch (dictLookup(pid, get())) {
    case Nothing -> Nothing
    case Just(q) ->
      switch (dequeue(q)) {
        case (Nothing, _) -> Nothing
        case (msg    , q) ->
          put(dictModify(pid, q, get())); msg
      }
  }
}
```

We attempt to look up the message queue associated with `pid`. If no such queue has been constructed yet, then there are no messages to dequeue hence we simply return `Nothing`. However, if a queue has already been constructed then there are two cases to consider:

1. The queue is empty: we simply return `Nothing`
2. The queue is non-empty: we dequeue a message. The `dequeue` function returns a message and the modified queue. We modify the dictionary by rebinding `pid` to point to the modified queue.

Using these two functions we give an interpretation of `Send` and `Recv`:

```
handler communication {
  case Return(x)              -> x
  case Recv(proc,resume)      ->
    var msg = dequeueMessage(getPid(proc));
    resume(msg)
  case Send(proc,msg,resume) ->
    enqueueMessage(getPid(proc), msg);
    resume(())
}
```

The implementation is straightforward.

4.4 Sieve of Eratosthenes example revisited

We need to make a few changes to sieve example from Section 2.4.1 in order to adapt it to our new concurrency implementation. The changes are only syntactical. For example, we changes the `generator` function as follows:

```
fun generator(n)() {
  var first = pspawn(sieve);
  foreach([2..n], fun(p) { first !! Candidate(p) });
  first !! Stop }
```

The function has become a curried function whose second parameter is unit. This means that applying `n` effectively returns a thunk which is precisely the type of function we can apply to a handler. The other changes are mundane: `spawn` becomes `pspawn` and `!` becomes `!!`.

We have to make the same changes in the `sieve` function. In addition to the aforementioned changes we also have to rewrite `recv` as `precv`:

```
fun sieve() {
  var myprime = fromCandidate(precv());
  print(intToString(myprime));
  fun loop(neighbour) {
    switch (precv()) {
      case Stop -> stop(neighbour)
      case Candidate(prime) ->
      if (prime `mod` myprime == 0) {
        loop(neighbour)
      } else {
        var neighbour =
          switch (neighbour) {
            case Just(pid) -> pid
            case Nothing   -> pspawn(sieve)
          };
        neighbour !! Candidate(prime);
        loop(Just(neighbour))
  } } }
  loop(Nothing)
}
```

Now we can wire everything together

```
run -<- pidgenerator
    -<- evalState(emptyDictionary()) -<- mailbox
    -<- evalState(emptyQueue())      -<- roundrobin
    -<  generator(10);
```

Next, we can compile the program and run it:

```
$ ./links -c concur_model.links -o concur_model
$ ./concur_model
2
3
5
7
```

The resulting program yields the same output as the original program we described in Section 2.4.1.

4.5 Shortcomings and limitations

Our implementation have some short comings and natural limitations. In our encoding we codify syntactic constructs like `spawn { ... }` and `!` as regular functions. An obvious implication of this is that our implementation is not backwards compatible. This is not a major issue for us since it was never our goal to stay backwards compatible. Nevertheless it is worth noting that using adapting this approach on an existing codebase is not free.

A further limitation of our implementation is that every concurrent computation must be given as a thunk to prevent premature evaluation of the concurrent computation, because `spawn` has been replaced by the function `pspawn`. In the built-in implementation one can spawn an arbitrary piece of Links code because the interpreter has special support for `spawn` primitive.

A shortcoming of our implementation is that process handles are simple aliases for integers thereby rendering the implementation unsafe as it becomes possible to manufacture process handles for non-existing processes. But this issue is orthogonal to the implementation of the concurrency model. The correct solution to this issue is to make `EProcess` an abstract type. However without a module system á la the ML module system it cumbersome to encode abstract types. Thus encoding `EProcess` as an abstract type in Links would obscure rather than clarify.

Perhaps the biggest advantage and disadvantage of our implementation is that it does not provide preemptive concurrency. On one hand it not being preemptive makes it considerably easier to reason about concurrent programs because in most

cases we can predict the interleaving of processes. On the other hand it places a burden on the programmer to control the interleaving of processes. We can relieve the programmer from this burden to some extend by inserting yields into standard library functions. In addition, we can have the compiler silently insert yields into user functions. In order to do so successfully requires great care as we have to make sure that any yield the compiler inserts appears in a handled context.

4.6 Summary

This chapter described an implementation of the message-passing concurrency model of Links. By using effect handlers to drive the implementation, we were able to modularly define scheduling policy and the communication protocol independently of one and another. We composed the handlers seamlessly to give a full interpretation of the sieve program. Furthermore, we identified and discussed some shortcomings and limitations.

Chapter 5

Experiments

In this chapter we discuss some experiments that we have performed with the compiler in order to obtain some insight into how it performs against the Links interpreter and the OCaml compiler. Furthermore, we also consider how to improve the performance of generated code.

5.1 Methodology

We have picked three benchmarks to discuss: state interpretation (Section 5.2), n-queens (Section 5.3), and our concurrency implementation (Section 5.4). The state interpretation benchmark provides insight into the cost of using a state handler. The n-queens benchmark is interesting because it crucially relies on multi-shot handlers. Lastly, the concurrency benchmark is interesting because it provides insight into how well the concurrency implementation performs against the built-in.

Our data collection method is similar to the method of Harris (2016). In the beginning of each experiment we log many environment settings, computer hardware details, number of users logged in at the time, etc. in order to be able to analyse the result meticulously. Each experiment consists of a single benchmark that we run with different configurations. For each configuration we run the particular benchmark 25 times. In our analysis we report the median execution time of the 25 runs either in raw time or relative to some baseline, each subsequent section clearly states which.

Our benchmark machine is a fairly standard desktop machine provided by the Informatics CaRD group. The hostname of the machine is "pamina". The

machine has a quad-core Intel i5-2400 processor where each core operates at 3.1 GHz. The four cores share a 6 MB Intel SmartCache. Furthermore, the machine has 8 GB of physical memory. It is running a 64 bit version of openSuSE 13.2 Linux with kernel version 3.16.7-24-desktop. The experiments were compiled using an experimental OCaml compiler `4.02.2+effects`, and further, they were ran with the runtime parameter `OCAMLRUNPARAM="s=300M"`.

5.2 State interpretation

State handling arises naturally in many applications. For example, our implementation of the concurrency model of Links in Chapter 4 makes extensive use of state and state-like interpretations. Ideally, we would want the state interpretation using handlers to be cost-free as it is the case with the *state monad* in Haskell (Kammar et al., 2013). To achieve this in practice we would have to heavily optimise handlers — something which the Links compiler does not do.

Nevertheless, we will consider how well our compiler performs compared to the Links interpreter and OCaml compiler. We use the stateful counting benchmark of Kammar et al. (2013) to benchmark state interpretations. The benchmark is fairly simple; the following is a Links port of the benchmark:

```
fun count() {
    var i = get();
    if (i == 0) { i }
    else { put(i - 1); count() }
}
```

We consider three different variations of this program: a handler variation (one program given above), a monadic variation which uses a state monad to interpret `get` and `put`, and a "pure" variation which implements state as a parameter to `count`. The latter variation represents in some sense the "abstraction-free" implementation of the counting program. We consider the monadic variation because it is (nowadays) a standard functional approach to handling state. Though, the Links compiler does not provide any special support for monads. For each implementation we instantiate the state parameter with 10000000.

Table 5.1 shows the speed of the different state implementation relative to the pure implementation compiled using the OCaml compiler (ocamlopt). Figure 5.1 visualises the results as a bar plot. As expected the Links interpreter performs much worse than the Links and OCaml compilers. Though, it is interesting to

	State implementation		
Compilation tool	Handler	Monadic	Pure
Links interpreter	0.0030	0.0001	0.0170
Links compiler	0.0089	0.0370	0.2315
ocamlopt	0.0250	0.0753	*1.0000

Table 5.1: Relative speed up of state implementations. The component marked with * is the baseline.

note that the handler interpretation performs better than the monadic interpretation. This is because the monadic implementation involves a larger series of function calls than the handler implementation, and as noted in previous work (Hillerström, 2015) function calls are rather expensive in the Links interpreter. More interestingly, we can see the Links compiler performs much worse than the OCaml compiler across all implementations. The low number for the handler implementation we can account for since the Links compiler compiles the handler as a multi-shot handler even though the state handler uses its continuation linearly, thus the handler unnecessarily copies its continuation before each invocation. However, we might reasonably expect that the performance of the pure state program should be similar for both the Links and OCaml compilers. The state program compiled by the Links compiler performs only at 23% of the program compiled by the OCaml compiler. We explore the reason for this performance in Section 5.2.1.

It is interestingly to note that the OCaml handler is rather slow. The handler performs only at 2.5% of the baseline program. In other words, the handler program is 97.5% slower than the baseline. Thus we can note that the handler abstraction is rather expensive in OCaml.

5.2.1 Recovering performance

We noted that the state programs compiled using the Links compiler performed rather poorly compared to the baseline implementation in OCaml. In this section we will try to account for difference in performance. Moreover, we will demonstrate how we can equilibrise some of the performance.

Given that the Links compiler conservatively implements the state handler as a multi-shot handler we can try to enforce linearisation of handlers in order to

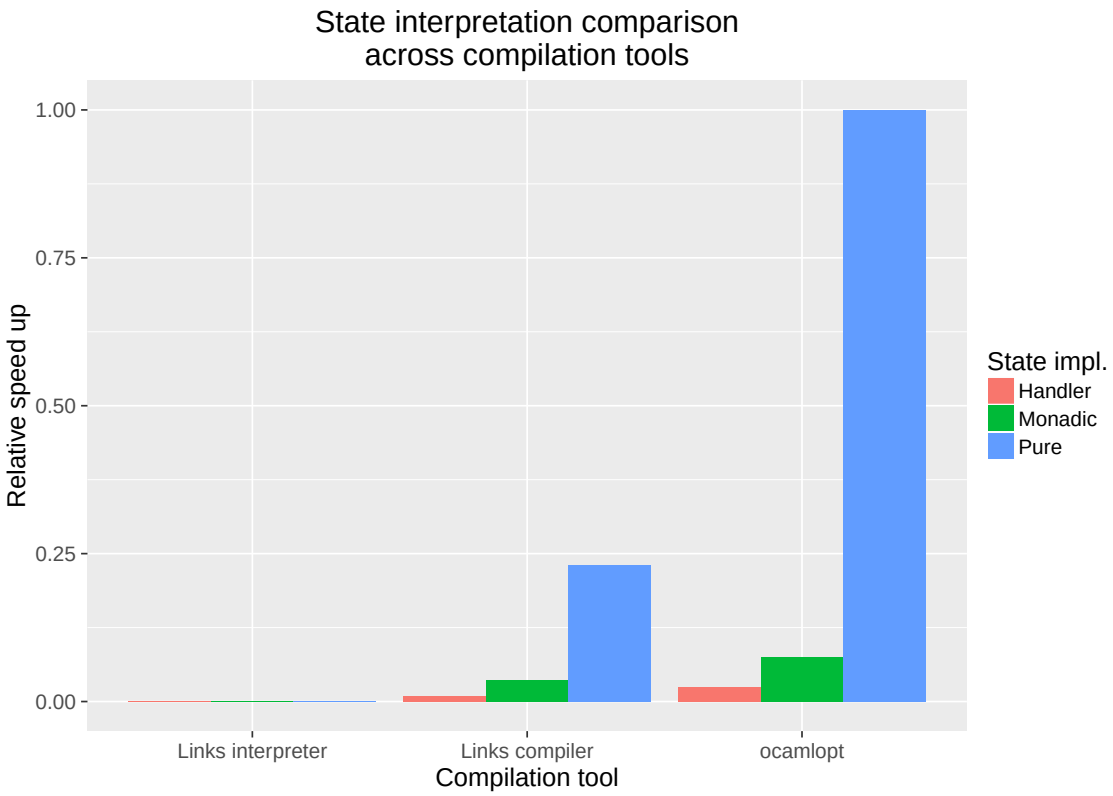

Figure 5.1: Bar plot of data from Table 5.1 (higher is better).

obtain some insight into how expensive it is to copy the continuation.

However, linearisation does not explain the poor performance of the pure state program. The explanation is, again, that the Links compiler does not perform any optimisations. In particular, the polymorphic built-in comparison operators are translated into their generic counterparts in OCaml/Lambda. This means that the equality operator in the state program gets translated into the generic equality operator in OCaml instead of the more efficient, hardware supported integer comparison operator. Since the equality testing occurs 10000000 times in the state program the overhead in occurred by the generic operator is going to be significant. This claim is supported by the results shown in Table 5.2. The "Links compiler/lin+eq" denotes a special version of the Links compiler in which we have hand-coded handlers linearisation and equality testing specialisation.

Figure 5.2 visualises the data. We achieve significant speed ups. In particular, the Links compiler out performs the OCaml compiler on the pure state program. The Links compiler achieves roughly 19% performance than the OCaml compiler on that particular program. The explanation for this difference is rather subtle: it relates to the module system of OCaml and the lack of a module system in

	State implementation		
Compilation tool	Handler	Monadic	Pure
Links interpreter	0.0030	0.0001	0.0017
Links compiler	0.0089	0.0370	0.2315
ocamlopt	0.0250	0.0753	*1.0000
Links compiler/lin+eq	0.0222	0.0438	1.1905

Table 5.2: Relative speed up of state implementations. The component marked with * is the baseline.

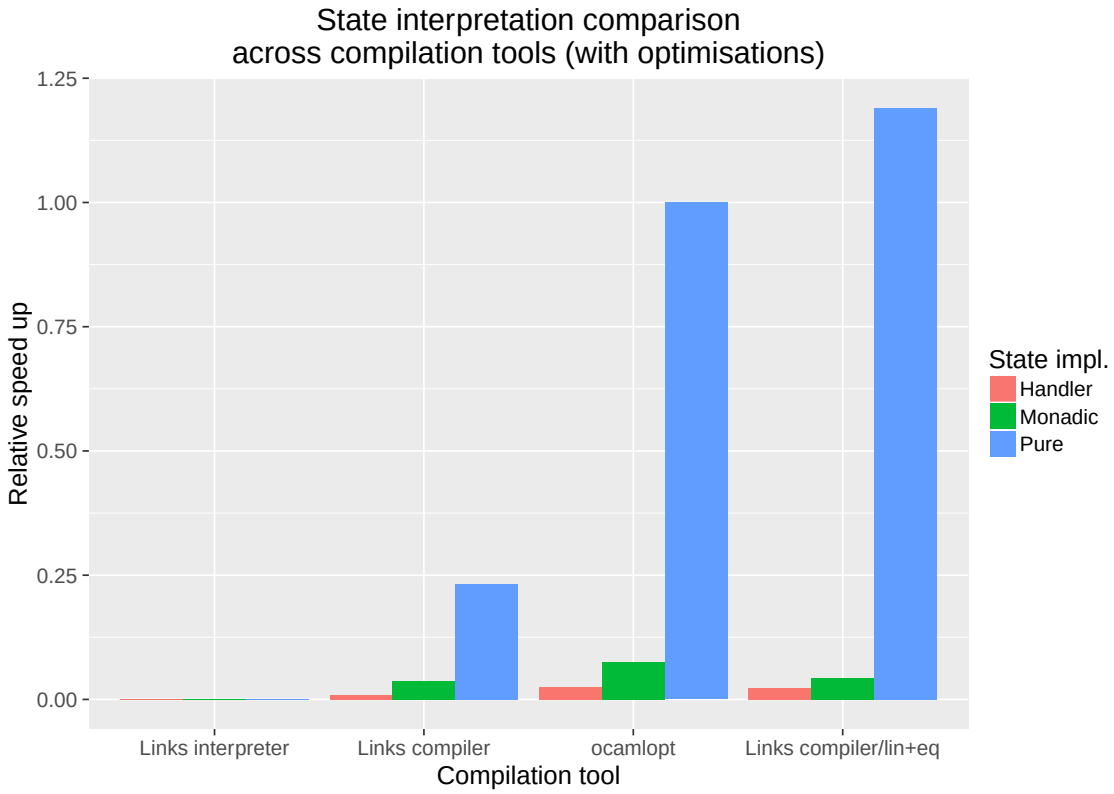

Figure 5.2: Bar plot of data in Table 5.2 (higher is better).

Links. Every OCaml source file constitutes a module. Furthermore, an accessible top-level function in one module can be invoked from another module in OCaml. In order to support cross-module invocation of functions OCaml maintains per module a global table which contains the callable top-level functions of that particular module. Since the function `count` in the state program is a global function it gets registered in the global table, which means the first invocation of `count` costs one look up in the global table in order to locate the function. In Links every function is resolved statically, because there is no concept of cross module function invocation. Therefore, we do not pay the initial look up cost.

Unsurprisingly, linearisation speeds up the handler significantly. It almost brings it on par with the OCaml implementation. Figure 5.3 depicts a comparison of the different state handler implementations relative to the OCaml implementation. The figure includes an additional special version of the Links compiler (denoted "Links compiler/lin") which only performs linearisation of handlers. We see that linearisation accounts for a large chunk of the overall improvement. Even with both optimisations the performance is still about 12% worse than the OCaml implementation. It is likely that we can close this gap by fine tuning the compiler.

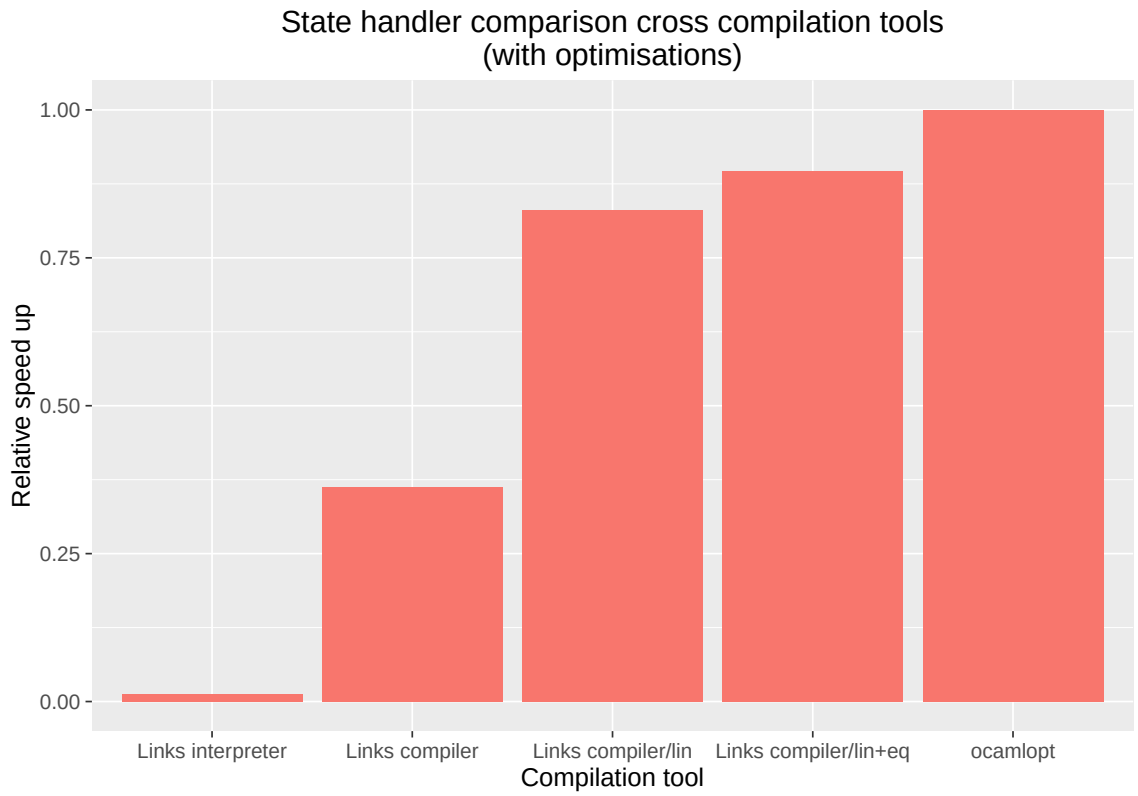

Figure 5.3: Plot of handler data from Table 5.2 with the addition of "Compiler-/lin" (higher is better).

5.3 N-Queens benchmark

The N-Queens benchmark program makes use of backtracking in order to find a solution to the queens problem for a given board size $N \times N$. The handler-based version of the program relies on multi-shot continuations in order to perform backtracking. Table 5.3 displays the speed ups relative to the Links interpreter.

	Links interpreter		**Links compiler**		**ocamlopt**
Board size	No handler	Handler	No handler	Handler	Handler
8×8	*1.0	0.97	5.93	5.93	5.93
12×12	*1.0	0.98	16.35	17.37	18.53
16×16	*1.0	0.99	246.62	239.03	621.48
20×20	*1.0	0.86	323.61	289.65	1694.21

Table 5.3: N-Queens benchmark relative speed ups. The component marked with * is the baseline across the row.

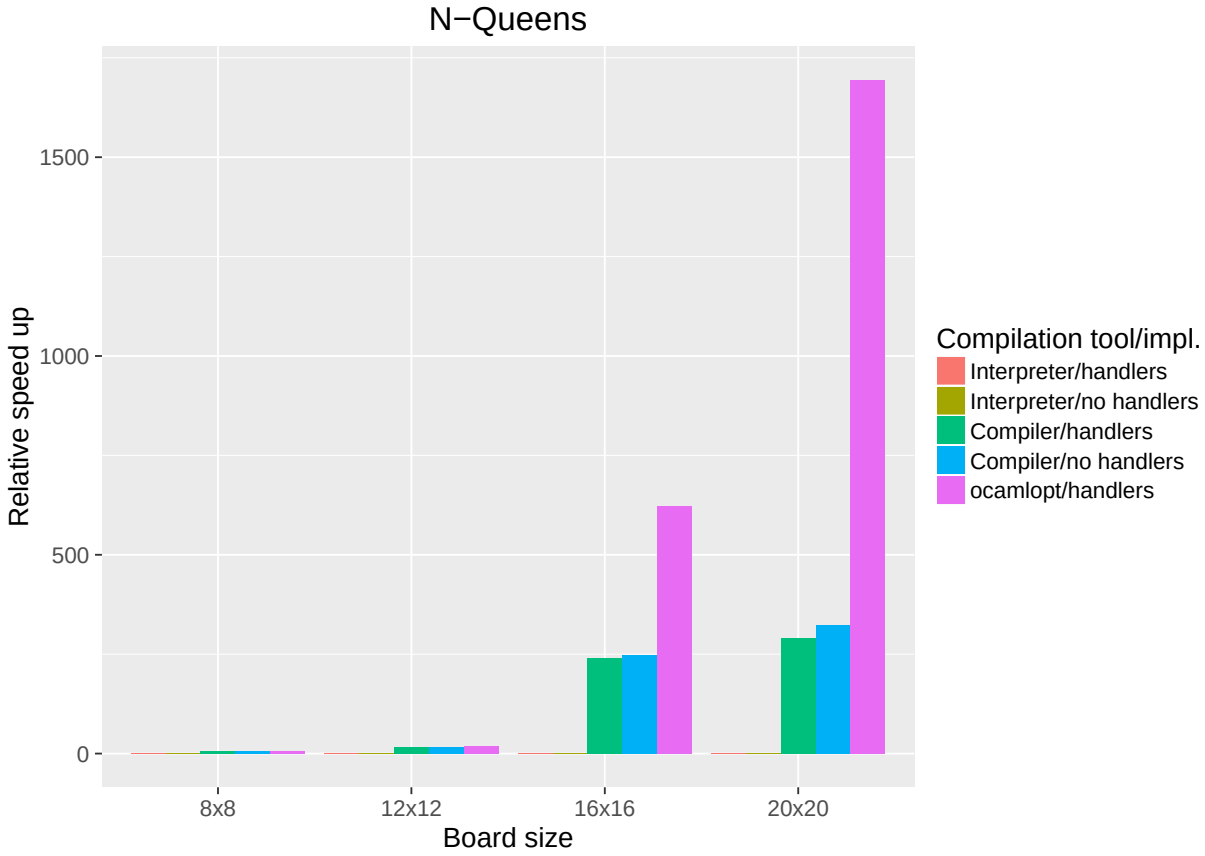

Figure 5.4: Plot of data from Table 5.3 (higher is better).

The baseline is for every board size the execution time of queens program without handlers using the Links interpreter. Figure 5.4 displays a bar plot of data from Table 5.3, whilst Figure 5.5 displays the same plot without the data for the ocamlopt compiler, which enjoys a massive speed up in 20×20-case. As the table and figures show the Links compiler beats the interpreter in every case. Interestingly, for the Links compiler the performance of the handler based program keeps well up with the performance handlerless program. This is likely because there is little or no need for backtracking for the smaller board sizes. As a result the continuation do not need to be cloned.

The massive speed up achieved by the ocamlopt compiler compared to the Links compiler is due to difference in data placement. The Lambda IR permits boxes that contain only primitive values (integers, floats, string, and characters) to be allocated on the stack rather than the heap. The Links compiler always allocates its boxes on the heap. Thus, the programs compiled by the Links compiler will perform more indirections.

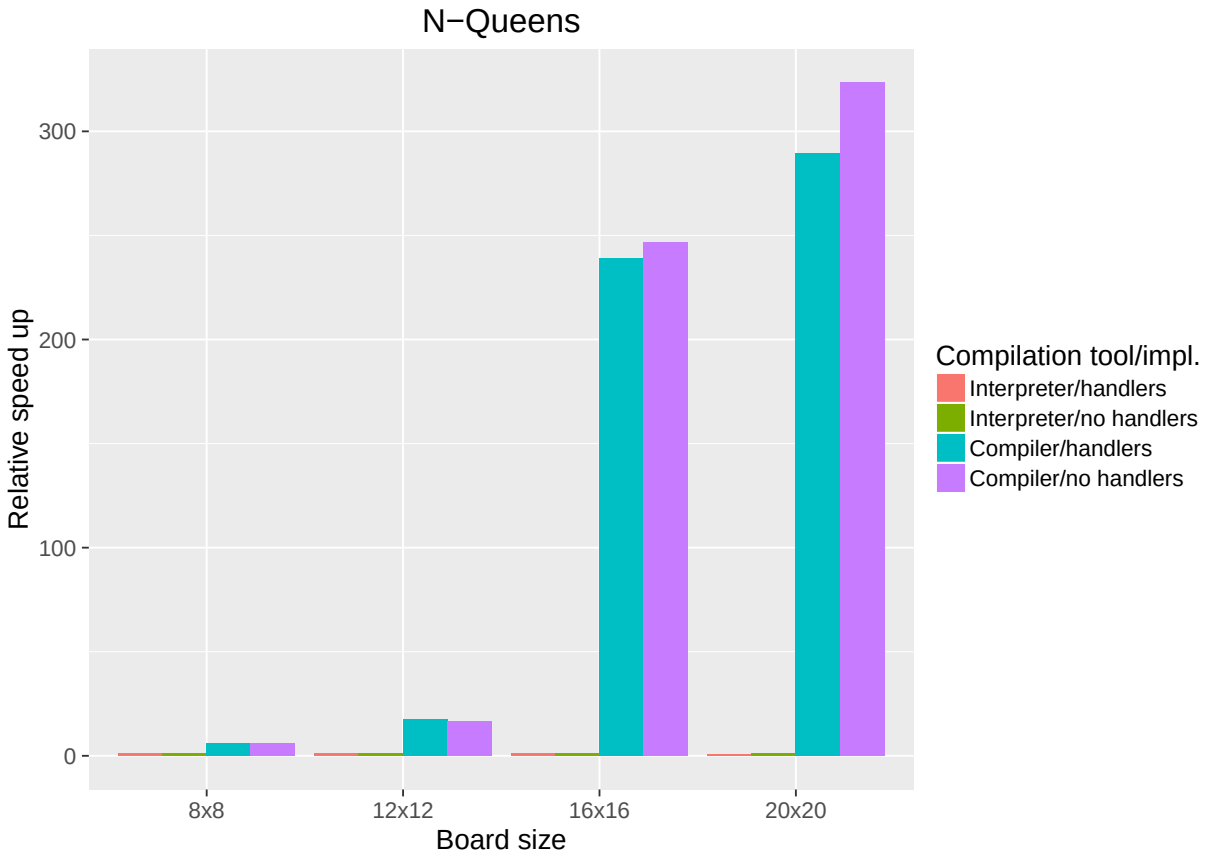

Figure 5.5: Plot of data from Table 5.3 without ocamlopt (higher is better).

N	101	201	401	601	801	1001
Number of process	27	47	80	111	140	169
Interpreter/handlers [ms]	992	2473	9589	24301	48679	84822
Interpreter/built-in [ms]	56	65	95	136	186	247
Compiler/handlers [ms]	21	38	103	231	409	658

Table 5.4: Concurrency implementation scaling (lower is better).

5.4 Concurrency implementation

We encoded the concurrency model of Links using handlers in Chapter 4. The motivation was to replace the built-in implementation. It is interesting to compare the handler encoding against the built-in implementation. Table 5.4 contains the data obtained by running the Sieve example (c.f. Sections 2.4.1 and 4.4) with different parameters. The variable N is the upper bound given to the generator process, the second row lists the number of concurrent processes that were spawned. The execution times are given in milliseconds. Figure 5.6 displays a line plot of the data. As we might expect the interpreter with the handler implementation performs poorly compared to the built-in and the compiler implementations. Figure 5.7 zooms in on the two implementations. We see that compiler implementation does not scale as well as the built-in implementation. When we have about 80 processes running the compiler implementation starts performing worse than the other.

The interpreter appears to scale surprisingly well. Needless to say, our compiled concurrency implementation should not really be out-performed by an interpreted version. If we enable the optimisations from Section 5.2.1 then we see that the compiled implementation performs better than the interpreter. Table 5.5 shows the data, and Figure 5.8 displays a visualisation of the data. In particular, it appears that the compiled implementation scales as nicely as the built-in implementation.

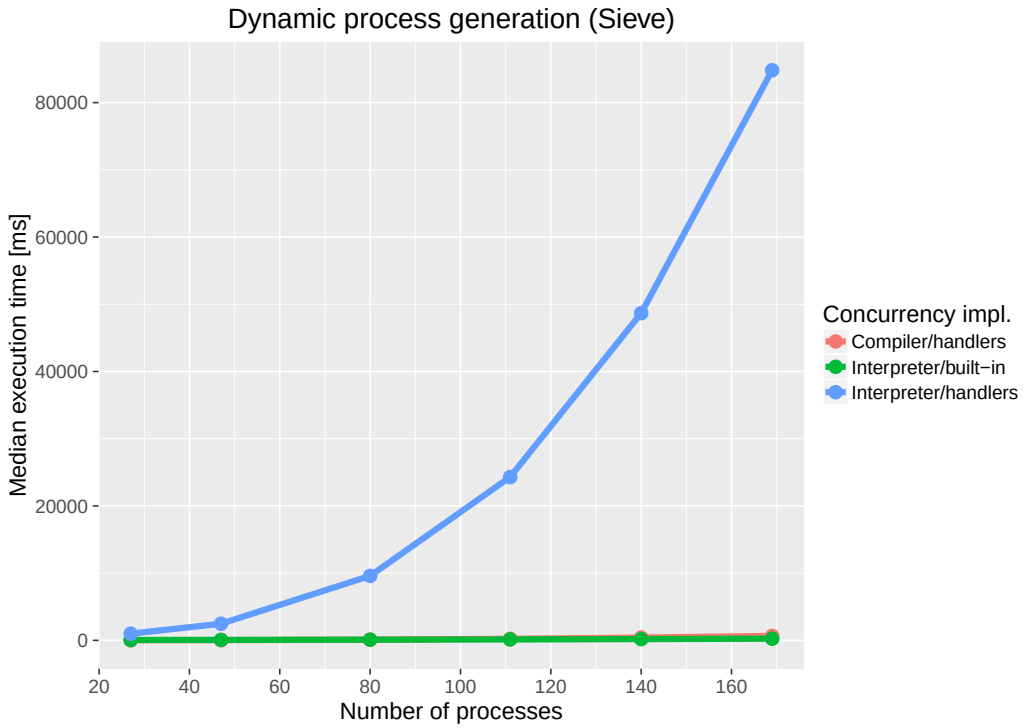

Figure 5.6: Concurrency implementation scaling (lower is better).

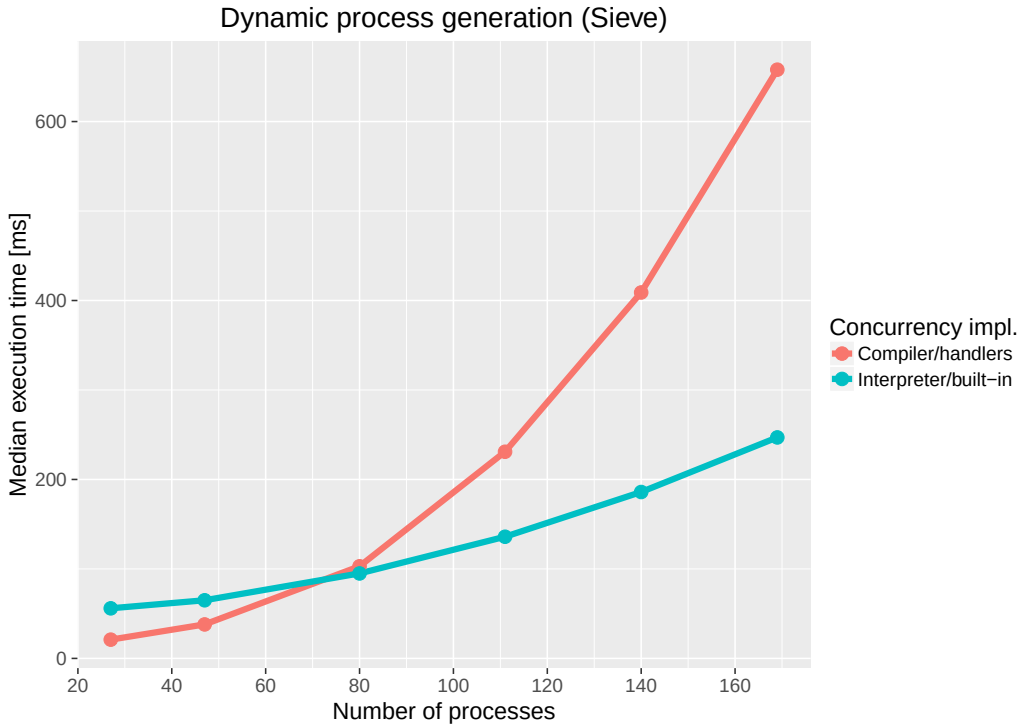

Figure 5.7: Compiler/handler vs Interpreter/built-in (lower is better).

N	101	201	401	601	801	1001
Number of process	27	47	80	111	140	169
Interpreter/built-in [ms]	56	65	95	136	186	247
Compiler/handlers [ms]	21	38	103	231	409	658
Compiler/handlers/lin+eq [ms]	17	22	36	61	99	153

Table 5.5: Concurrency implementation scaling with hand-coded optimisations.

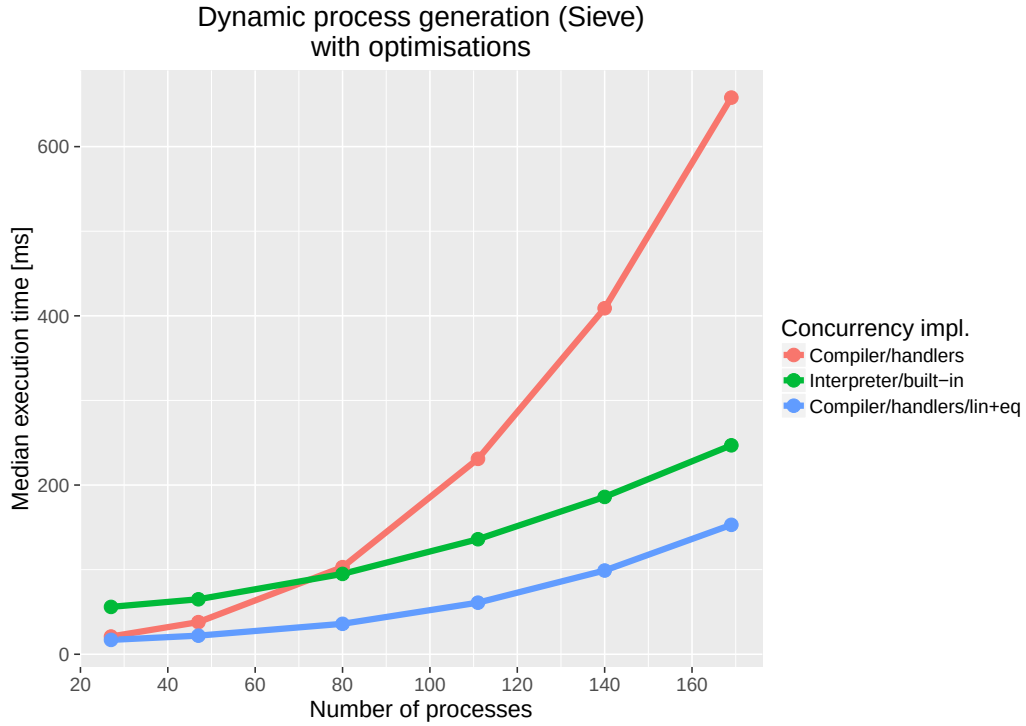

Figure 5.8: Line plot of data from Table 5.5 (lower is better).

Chapter 6

Conclusions and future work

We presented a compiler for the functional programming Links with an emphasise on the compilation of effect handlers. The compiler interfaces with the OCaml Multicore backend to generate native code. Specifically, we demonstrated a translation from the Links IR to OCaml IR known as Lambda. The Lambda intermediate language contains primitives to support compilation of linear effect handlers. Our translation demonstrates how to encode the multi-shot handlers in Lambda using a cloning primitive to copy continuations prior to invocation. However, whether our translation is faithful is unclear as Lambda does not have a formally specified semantics.

By contrast, the Links IR now have a formally specified semantics as we presented a core calculus $\lambda^{\rho}_{\text{eff}}$ that captures the essence of the Links IR.

Rather than baking concurrency support into the compiler, we presented a modular implementation of the concurrency model of Links using effect handlers. Our implementation closely approximates the built-in implementation provided by the Links interpreter. By taking advantage of the effect system, we manage to encode a process abstraction akin to the built-in one, which tracks the effects that a process may perform. We also identified some shortcomings and limitations of our concurrency implementation (c.f. Section 4.5), in particular, we do not have preemptive concurrency since the implementation uses cooperative routines to provide concurrency.

6.1 Critical evaluation

A constructive criticism is that I have not presented a good example of the usefulness of multi-shot handlers in concurrent programming. Though, I imagine that multi-shot handlers can prove useful in database transactions to implement a feature such as partial aborts in the style of Le and Fluet (2015). In addition, it would make a much more natural story to have examples that involve the unique features of Links.

In the clarity of hindsight, I should have invested more time on setting up a benchmarking infrastructure early on to provide me with continuous feedback. It was only in the latter stages of the project that I managed to set this up, and it was incredible fruitful to see graphs.

6.2 Future work

We can view Links as an experimental frontend to OCaml with a more expressive type system. Thus we can try to use Links to answer research questions that are either hard or impossible to answer in OCaml presently.

Optimisations of handlers Given the results presented in Chapter 5 to promote handlers that use their continuations linearly to linear handlers at compile time. However, every linear handler in the context of multi-shot handler must be demoted to a multi-shot handler. Otherwise linear continuations risk being consumed more than once. We believe that we can use the existing linear type system of Links to track the linearity of handlers. During code generation we can specialise the run-time representation of handlers according to their linearity.

Traversal of the handler stack is expensive for deep pipelines of handlers. If the handler stack, or part of it, is known statically, then we can imagine instantiating abstract operations already at compile time.

Improving the concurrency implementation In Section 4.5 we identified several shortcomings of the concurrency implementation. In the future we plan to improve on these. However, it may turn out be extremely hard to simulate preemptive concurrency. Caution must be taken if we are to have the compiler silently inserting yields into user-defined code, because the yields must only be inserted in the scope of a suitable handler.

Support for parallelism We have not made use of the multicore capabilities of the Multicore OCaml backend. We plan to enable multicore support in the future. Alternatively, we should be able to take an established parallel runtime such as MPI (MPI Forum, 2012) off the shelf and substitute it in place of the coroutine implementation.

Applications of multi-shot handlers in concurrency Multi-shot handlers ought to be useful in applications where we want to replay a computation. For example, we believe multi-shot handlers could provide an elegant abstraction for database transactions. In particular, they may be useful for implementing partial abort transactions (Le and Fluet, 2015) which require delimited continuations.

Shallow handlers In this dissertation we have only discussed so-called *deep handlers* which handle abstract operations uniformly. But Links also has so-called *shallow handlers* (Kammar et al., 2013). These handlers permit a nonuniform interpretation of abstract operations. With shallow handlers one must explicitly reinvoke the handler each time the continuation is used inside an operation clause. An advantage is that it makes it easy to switch to a different handler midway through a computation. A disadvantage is that shallow handlers are less easy to optimise than deep handlers Wu and Schrijvers (2015).

The Multicore OCaml backend does not yet have support for shallow handlers. It remains a question how to provide compile shallow handlers to native code.

Bibliography

G. R. Andrews. *Foundations of Multithreaded, Parallel, and Distributed Programming*. Addison-Wesley, 2000. ISBN 0201357526.

R. Atkey. Parameterised notions of computation. *J. Funct. Program.*, 19(3-4): 335–376, 2009. URL `http://dx.doi.org/10.1017/S095679680900728X`.

A. Bauer and M. Pretnar. Programming with algebraic effects and handlers. *J. Log. Algebr. Meth. Program.*, 84(1):108–123, 2015. URL `http://dx.doi.org/10.1016/j.jlamp.2014.02.001`.

E. Brady. Programming and reasoning with algebraic effects and dependent types. In *ICFP*. ACM, 2013.

C. Bruggeman, O. Waddell, and R. K. Dybvig. Representing control in the presence of one-shot continuations. In *ACM SIGPLAN Notices*, volume 31, pages 99–107. ACM, 1996.

P. Chambert. Semantics of the Lambda intermediate language. OCaml Workshop, 2016.

E. Cooper, S. Lindley, P. Wadler, and J. Yallop. Links: Web programming without tiers. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W. P. de Roever, editors, *Formal Methods for Components and Objects, 5th International Symposium, FMCO 2006, Amsterdam, The Netherlands, November 7-10, 2006, Revised Lectures*, volume 4709 of *Lecture Notes in Computer Science*, pages 266–296. Springer, 2006. URL `http://dx.doi.org/10.1007/978-3-540-74792-5_12`.

S. Dolan. Malfunctional programming. ML Workshop, 2016.

S. Dolan, L. White, K. Sivaramakrishnan, J. Yallop, and A. Madhavapeddy. Effective concurrency through algebraic effects. OCaml Workshop, 2015.

M. Felleisen and D. P. Friedman. Control operators, the secd-machine, and the λ-calculus. In *The Proceedings of the Conference on Formal Description of Programming Concepts III, Ebberup, Denmark*, pages 193–217. Elsevier, 1987.

C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *Proceedings of the ACM SIGPLAN 1993 Conference*

on Programming Language Design and Implementation, PLDI '93, pages 237–247, New York, NY, USA, 1993. ACM. ISBN 0-89791-598-4. URL http://doi.acm.org/10.1145/155090.155113.

GHC. GHC commentary: The runtime system. https://ghc.haskell.org/trac/ghc/wiki/Commentary/Rts, 2014.

T. Harris. Do not believe everything you read in the papers. Talk, Scottish Programming Language Seminar, Heriot-Watt University, June 2016. https://timharris.uk/misc/2016-nicta.pdf.

C. Hewitt and H. G. Baker. Laws for communicating parallel processes. In *IFIP Congress*, pages 987–992, 1977.

D. Hillerström. Handlers for algebraic effects in Links. Master's thesis, University of Edinburgh, Scotland, 2015.

D. Hillerström. First-class message-passing concurrency with handlers. Student Research Competition at ICFP, 2016.

D. Hillerström and S. Lindley. Liberating effects with rows and handlers. To appear at TyDe, 2016.

D. Hillerström, S. Lindley, and K. Sivaramakrishnan. Compiling Links effect handlers to the OCaml backend. ML Workshop, 2016.

S. Holmes. Compiling Links server-side code. Bachelor thesis, The University of Edinburgh, 2009.

HotSpotVM. Java SE HotSpot at a glance. http://www.oracle.com/technetwork/java/javase/tech/index-jsp-137187.html, 2016.

O. Kammar, S. Lindley, and N. Oury. Handlers in action. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 145–158, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2326-0. URL http://doi.acm.org/10.1145/2500365.2500590.

O. Kiselyov and H. Ishii. Freer monads, more extensible effects. In B. Lippmeier, editor, *Proceedings of the 8th ACM SIGPLAN Symposium on Haskell, Haskell 2015, Vancouver, BC, Canada, September 3-4, 2015*, pages 94–105. ACM, 2015. URL http://doi.acm.org/10.1145/2804302.2804319.

O. Kiselyov, A. Sabry, and C. Swords. Extensible effects: an alternative to monad transformers. In C. Shan, editor, *Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell, Boston, MA, USA, September 23-24, 2013*, pages 59–70. ACM, 2013. URL http://doi.acm.org/10.1145/2503778.2503791.

M. Le and M. Fluet. Partial aborts for transactions via first-class continuations. In K. Fisher and J. H. Reppy, editors, *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015*, pages 230–242. ACM, 2015. doi: 10.1145/2784731.2784736. URL http://doi.acm.org/10.1145/2784731.2784736.

D. Leijen. Type directed compilation of row-typed algebraic effects. Technical report, Microsoft Research, 2016. URL `https://www.microsoft.com/en-us/research/wp-content/uploads/2016/08/algeff-tr-2016-1.pdf`.

P. B. Levy, J. Power, and H. Thielecke. Modelling environments in call-by-value programming languages. *Inf. Comput.*, 185(2):182–210, 2003.

S. Lindley and J. Cheney. Row-based effect types for database integration. In B. C. Pierce, editor, *Proceedings of TLDI 2012: The Seventh ACM SIGPLAN Workshop on Types in Languages Design and Implementation, Philadelphia, PA, USA, Saturday, January 28, 2012*, pages 91–102. ACM, 2012. URL `http://doi.acm.org/10.1145/2103786.2103798`.

S. Lindley, C. McBride, and C. McLaughlin. Do be do be do, 2016. URL `http://homepages.inf.ed.ac.uk/slindley/papers/frankly-draft-july2016.pdf`. Draft, July.

Links. Basic out-of-date documentation. `http://groups.inf.ed.ac.uk/links/quick-help.html`, 2016.

C. McBride. Shonky, 2016. `https://github.com/pigworker/shonky`.

Microsoft Corp. Common language runtime (CLR). `http://msdn.microsoft.com/en-us/library/8bs2ecf4(v=vs.110).aspx`, 2016.

MPI Forum. MPI: A message-passing interface standard version 3.0, 2012.

C. Okasaki. *Purely Functional Data Structures*. Cambridge University Press, New York, NY, USA, 1998. ISBN 0-521-63124-6.

G. D. Plotkin and J. Power. Adequacy for algebraic effects. In F. Honsell and M. Miculan, editors, *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*, volume 2030 of *Lecture Notes in Computer Science*, pages 1–24. Springer, 2001. URL `http://dx.doi.org/10.1007/3-540-45315-6_1`.

G. D. Plotkin and J. Power. Algebraic operations and generic effects. *Applied Categorical Structures*, 11(1):69–94, 2003. URL `http://dx.doi.org/10.1023/A:1023064908962`.

G. D. Plotkin and M. Pretnar. Handling algebraic effects. *Logical Methods in Computer Science*, 9(4), 2013. URL `http://dx.doi.org/10.2168/LMCS-9(4:23)2013`.

M. Pretnar. An introduction to algebraic effects and handlers. invited tutorial paper. *Electr. Notes Theor. Comput. Sci.*, 319:19–35, 2015. URL `http://dx.doi.org/10.1016/j.entcs.2015.12.003`.

A. H. Saleh and T. Schrijvers. Efficient algebraic effect handlers for prolog. *arXiv preprint arXiv:1608.00816*, 2016.

K. Sivaramakrishnan, T. Harris, S. Marlow, and S. P. Jones. Composable scheduler activations for haskell. *Journal of Functional Programming*, 2016. Accepted.

W. Swierstra. Data types à la carte. *J. Funct. Program.*, 18(4):423–436, 2008. doi: 10.1017/S0956796808006758. URL `http://dx.doi.org/10.1017/S0956796808006758`.

A. van der Ploeg and O. Kiselyov. Reflection without remorse: revealing a hidden sequence to speed up monadic reflection. 2014.

N. Wu and T. Schrijvers. Fusion for free - efficient algebraic effect handlers. In R. Hinze and J. Voigtländer, editors, *Mathematics of Program Construction - 12th International Conference, MPC 2015, Königswinter, Germany, June 29 - July 1, 2015. Proceedings*, volume 9129 of *Lecture Notes in Computer Science*, pages 302–322. Springer, 2015. URL `http://dx.doi.org/10.1007/978-3-319-19797-5_15`.

N. Wu, T. Schrijvers, and R. Hinze. Effect handlers in scope. In *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell*, Haskell '14, pages 1–12, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3041-1. URL `http://doi.acm.org/10.1145/2633357.2633358`.