Liberating Effects with Rows and Handlers

Daniel Hillerström
The University of Edinburgh

Sam Lindley
The University of Edinburgh

Abstract

Algebraic effects and effect handlers provide a modular abstraction for effectful programming. They support user-defined effects, as in Haskell, in conjunction with direct-style effectful programming, as in ML. They also present a structured interface to programming with delimited continuations.

In order to be modular, it is natural for an effect system to support extensible effects. Row polymorphism is a natural abstraction for modelling extensibility at the level of types. In this paper we argue that the abstraction required to implement extensible effects and their handlers is exactly row polymorphism.

We use the Links functional web programming language as a platform to substantiate this claim. Links is a natural starting point as it uses row polymorphism for polymorphic variants, records, and its built-in effect types. It also has infrastructure for manipulating continuations. Through a small extension to Links we smoothly add support for effect handlers, making essential use of rows in the frontend and first-class continuations in the backend.

We demonstrate the usability of our implementation by modelling the mathematical game of Nim as an abstract computation. We interpret this abstract computation in a variety of ways, illustrating how rows and handlers support modularity and smooth composition of effectful computations.

We present a core calculus of row-polymorphic effects and handlers based on a variant of A-normal form used in the intermediate representation of Links. We give an operational semantics for the calculus and a novel generalisation of the CEK machine that implements the operational semantics, and prove that the two coincide.

1. Introduction

Algebraic effects [26] and effect handlers [27] are a more modular alternative to monads for managing user-defined computational effects [6, 13, 15]. Effect handlers generalise exception handlers, providing a mechanism for interpreting arbitrary algebraic effects, and they present a structured interface to programming with delimited continuations.

As a simple example consider a choice effect given by a single effectful operation:

```
Choose : Bool
```


We can write an abstract computation M that invokes `Choose`, independently of specifying the meaning of `Choose`. We can then handle M in multiple ways. For instance, we can define a handler `allResults` that interprets the operation as nondeterministic choice, returning a list of every possible outcome of M. We can also define a different handler `coin` that interprets `Choose` as random choice, returning a single value that depends on all of the random choices made in M. Where effect handlers really come into their own as a programming abstraction is when we start composing them: we can handle some effects while forwarding all others, using row types to statically track the forwarded effects. We make extensive use of handler composition in Section 2.

Many existing implementations of effect handlers are Haskell libraries. Notable examples include the effect handlers library of Kammar et al. [13], the extensible effects library of Kiselyov et al. [15] and Kiselyov and Ishii [14], and implementations based on variants of Swierstra's *data types à la carte* technique [31], such as the work of Wu et al. [33] on scoped effect handlers. Another notable effect handlers library is the Idris effects library [6]. Each of these libraries uses its own sophisticated encoding of an abstraction which amounts to a restricted form of row polymorphism. In this work we present the first, to our knowledge, implementation of effect handlers using genuine Remy-style row polymorphism [30].

Links [8] is a functional programming language for building web applications. 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 [11]. For the client, the IR is compiled to JavaScript. For the server, the IR is interpreted using a variant of the CEK machine [10]. For the database, the IR is translated into an SQL query, taking advantage of the effect type system and the subformula property to guarantee query generation [20].

Links is a strict language with Hindley-Milner type inference. Links has a row type system for polymorphic variants, records, and its built-in effect types (for concurrency and database integration [8, 20]). It also has support for manipulating first-class continuations, a feature which is central to implementing effect handlers.

The row-polymorphic effect type system and continuation support make Links a natural choice for experimenting with row-based algebraic effects and effect handlers. We have implemented an effect handlers extension to Links. Currently, it is supported only on the server-side. The frontend to our implementation makes essential use of row polymorphism, while the backend is implemented as a novel generalisation of the CEK machine.

Our main contributions are as follows.

- An implementation of effect handlers using Remy-style row polymorphism [30].
- A demonstration of the usability of our implementation illustrating how rows and handler support modularity and smooth composition of effectful computations.

- A formalisation of our implementation including a small-step call-by-value operational semantics and an abstract machine semantics, based on a novel generalisation of the CEK machine to account for effect handlers.
- A strong correspondence proof between the small-step and abstract machine semantics: every reduction in the operational semantics corresponds to a sequence of administrative steps followed by a β-step in the abstract machine.

The rest of the paper is structured as follows: Section 2 gives a tutorial introduction to programming with handlers in Links. In Section 3 we present a core calculus $\lambda^{\rho}_{\text{eff}}$ along with a type-and-effect system and a small-step operational semantics. In Section 4 we relate the operational semantics to an abstract machine semantics, that captures the essence of our implementation. In Section 5 we discuss implementation details. Related work is discussed in Section 6. Finally, in Section 7 we conclude and discuss future work.

2. Programming with Handlers in Links

To demonstrate that handlers and rows provide an elegant and modular abstraction for effectful programming, we use a simplified version of the mathematical game Nim [5] as a running example.

Starting from an abstract representation of the game, we iteratively extend it with cheat detection and high score tracking capabilities through smooth composition of handlers, without needing to change the initial representation.

2.1 The Game of Nim and Effect Rows

The game of Nim is played between two players: Alice and Bob. The game begins with a heap of *n* sticks. The players alternate to take one, two, or three sticks from the heap. Alice makes the first move. The player who takes the last stick wins the game.

We abstract over the notion of making a move by defining it as an abstract effectful operation `Move:(Player,Int) {}-> Int`, where `Player` is variant type with two constructors `Alice` and `Bob`. The first parameter to `Move` is the active player, the second parameter is the current number of sticks on the heap. We refer to the pair `(Player,Int)` as a *game configuration*. We will discuss the meaning of the braces (`{}`) prefix on the arrow shortly. In Links, abstract operations like `Move`, are invoked using the `do` primitive, for instance

```
do Move(Alice,3)
```

invokes the `Move` operation with values `Alice` and `3`. Operation names, data constructors, and type aliases all begin with a capital letter. Records, variants, and effect signatures all have row types. All typing is structural in Links, thus it is unnecessary to declare a row occupant, such as an operation, before use. However, we consider it good practice to wrap the invocation of operations as functions. Mainly because this lets us compose effects with functions seamlessly, and moreover, sometimes we want to do more than just invoking an operation, we will see an example of this in Section 2.4. We wrap `Move` as follows:

```
sig move :
  (Player,Int) {Move:(Player,Int) {}-> Int|e}-> Int
fun move(p,n) {do Move(p,n)}
```

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 function `move` invokes the operation `Move` with the parameters `p` and `n`.

In the type signature, the function arrow (`->`) is prefixed by a row enclosed in curly braces. This row is the effect signature, or *effect row*, of the function. The presence of `Move` in the effect row indicates that the function may perform the `Move` operation. Furthermore, the effect row is equipped with an effect variable `e`, which can be instantiated with additional operations. This means that `move` may be invoked in the scope of additional effects. We say an effect row is *closed* if it has no effect variable, and *open* if it does. In general an effect row consists of an unordered collection of operation specifications and an optional effect variable. An operation specification either specifies that an operation is admissible (or *present*) and has a particular type signature, or that it is absent, or that it is polymorphic in its presence. We discuss the use of absence in Section 2.5.

The effect row on the type signature of the `Move` operation itself is empty, denoted by a pair of braces (`{}`). This is always the case for abstract operations as any effects they ultimately have are conferred by their handlers.

The Nim game is modelled as two mutually recursive functions `aliceTurn` and `bobTurn`. Here we show `aliceTurn`:

```
sig aliceTurn :
  (Int) {Move:(Player,Int) {}-> Int|_}~> Player
fun aliceTurn(n) {
  if (n <= 0) Bob
  else bobTurn(n - move(Alice,n)) }
```

The parameter `n` is the current number of sticks on the heap. If *n* is zero then `Bob` wins. Otherwise, Alice makes a move and it is now Bob's turn. The definition of `bobTurn` is completely symmetric, so we omit it here for brevity.

Two observations are worth making about the effect signature of `aliceTurn`. First, the effect variable is anonymous (`_`): type (or effect) variables need not be named when they appear only once. Second, the function arrow is squiggly (`~>`), which is syntactic sugar for denoting that the computation has the *wild* effect. The wild effect captures all 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. In our current implementation intrinsic effects cannot be handled by a handler, instead they are given a predefined interpretation by the interpreter.

Links employs a strict evaluation strategy, so we thunk computations that we wish to handle, and define the following type alias:

```
typename Comp(e::Row,a) = () { |e}~> a;
```

The keyword `typename` is used to define type aliases. The `Comp` type captures our notion of abstract computation, it is an alias for a thunk with an empty, open effect row and return type `a`.

The `game` function begins a game with a given number of sticks. Alice starts:

```
sig game : (Int) ->
            Comp({Move:(Player,Int) {}-> Int|_}, Player)
fun game(n)() {aliceTurn(n)}
```

2.2 Strategies and Handlers

In general, algebraic effects come with equations [26], but as with most other implementations of effect handlers, we do not consider equations. Thus, on their own, abstract operations have no meaning; handlers give them a semantics. We can use handlers to encode particular strategies for Alice and Bob by interpreting the operation `Move`. We start by considering the *perfect* strategy, defined by $ps(n) \stackrel{\text{def}}{=} \max\{1, n \mod 4\}$, where n is the number of sticks left in the game. If player p adopts the perfect strategy, then p is guaranteed to win if on p's turn n is not divisible by four. We define a handler `pp` (short for *perfect-vs-perfect*), which assigns the perfect strategy to both players

```
1 sig pp : (Comp({Move:(Player,Int) {}-> Int|e}, a)) ->
2          Comp({Move-                          |e}, a)
3 fun pp(m)() {
4   handle(m) {
5     case Return(x)    -> x
6     case Move(p,n,k) -> k(maximum(1, n `mod` 4)) }}
```

We describe the handler line by line.

Lines 1 and 2 give the type of `pp`: it takes a computation, that may invoke the `Move` operation, and yields another computation where the operation is absent (denoted by `Move-`). The computation returns a value of type `a`. We may omit this type signature altogether as the type system is capable of inferring the appropriate types.

Lines 3 and 4 begin the definition. The curried function `pp` wraps the actual handler, which is applied to the argument m using the `handle` construct, which specifies how to interpret abstract operations through a sequence of clauses.

Line 5 is a *return clause*. It defines how to handle the final return value of the input computation. In this case, this value is simply returned as is.

Line 6 is an *operation clause*. It expresses how to handle `Move`. In general, an operation clause takes the form $Op(p_1, \ldots, p_n, k) \to M$, where $p_1, \ldots, p_n$ are patterns that bind the operation parameters and k is a pattern that binds the continuation of the computation in M. In this case `p` and `n` are bound to the active player and number of sticks in the heap, respectively. The continuation is invoked with the perfect strategy, irrespective of the player.

The handler `pp` returns a computation. For convenience we define an auxiliary function to *run* (or force) a thunk:

```
sig run : (Comp({}, a)) {}~> a
fun run(m) { m() }
```

We can now compute the winner of a game in which both players play the perfect strategy:

```
links> run(pp(game(7)));
Alice : Player
links> run(pp(game(12)));
Bob : Player
```

Syntactic Sugar The input computation in `pp` is immediately supplied to `handle`. This abstract-over-handle idiom arises frequently, so Links provides syntactic sugar for it. We can give a more succinct definition of `pp` using the `handler` keyword:

```
handler pp {
  case Return(x)    -> x
  case Move(_,n,k) -> k(maximum(1,n `mod` 4)) }
```

We may optionally choose to name the input computation by adding a parameter to the `handler` keyword in square brackets, for instance, `handler[m] pp {...}`. Naming the computation can be useful if we want to invoke the wrapper function recursively from inside the handler.

2.3 Game Trees and Multi-shot Continuations

The handler `pp` computes the winner of a particular game. It only considers one scenario in which both players play the same strategy, but we can use handlers to compute other data about a game. For instance, we can give an interpretation that computes the *game tree*. Figure 1 shows an example game tree. Each node represents the active player, and each edge corresponds to a possible move for that player. We define a game tree inductively:

```
typename GTree = [|Take:(Player,[(Int,GTree)])
                  |Winner:(Player)|];
```

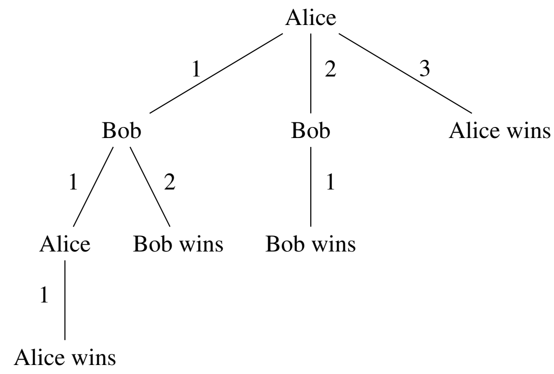

Figure 1: Game Tree Generated by `gametree(game(3))`.

The syntax `[|...|]` denotes a (polymorphic) variant type in Links in which components of the variant type are delimited by the pipe symbol (`|`). A `Take` node includes the active player and a list of possible moves, where each move is paired with the subsequent game tree. A `Winner` leaf denotes the winner of a game.

We define a handler `gametree` that generates game trees:

```
sig gametree :
   (Comp({Move:(Player,Int) {}-> Int |e}, Player)) ->
    Comp({Move-                      |e}, GTree)
handler gametree {
  case Return(x)    -> Winner(x)
  case Move(p,n,k) ->
    var subgames = map(k, validMoves(n));
    var subtrees = zip([1,2,3], subgames);
    Take(p, subtrees) }
```

The effect signatures of `gametree` and `pp` are identical, though their interpretations of `Move` differ. The return clause wraps the winning player `x` in a leaf node. The operation clause for `Move` reifies the move as a node in the game tree. The `var` keyword denotes a let binding. The crucial part is the invocation of `map` which applies the continuation multiple times, once for each valid move, enumerating every possible subgame. The function `validMoves` is a simple filter:

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

Figure 1 shows the game tree generated by the handler when $n = 3$.

2.4 Cheating and Forwarding

Thusfar we have considered a single operation `Move`, but in general we may allow arbitrary algebraic operations. We could define a monolithic handler that interprets every operation that may occur in a computation. However, a more modular alternative is to define a series of fine-grained, specialised handlers that each handle a particular operation, and then compose them together to fully interpret a computation. Fortunately, handlers compose seamlessly. Composed handlers can cooperate to interpret an abstract computation. Each handler operates on a subset of the abstract operations, leaving the remainder for other handlers. Consequently, we obtain a considerable amount of flexibility as it becomes possible to reinterpret computations by swapping in and out individual handlers.

We defer a full discussion of the role that row polymorphism plays during composition of handlers until Section 2.5. In this section we omit type signatures for handlers and instead focus on the dynamic semantics of handler composition by augmenting the game model with a cheat detection mechanism. A cheating strategy might remove all remaining sticks from the heap, thus winning in a single move. We introduce an additional operation `Cheat` to signal that a cheater has been detected. The operation is parameterised by the player, who was caught cheating:

```
sig cheat : (Player) {Cheat:(Player) {}-> Zero|_}-> _
fun cheat(p) { switch (do Cheat(p)) { } }
```

The `Cheat` operation can never return a value as its return type is the empty type `Zero`. Thus invoking `Cheat` amounts to raising an exception. Concretely an operation clause for `Cheat` can never invoke the continuation. The `switch(e){...}` construct pattern matches on the expression `e`, through a possibly empty list of clauses. We define an *exception* handler that interprets `Cheat` by outputting an error message and exiting the program:

```
handler report {
  case Return(x)       -> x
  case Cheat(Alice,_) -> error("Alice cheated!")
  case Cheat(Bob,_)   -> error("Bob cheated!")
}
```

We implement the heart of the cheat detection machinery as a handler:

```
handler checker {
  case Return(x)   -> x
  case Move(p,n,k) -> var m = move(p,n);
                      if (m `elem` validMoves(n)) k(m)
                      else cheat(p) }
```

To detect cheating the handler analyses the active player's move. If it is legal, then the game continues. Otherwise, the `Cheat` operation is invoked to signal that cheating has occurred. We may compose `pp` with `report` and `checker` to give an interpretation of a game in which no player may cheat. To make handler composition syntactically lightweight we define a pipeline operator (`-<-`) for composing handlers and another operator (`-<`) for applying a computation to a pipeline of handlers:

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

The keyword `op` is used to define infix binary operators. The operators are meant to indicate that unhandled operations are forwarded from right to left in a pipeline. In order to run a pipeline of handlers, we may apply the closed handler `run`:

```
links> run -<- pp -<- report -<- checker -< game(7);
Alice : Player
```

The `Cheat` operation is never invoked as both players play the same legal strategy. Let us define another handler that assigns the perfect strategy to `Alice` and a cheating strategy to `Bob`

```
handler pc {
  case Return(x)        -> x
  case Move(Alice,n,k) -> k(maximum(1, n `mod` 4))
  case Move(Bob,n,k)   -> k(n) }
```

Now, the cheat detection handler catches Bob:

```
links> run -<- pc -<- report -<- checker -< game(7);
*** Fatal error : Bob cheated!
```

The order of composition is important as `pc` and `checker` both handle moves. Bob gets away with cheating if we swap the two handlers:

```
links> run -<- pp -<- report -<- checker
           -<- pc -< game(7);
Bob : Player
```

Here we also use `pp`, because the type system does not know that `checker` is not performing any `Move` operations.

2.5 Composition and Row Polymorphism

In this section we discuss the typing of composed handlers. First, consider the type signature for the `report` handler:

```
sig report :
  (Comp({Cheat:(Player) {}-> Zero|e1}, a)) ->
   Comp({Cheat{p}                 |e1}, a)
```

In general, a handler accepts a computation as input and produces another computation as output. Moreover, handlers have *open* input and output effect rows, which both share the same effect variable, as a consequence both rows mention the same operation names. However, some of these operation names may be marked as absent or polymorphic in their presence. In the output effect row of `report`, the syntax `Cheat{p}` denotes that the operation is presence polymorphic. The type variable `p` can be instantiated to either present with a particular type (`:A`) or absent (`-`). Presence polymorphism is useful for seamless composition of handlers. We illustrate why by type checking the composition:

```
var f = run -<- (pp -<- report);
```

The type signature for `pp` is:

```
sig pp : (Comp({Move:(Player,Int) {}-> Int|e2}, a)) ->
          Comp({Move-                      |e2}, a)
```

The output effects of `report` must be compatible with the input effects of `pp`, therefore the composition gives rise to the following unification constraint:

```
{Move:(Player,Int) {}-> Int|e2} ∼ {Cheat{p}|e1}
```

which is solved by introducing a fresh effect variable `e3`, instantiating `e1` to `{Move:(Player,Int) {}-> Int|e2}`, and `e2` to `{Cheat{p}|e3}`. Thus the unified type is:

```
{Move:(Player,Int) {}-> Int, Cheat{p}|e3}
```

Note that with rows the order of operations is unimportant. The output of `pp` must be compatible with the input to `run`, giving rise to the constraint:

```
{Move-, Cheat{p}|e3} ∼ {}
```

which is solved by instantiating `p` to `-` and `e3` to `{}`. Thus `f` has type:

```
(Comp({Move:(Player,Int) {}-> Int,
       Cheat:(Player) {}-> Zero}, a)) {}~> a
```

Now we consider the type signature for the `checker` handler:

```
sig checker :
  (Comp({Cheat:(Player) {}-> Zero,
         Move: (Player,Int) {}-> Int|e},a)) ->
     Comp({Cheat:(Player) {}-> Zero,
           Move: (Player,Int) {}-> Int|e},a)
```

The reason `Cheat:(Player) {}-> Zero` appears in the input effect is because `Cheat` is not handled, so if a `Cheat` operation is forwarded it must have the correct type.

Remy's $\Pi ML'$ What we actually require for soundness is that *if* the `Cheat` effect is present then it must have type `(Player) {}-> Zero`, as that is the type it has in the output. In a slightly more refined system along the lines of Remy's $\Pi ML'$ [30], we could specify this as follows:

```
sig checker :
  (Comp({Cheat{_}:(Player) {}-> Zero,
         Move:    (Player,Int) {}-> Int|e},a)) ->
   Comp({Cheat:    (Player) {}-> Zero,
         Move:    (Player,Int) {}-> Int|e},a)
```

In the input row `Cheat` may or may not be present, but if it is then it must have type `(Player) {}-> Zero`.

2.6 Choice and Built-in Effects

In this section we implement the choice effect described in the introduction. We let `Bob` choose which strategy he will adopt. First, we define a wrapper for the choice operation.

```
sig choose : Comp({Choose:Bool|_}, Bool)
fun choose() {do Choose}
```

Using this operation we define a strategy selecting function in which Bob decides between playing the perfect or cheating strategy

```
fun bobChooses(m)()
 { if (choose()) pc(m)() else pp(m)() }
```

We can give a nondeterministic interpretation of `Choose` that infuses Bob with oracular powers that enable him to explore both alternatives. We define it as a handler `allResults`

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

The handler wraps the result of the input computation into a singleton list. In the `Choose`-clause the handler accumulates the results of either alternative by invoking the continuation twice.

Now, we can put everything together:

```
links> run -<- allResults -<- bobChooses -< game(7);
[Alice,Bob] : [Player]
```

Thus Bob only wins when he cheats.

Alternatively, we can replace Bob's oracular powers with a fair coin and let him perform a coin flip to decide which strategy to pick. We use Links' built-in random number generator, which returns a float from the interval $[0.0; 1.0]$:

```
sig coin : (Comp({Choose:Bool|e}, a)) ->
            Comp({Choose{_}|e}, a)
handler coin {
  case Return(x) -> x
  case Choose(k) -> if (random() > 0.5) k(true)
                    else k(false) }
```

The handler uniformly interprets `Choose` as `true` or `false`. Thus, using this handler Bob will be equally likely to either play the perfect strategy or fall victim to cheating. The computation

```
links> run -<- coin -<- bobChooses -< game(7);
```

returns either `Alice` or `Bob`. Built-in effects interact smoothly with the rest of the system.

2.7 A Scoreboard and Parameterised Handlers

As a final extension we add a scoreboard that accumulates the number of wins for each player. The scoreboard is updated after each game. We represent state as an effect with operations for reading (`Get : s`) and updating (`Put : s {}-> ()`) state of type `s`. We wrap them in the usual way:

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

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

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

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

The main difference compared to an unparameterised handler is that the continuation `k` is a curried function which 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.

We represent high scores as an association list and refer to a value of this type as the game state:

```
typename GState = [(Player,Int)];
```

We define an initial state `s0 = [(Alice,0),(Bob,0)]`. We now need a mechanism to update the game state when a game finishes. Recall that `game(n)` returns a computation whose type is:

```
Comp({Move:(Player,Int) {}-> Int|_}, Player).
```

The computation returns the winner of the game. We may exploit the fact that the return clauses of handlers are invoked in the order of composition, therefore we define a simple post-processing handler, that contains only a `Return` case, to update the scoreboard:

```
sig scoreUpdater :
 (Comp({Get:GState,Put:(GState) {}-> ()|e}, Player)) ->
    Comp({Get:GState,Put:(GState) {}-> ()|e}, Player)
handler scoreUpdater {
  case Return(x) -> var s = updateScore(x, get());
                    put(s); x }
```

The function `updateScore` is pure, it simply returns a copy of the given game state, in which the number of wins for the given player `p` has been incremented by one. The handler then reads and updates the game state. Accordingly, the composition `scoreUpdater(game(n))` causes the effect row to grow:

```
Comp({Move:(Player,Int) {}-> Int,
      Get:GState,Put:(GState) {}-> ()|_}, Player).
```

In a similar fashion, we define a handler that prints the scoreboard:

```
sig printer : (Comp({Get:GState|e}, a)) ->
               Comp({Get:GState|e}, a)
handler printer
 {  case Return(x) -> printBoard(get()); x }
```

The function `printBoard` is impure as it prints an ASCII representation of the given game state to standard out. To make matters more interesting we add replay functionality, which we implement by invoking a handler recursively on its input computation:

```
sig replay : (Int) -> (Comp({ |e}, a)) -> Comp({ |e}, a)
handler[m] replay(n)
 {case Return(x) -> if (n <= 1) x else replay(n-1)(m)()}
```

The `replay` handler reevaluates the computation `m` precisely `n` times. Note, that the handler's effect signature is an empty, open row. This means the handler forwards every operation that might occur to subsequent handlers. Now, we can wire everything together:

```
links> run -<- state(s0) -<- printer -<- replay(10) -<-
      coin -<- bobChooses -<- scoreUpdater -< game(7);
```

Figure 2 shows a possible output. In the same manner, we can effortlessly merge the cheating infrastructure into the pipeline, without changing the underlying computation.

```
/=====================\
|     NIM HIGHSCORE    |
|=====================|
|   Player   |  #Wins  |
|============|=========|
| Alice      |       7 |
|============|=========|
| Bob        |       3 |
\=====================/
```

Figure 2: Print of the Nim scoreboard after 10 games with $n = 7$, and where Alice played the perfect strategy and Bob chose between the perfect and cheating strategies.

Types

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

Figure 3: Types, effects, kinds, and environments

3. A Calculus of Handlers and Rows

In this section, we present a type and effect system and a small-step operational semantics for $\lambda_{\text{eff}}^{\rho}$ (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.

A key advantage of row polymorphism is that it integrates rather smoothly with Hindley-Milner type inference. We concern ourselves only with the explicitly-typed core language, as the treatment of type inference is quite standard.

The design of $\lambda_{\text{eff}}^{\rho}$ is inspired by the λ-calculi of Kammar et al. [13], Pretnar [29], and Lindley and Cheney [20]. As in the work of Kammar et al. [13], each handler can have its own effect signature. As in the work of Pretnar [29], the underlying formalism is fine-grain call-by-value [18], which names each intermediate computation like in A-normal form [11], but unlike A-normal form is closed under β-reduction. As in the work of Lindley and Cheney [20], the effect system is based on row polymorphism.

3.1 Types

The grammars of types, effects, kinds, and type and kind environments are given in Figure 3.

Value Types The function type $A \to C$ takes an argument of type A and returns a computation 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 given by labels of row R. Dually, the variant type $[R]$ represents a sum of fields tagged by the labels of row R. The handler type $C \Rightarrow D$ transforms a computation of type C into a computation 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.

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

Figure 4: Term Syntax

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.

Kinds We have three kinds: *Type*, $Row_{\mathcal{L}}$ and *Presence* which classify value types, row types and presence types, respectively. Row kinds are annotated with a set of labels $\mathcal{L}$. The kind of a complete row is $Row_{\emptyset}$. More generally, the kind $Row_{\mathcal{L}}$ denotes a partial row which 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 Terms

The terms are given in Figure 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 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 ($\mathbf{let}\ \langle \ell = x; y\rangle \leftarrow V\ \mathbf{in}\ N$) 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 (**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 (**return** V) returns value V. The expression (**let** $x \leftarrow M$ **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 (**handle** M **with** H) runs a computation M with handler definition H. A handler definition H consists of a return clause **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 a the continuation k_i in M_i.

We write $Id(M)$ for **handle** M **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.

3.3 Static Semantics

The kinding rules are given in Figure 5 and the typing rules are given in Figure 6.

The kinding judgement $\Delta \vdash \alpha : K$ asserts that the type variable α 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 $\Delta; \Gamma \vdash M : A!E$ states that the term M has type A and effects E under kind environment Δ and type environment Γ. In typing judgements, we implicitly assume that Γ, E and A are well-kinded with respect to Δ. We define the functions $FTV(\Gamma)$ and $FTV(E)$ to be the set of free type variables in Γ and E, respectively.

The kind 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 B given that the computation M is typeable under effect context E, and that H is a handler which transforms a computation of type A with effect signature E into another computation of type B with effect signature E'.

The T-HANDLER rule is crucial. The input effect E and the output effect E' must share the same suffix R. This means that E' 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 [13] in which the handler is implicitly wrapped around the continuation, such that any subsequent operations are handled uniformly by the same handler. The Links implementation also supports *shallow* handlers [13], in which the continuation is instead annotated with the input effect and one has to explicitly reinvoke the handler after applying the continuation inside an operation clause.

3.4 Operational Semantics

We give a small-step operational semantics for $\lambda^{\rho}_{\text{eff}}$. Figure 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. We use *evaluation contexts* to simplify the evaluation rules, by allowing us to focus on an active expression. The interesting rules are the handler rules.

TYVAR
$$\frac{}{\Delta, \alpha : K \vdash \alpha : K}$$

FORALL
$$\frac{\Delta, \alpha : K \vdash A : \mathsf{Type} \qquad \Delta, \alpha : K \vdash R : \mathsf{Row}_\emptyset}{\Delta \vdash (\forall \alpha^K . A!\{R\}) : \mathsf{Type}}$$

FUN
$$\frac{\Delta \vdash A : \mathsf{Type} \qquad \Delta \vdash R : \mathsf{Row}_\emptyset \qquad \Delta \vdash B : \mathsf{Type}}{\Delta \vdash (A \to B!\{R\}) : \mathsf{Type}}$$

RECORD
$$\frac{\Delta \vdash R : \mathsf{Row}_\emptyset}{\Delta \vdash \langle R \rangle : \mathsf{Type}}$$

VARIANT
$$\frac{\Delta \vdash R : \mathsf{Row}_\emptyset}{\Delta \vdash [R] : \mathsf{Type}}$$

PRESENT
$$\frac{\Delta \vdash A : \mathsf{Type}}{\Delta \vdash \mathsf{Pre}(A) : \mathsf{Presence}}$$

ABSENT
$$\frac{}{\Delta \vdash \mathsf{Abs} : \mathsf{Presence}}$$

EMPTYROW
$$\frac{}{\Delta \vdash \cdot : \mathsf{Row}_{\mathcal{L}}}$$

EXTENDROW
$$\frac{\Delta \vdash P : \mathsf{Presence} \qquad \Delta \vdash R : \mathsf{Row}_{\mathcal{L} \uplus \{\ell\}}}{\Delta \vdash \ell : P; R : \mathsf{Row}_{\mathcal{L}}}$$

Figure 5: Kinding 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. In our abstract machine semantics we realise this behaviour using explicit forwarding operations, but more efficient implementations are perfectly feasible.

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

Theorem 3.1 (Subject Reduction)**.** *If $\Delta; \Gamma \vdash M : A!E$ and $M \rightsquigarrow M'$, then $\Delta; \Gamma \vdash M' : A!E$.*

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. *We say that computation term N is normal with respect to effect E, if N is either of the form **return** V, or $\mathcal{E}[\mathbf{do}\ \ell\ W]$, 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 **return** V.

Theorem 3.3 (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.*

4. Abstract Machine Semantics

In this section we present an abstract machine semantics for $\lambda^{\rho}_{\text{eff}}$, which is closely related to the actual implementation of effect

Values

$$\frac{x : A \in \Gamma}{\Delta; \Gamma \vdash x : A}\ \textsc{T-Var}$$

$$\frac{\Delta; \Gamma, x : A \vdash M : C}{\Delta; \Gamma \vdash \lambda x^A . M : A \to C}\ \textsc{T-Lam}$$

$$\frac{\Delta, \alpha : K; \Gamma \vdash M : A!E \qquad \alpha \notin FTV(\Gamma)}{\Delta; \Gamma \vdash \Lambda \alpha^K . M : \forall \alpha^K . A!E}\ \textsc{T-PolyLam}$$

$$\frac{}{\Delta; \Gamma \vdash \langle\rangle : \langle\rangle}\ \textsc{T-Unit}$$

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

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

Computations

$$\frac{\Delta; \Gamma \vdash V : A \to C \qquad \Delta; \Gamma \vdash W : B}{\Delta; \Gamma \vdash V\, W : C}\ \textsc{T-App}$$

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

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

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

$$\frac{\Delta; \Gamma \vdash V : []}{\Delta; \Gamma \vdash \mathbf{absurd}^C\ V : C}\ \textsc{T-Absurd}$$

$$\frac{\Delta; \Gamma \vdash V : A}{\Delta; \Gamma \vdash \mathbf{return}\ V : A!E}\ \textsc{T-Return}$$

$$\frac{\Delta; \Gamma \vdash M : A!E \qquad \Delta; \Gamma, x : A \vdash N : B!E}{\Delta; \Gamma \vdash \mathbf{let}\ x \leftarrow M\ \mathbf{in}\ N : B!E}\ \textsc{T-Let}$$

$$\frac{\Delta; \Gamma \vdash V : A \qquad E = \{\ell : A \to B; R\}}{\Delta; \Gamma \vdash (\mathbf{do}\ \ell\ V)^E : B!E}\ \textsc{T-Do}$$

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

Handlers

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

Figure 6: Typing Rules

$$\begin{array}{lrcll}
\textsc{S-App} & (\lambda x^A . M)V & \rightsquigarrow & M[V/x] & \\
\textsc{S-TyApp} & (\Lambda \alpha^K . M)A & \rightsquigarrow & M[A/\alpha] & \\
\textsc{S-Split} & \mathbf{let}\ \langle \ell = x; y\rangle \leftarrow \langle \ell = V; W\rangle\ \mathbf{in}\ N & \rightsquigarrow & N[V/x, W/y] & \\
\textsc{S-Case}_1 & \mathbf{case}\ (\ell\, V)^R\{\ell\, x \mapsto M; y \mapsto N\} & \rightsquigarrow & M[V/x] & \\
\textsc{S-Case}_2 & \mathbf{case}\ (\ell\, V)^R\{\ell'\, x \mapsto M; y \mapsto N\} & \rightsquigarrow & N[(\ell\, V)^R/y], & \text{if } \ell \neq \ell' \\
\textsc{S-Let} & \mathbf{let}\ x \leftarrow \mathbf{return}\ V\ \mathbf{in}\ N & \rightsquigarrow & N[V/x] & \\
\textsc{S-Handle-Ret} & \mathbf{handle}\ (\mathbf{return}\ V)\ \mathbf{with}\ H & \rightsquigarrow & M[V/x], & \text{where } \{\mathbf{return}\ x \mapsto M\} \in H \\
\textsc{S-Handle-Op} & \mathbf{handle}\ \mathcal{E}[\mathbf{do}\ \ell\ V]\ \mathbf{with}\ H & \rightsquigarrow & M[V/x, \lambda y.\, \mathbf{handle}\ \mathcal{E}[\mathbf{return}\ y]\ \mathbf{with}\ H/k], & \\
 & & & & \text{where } \ell \notin BL(\mathcal{E}) \text{ and } \{\ell\ x\ k \mapsto M\} \in H
\end{array}$$

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

$$\frac{M \rightsquigarrow N}{\mathcal{E}[M] \rightsquigarrow \mathcal{E}[N]}\ \textsc{S-Lift}$$

Figure 7: Small-step Operational Semantics

Configurations	$\mathcal{C} ::= \langle M \mid \gamma \mid \kappa \rangle$
	$\mid \langle M \mid \gamma \mid \kappa \mid \kappa' \rangle_{\mathsf{op}}$
Value environments	$\gamma ::= \emptyset \mid \gamma[x \mapsto v]$
Values	$v, w ::= (\gamma, \lambda x^A . M) \mid (\gamma, \Lambda \alpha^K . M)$
	$\mid \langle\rangle \mid \langle \ell = v; w \rangle \mid (\ell\, v)^R \mid \kappa^A$
Continuations	$\kappa ::= [\,] \mid \delta :: \kappa$
Continuation frames	$\delta ::= (\sigma, \chi)$
Pure continuations	$\sigma ::= [\,] \mid \phi :: \sigma$
Pure continuation frames	$\phi ::= (\gamma, x, N)$
Handler closures	$\chi ::= (\gamma, H)$

Figure 8: Abstract Machine Syntax

handlers in Links. We prove that the abstract machine simulates the operational semantics in the sense that each reduction in the small step semantics corresponds exactly to a finite sequence of one or more steps of the abstract machine.

The Links interpreter is based on a *CEK*-style abstract machine [10] and operates directly on ANF terms [11]. The standard CEK machine operates on configurations which are triples of the form $\langle C \mid E \mid K \rangle$.

- The control C is the expression currently being evaluated.
- The environment E binds the free variables.
- The continuation K instructs the machine what to do once it is done evaluating the current term in the C component.

In order to accommodate handlers we generalise the CEK machine. The syntax of abstract machine states is given in Figure 8. Just like in the standard CEK machine, a standard configuration $\mathcal{C} = \langle M \mid \gamma \mid \kappa \rangle$ of our abstract machine is a triple of a computation term M, an environment γ mapping free variables to values, a continuation κ. However, our continuations differ from the standard machine. On the one hand, they are somewhat simplified, due to our strict separation between computations and values. On the other hand, they have considerably more structure in order to accommodate effects and handlers. In order to account for forwarding of unhandled operations, configurations occasionally gain an additional continuation argument.

Values consist of function closures, type function closures, records, variants, and captured continuations. A continuation κ consists of a stack of continuation frames $[\delta_1, \ldots, \delta_n]$. We choose to annotate captured continuations with their input type in order to make the results of Section 4.1 easier to state. Intuitively, each continuation frame $\delta = (\sigma, \chi)$ represents the pure continuation σ, corresponding to a sequence of let bindings, inside a particular handler closure χ. A pure continuation is a stack of pure continuation frames. A pure continuation frame (γ, x, N) closes a let-binding **let** $x = [\ ]$ **in** N over environment γ. A handler closure (γ, H) closes a handler definition H over environment γ.

We write $[\,]$ for an empty stack, $x :: s$ for the result of pushing x on top of stack s, and $s +\!\!+\, s'$ for the concatenation of stack s on top of s'. We use pattern matching to deconstruct stacks.

The abstract machine semantics is given in Figure 9. The transition function is given by $\longrightarrow$. This depends on an interpretation function $[\![-]\!]$ for values.

The machine is initialised (M-INIT) by placing a term in a configuration alongside the empty environment and identity continuation κ_0. The rules (M-APP), (M-TYAPP), (M-SPLIT), and (M-CASE) enact the elimination of values. Note that (M-APP) handles application of both closures and of captured continuations. The rules (M-LET) and (M-HANDLE) extend the current continuation with let bindings and handlers respectively. The rule (M-RETCONT) binds a returned value if there is a pure continuation in the current continuation frame. The rule (M-RETHANDLER) invokes the return clause of a handler if there is no pure continuation in the current continuation frame, but there is a handler. The rule (M-RETTOP) returns a final value if the continuation is empty. The rule (M-OP) switches to a special four place configuration in order to handle an operation. The fourth component of the configuration is an auxiliary forwarding continuation, which keeps track of the continuation frames through which the operation has been forwarded. It is initialised to be empty. The rule (M-OP-HANDLE) uses the current handler to handle an operation if the label matches one of the operation clauses of the current handler. The captured continuation is assigned the forwarding continuation with the current continuation frame appended to the bottom of it. The rule (M-OP-FORWARD) appends the current continuation frame onto the bottom of the forwarding continuation. Notice that if the main continuation is empty then the machine gets stuck. This occurs when an operation is unhandled, and the forwarding continuation describes the succession of handlers that have failed to handle the operation along with any pure continuations that were encountered along the way.

Assuming the input is a well-typed closed computation term $\vdash M : A!E$, the machine will either return a value of type A, or it will get stuck failing to handle an operation appearing in E. We now make the correspondence between the operational semantics and the abstract machine more precise.

4.1 Correctness

The (M-INIT) rule immediately gives us a canonical way to map a computation term onto the abstract machine. A more interesting question is how to map an arbitrary configuration to a computation term. Figure 10 describes such a mapping $(\!|-|\!)$ from configurations to terms via a collection of mutually recursive functions defined on configurations, continuations, computation terms, handler definitions, value terms, and values. We write $dom(\gamma)$ for the domain of γ, and $\gamma \backslash \{x_1, \ldots, x_n\}$ for the restriction of environment γ to $dom(\gamma) \backslash \{x_1, \ldots, x_n\}$.

The $(\!|-|\!)$ function enables us to classify the abstract machine reduction rules in according to how they relate to the operational semantics.

The rules (M-INIT) and (M-RETTOP) just concern initial input and final output, neither of which is a feature of the operational semantics, so we can ignore them. The rules (M-APPCONT), (M-LET), (M-HANDLE), (M-OP), and (M-OP-FORWARD) are administrative in the sense that $(\!|-|\!)$ is invariant under these rules. This leaves the β-rules (M-APP), (M-TYAPP), (M-SPLIT), (M-CASE), (M-RETCONT), (M-RETHANDLER), and (M-OP-HANDLE). Each of these corresponds directly with performing a reduction in the operational semantics.

We write $\longrightarrow_a$ for administrative steps, $\longrightarrow_\beta$ for β-steps, and $\Longrightarrow$ for a sequence of steps of the form $\longrightarrow_a^* \longrightarrow_\beta$.

The following lemma describes how we can simulate each reduction in the operational semantics by a sequence of administrative steps followed by one β-step in the abstract machine. The idea is to represent a computation term M by the equivalence class of configurations $\mathcal{C}$ such that $(\!|\mathcal{C}|\!) = Id(M)$. The Id wrapper captures the top-level identity handler.

Lemma 4.1. *If $M \rightsquigarrow N$, then for any $\mathcal{C}$ such that $(\!|\mathcal{C}|\!) = Id(M)$ there exists $\mathcal{C}'$ such that $\mathcal{C} \Longrightarrow \mathcal{C}'$ and $(\!|\mathcal{C}'|\!) = Id(N)$.*

Proof. By induction on the derivation of $M \rightsquigarrow N$. If $(\!|\mathcal{C}|\!) = Id(M)$, then the underlying structure of the term in the configuration $\mathcal{C}$ must be the same as M, as $(\!|-|\!)$ is homomorphic on computation terms. Some value subterms of M may appear in the environment, and part of the evaluation context of M may appear in the contin-

Identity continuation

$$\kappa_0 = [([\,], (\emptyset, \{\mathbf{return}\ x \mapsto x\}))]$$

Transition function

$$
\begin{array}{llcll}
\textsc{M-Init} & M & \longrightarrow & \langle M \mid \emptyset \mid \kappa_0 \rangle & \\[1ex]
\textsc{M-App} & \langle V\ W \mid \gamma \mid \kappa \rangle & \longrightarrow & \langle M \mid \gamma'[x \mapsto [\![W]\!]\gamma] \mid \kappa \rangle, & \text{if } [\![V]\!]\gamma = (\gamma', \lambda x^A.\, M) \\
\textsc{M-AppCont} & \langle V\ W \mid \gamma \mid \kappa \rangle & \longrightarrow & \langle \mathbf{return}\ W \mid \gamma \mid \kappa' \mathbin{+\!\!+} \kappa \rangle, & \text{if } [\![V]\!]\gamma = (\kappa')^A \\
\textsc{M-TyApp} & \langle M\ A \mid \gamma \mid \kappa \rangle & \longrightarrow & \langle M[A/\alpha] \mid \gamma' \mid \kappa \rangle, & \text{if } [\![V]\!]\gamma = (\gamma', \Lambda\alpha^K.\, M) \\
\textsc{M-Split} & \langle \mathbf{let}\ \langle \ell = x; y \rangle \leftarrow V\ \mathbf{in}\ N \mid \gamma \mid \kappa \rangle & \longrightarrow & \langle N \mid \gamma[x \mapsto v, y \mapsto w] \mid \kappa \rangle, & \text{if } [\![V]\!]\gamma = \langle \ell = v; w \rangle \\
\textsc{M-Case} & \langle \mathbf{case}\ V\, \{\ell\, x \mapsto M; y \mapsto N\} \mid \gamma \mid \kappa \rangle & \longrightarrow & \begin{cases} \langle M \mid \gamma[x \mapsto v] \mid \kappa \rangle, & \text{if } [\![V]\!]\gamma = \ell\, v \\ \langle N \mid \gamma[y \mapsto \ell'\, v] \mid \kappa \rangle, & \text{if } [\![V]\!]\gamma = \ell'\, v \text{ and } \ell \neq \ell' \end{cases} & \\
\textsc{M-Let} & \langle \mathbf{let}\ x \leftarrow M\ \mathbf{in}\ N \mid \gamma \mid (\sigma, \chi) :: \kappa \rangle & \longrightarrow & \langle M \mid \gamma \mid ((\gamma, x, N) :: \sigma, \chi) :: \kappa \rangle & \\
\textsc{M-Handle} & \langle \mathbf{handle}\ M\ \mathbf{with}\ H \mid \gamma \mid \kappa \rangle & \longrightarrow & \langle M \mid \gamma \mid ([\,], (\gamma, H)) :: \kappa \rangle & \\
\textsc{M-RetCont} & \langle \mathbf{return}\ V \mid \gamma \mid ((\gamma', x, N) :: \sigma, \chi) :: \kappa \rangle & \longrightarrow & \langle N \mid \gamma'[x \mapsto [\![V]\!]\gamma] \mid (\sigma, \chi) :: \kappa \rangle & \\
\textsc{M-RetHandler} & \langle \mathbf{return}\ V \mid \gamma \mid ([\,], (\gamma', H)) :: \kappa \rangle & \longrightarrow & \langle M \mid \gamma'[x \mapsto [\![V]\!]\gamma] \mid \kappa \rangle, & \\
 & & & & \text{if } H(\mathbf{return}) = \{\mathbf{return}\ x \mapsto M\} \\
\textsc{M-RetTop} & \langle \mathbf{return}\ V \mid \gamma \mid [\,] \rangle & \longrightarrow & [\![V]\!]\gamma & \\
\textsc{M-Op} & \langle (\mathbf{do}\ \ell\ V)^E \mid \gamma \mid \kappa \rangle & \longrightarrow & \langle (\mathbf{do}\ \ell\ V)^E \mid \gamma \mid \kappa \mid [\,] \rangle_{\mathsf{op}} & \\
\textsc{M-Op-Handle} & \langle (\mathbf{do}\ \ell\ V)^E \mid \gamma \mid (\sigma, (\gamma', H)) :: \kappa \mid \kappa' \rangle_{\mathsf{op}} & \longrightarrow & \langle M \mid \gamma'[x \mapsto [\![V]\!]\gamma, k \mapsto (\kappa' \mathbin{+\!\!+} [(\sigma, (\gamma', H))])^B] \mid \kappa \rangle, & \\
 & & & & \text{if } \ell : A \to B \in E \text{ and } H(\ell) = \{\ell\ x\ k \mapsto M\} \\
\textsc{M-Op-Forward} & \langle (\mathbf{do}\ \ell\ V)^E \mid \gamma \mid (\sigma, (\gamma', H)) :: \kappa \mid \kappa' \rangle_{\mathsf{op}} & \longrightarrow & \langle (\mathbf{do}\ \ell\ V)^E \mid \gamma \mid \kappa \mid \kappa' \mathbin{+\!\!+} [(\sigma, (\gamma', H))] \rangle_{\mathsf{op}}, & \text{if } H(\ell) = \emptyset
\end{array}
$$

Value interpretation

$$
\begin{array}{lll}
[\![x]\!]\gamma = \gamma(x) & [\![\lambda x^A.M]\!]\gamma = (\gamma, \lambda x^A.M) & [\![\Lambda\alpha^K.M]\!]\gamma = (\gamma, \Lambda\alpha^K.M) \\
[\![\langle\rangle]\!]\gamma = \langle\rangle & [\![\langle \ell = V; W \rangle]\!]\gamma = \langle \ell = [\![V]\!]\gamma; [\![W]\!]\gamma \rangle & [\![(\ell\, V)^R]\!]\gamma = (\ell\ [\![V]\!]\gamma)^R
\end{array}
$$

Figure 9: Abstract Machine Semantics

Configurations

$$(\!|\langle M \mid \gamma \mid \kappa \rangle|\!) = (\!|\kappa|\!)((\!|M|\!)\gamma) \qquad\qquad (\!|\langle M \mid \gamma \mid \kappa \mid \kappa' \rangle_{\mathsf{op}}|\!) = (\!|\kappa' \mathbin{+\!\!+} \kappa|\!)((\!|M|\!)\gamma) = (\!|\kappa'|\!)((\!|\kappa|\!)(\!|M|\!)\gamma)$$

Continuations

$$
\begin{aligned}
(\!|[\,]|\!)M &= M \\
(\!|((\gamma, x, N) :: \sigma, \chi) :: \kappa|\!)M &= (\!|(\sigma, \chi) :: \kappa|\!)(\mathbf{let}\ x \leftarrow M\ \mathbf{in}\ (\!|N|\!)(\gamma \backslash \{x\})) \\
(\!|([\,], (\gamma, H)) :: \kappa|\!)M &= (\!|\kappa|\!)(\mathbf{handle}\ M\ \mathbf{with}\ (\!|H|\!)\gamma)
\end{aligned}
$$

Computation terms

$$
\begin{aligned}
(\!|V\ W|\!)\gamma &= (\!|V|\!)\gamma\ (\!|W|\!)\gamma \\
(\!|V\ A|\!)\gamma &= (\!|V|\!)\gamma\ A \\
(\!|\mathbf{let}\ \langle \ell = x; y \rangle \leftarrow V\ \mathbf{in}\ N|\!)\gamma &= \mathbf{let}\ \langle \ell = x; y \rangle \leftarrow (\!|V|\!)\gamma\ \mathbf{in}\ (\!|N|\!)(\gamma \backslash \{x, y\}) \\
(\!|\mathbf{case}\ V\, \{\ell\ x \mapsto M; y \mapsto N\}|\!)\gamma &= \mathbf{case}\ (\!|V|\!)\gamma\ \{\ell\ x \mapsto (\!|M|\!)(\gamma \backslash \{x\}); y \mapsto (\!|N|\!)(\gamma \backslash \{y\})\} \\
(\!|\mathbf{return}\ V|\!)\gamma &= \mathbf{return}\ (\!|V|\!)\gamma \\
(\!|\mathbf{let}\ x \leftarrow M\ \mathbf{in}\ N|\!)\gamma &= \mathbf{let}\ x \leftarrow (\!|M|\!)\gamma\ \mathbf{in}\ (\!|N|\!)(\gamma \backslash \{x\}) \\
(\!|\mathbf{do}\ \ell\ V|\!)\gamma &= \mathbf{do}\ \ell\ (\!|V|\!)\gamma \\
(\!|\mathbf{handle}\ M\ \mathbf{with}\ H|\!)\gamma &= \mathbf{handle}\ (\!|M|\!)\gamma\ \mathbf{with}\ (\!|H|\!)\gamma
\end{aligned}
$$

Handler definitions

$$
\begin{aligned}
(\!|\{\mathbf{return}\ x \mapsto M\}|\!)\gamma &= \{\mathbf{return}\ x \mapsto (\!|M|\!)(\gamma \backslash \{x\})\} \\
(\!|\{\ell\ x\ k \mapsto M\} \uplus H|\!)\gamma &= \{\ell\ x\ k \mapsto (\!|M|\!)(\gamma \backslash \{x, k\}\} \uplus (\!|H|\!)\gamma
\end{aligned}
$$

Value terms and values

$$
\begin{array}{rlrl}
(\!|x|\!)\gamma &= (\!|v|\!), \quad \text{if } \gamma(x) = v & & \\
(\!|x|\!)\gamma &= x, \quad \text{if } x \notin dom(\gamma) & & \\
(\!|\lambda x^A.M|\!)\gamma &= \lambda x^A.(\!|M|\!)(\gamma \backslash \{x\}) & (\!|(\gamma, \lambda x^A.M)|\!) &= \lambda x^A.(\!|M|\!)(\gamma \backslash \{x\}) \\
(\!|\Lambda\alpha^K.M|\!)\gamma &= \Lambda\alpha^K.(\!|M|\!)\gamma & (\!|(\gamma, \Lambda\alpha^K.M)|\!) &= \Lambda\alpha^K.(\!|M|\!)\gamma \\
(\!|\langle\rangle|\!)\gamma &= \langle\rangle & (\!|\langle\rangle|\!) &= \langle\rangle \\
(\!|\langle \ell = V; W \rangle|\!)\gamma &= \langle \ell = (\!|V|\!)\gamma; (\!|W|\!)\gamma \rangle & (\!|\langle \ell = v; w \rangle|\!) &= \langle \ell = (\!|v|\!); (\!|w|\!) \rangle \\
(\!|(\ell\ V)^R|\!)\gamma &= (\ell\ (\!|V|\!)\gamma)^R & (\!|(\ell\ v)^R|\!) &= (\ell\ (\!|v|\!))^R \\
 & & (\!|\kappa^A|\!) &= \lambda x^A.(\!|\kappa|\!)(\mathbf{return}\ x)
\end{array}
$$

Figure 10: Mapping from Abstract Machine Configurations to Terms

uation. Administrative reductions update $\mathcal{C}$ by growing the continuation, whilst maintaining the invariant that $(\!|\mathcal{C}|\!) = Id(M)$. This process corresponds directly to traversing an evaluation context. It is straightforward to see that only a finite number of administrative reductions can occur consecutively as they either reduce the size of M or leave it unchanged and reduce the size of κ in the case of forwarding. Eventually the control part of the continuation will contain a redex corresponding to the active redex in M and it will then transition via a β-rule to a configuration $\mathcal{C}'$, such that $(\!|\mathcal{C}'|\!) = Id(N)$. □

The correspondence here is rather strong: there is a one-to-one mapping between $\rightsquigarrow$ and $\Longrightarrow$. The inverse of the lemma is straightforward as the semantics is deterministic. Notice that Lemma 4.1 does not require that M be well-typed. We have chosen here not to perform type-erasure, but it is straightforward to adapt the results to semantics in which all polymorphism is erased and all type annotations are erased.

Theorem 4.2 (Simulation)**.** *If* $\vdash M : A!E$*, and* $M \rightsquigarrow^+ N$*, such that* N *is normal with respect to* E*, then* $M \longrightarrow^+ \mathcal{C}$*, such that* $(\!|\mathcal{C}|\!) = N$*.*

Proof. By repeated application of Lemma 4.1. □

5. Implementation

Our implementation of handlers is based on a mild syntactic extension to Links: the syntax is extended with the `do` construct for invoking operations and the `handle(m) {...}` for handling abstract computations.

Syntactic Sugar We provide syntactic sugar to make it more convenient to program with handlers. The function $\mathcal{D}$ is a source-to-source translation that elaborates the sugar. Figure 11 shows the cases for handlers only; $\mathcal{D}$ is a homomorphism on the other syntax constructors. Crucially, handlers are desugared into a curried function, which returns a thunk after the variable `m` gets bound. This important to ensure that handlers compose smoothly. For the same reason a parameterised handler desugars into a curried function, where the parameters precede the computation argument `m`. The parameters are passed around by enclosing each operation clause by a function. Thus, the initial parameter values are applied directly to the `handle` expression.

Backend The Links interpreter is based on a CEK machine for ANF expressions. We have generalised this machine to support handlers based on the abstract machine of Section 4. The interpreter maintains a stack of handlers with first-in-last-out semantics, which makes it straightforward to implement effect forwarding. The invocation of an operation causes the interpreter to unwind the stack to find a suitable handler for the operation.

Row Polymorphism We have extended Links with support for user-defined operations, making use of the existing row type system. The current row type system is based on that of Rémy [30], adapted to support effect typing in a similar manner to the work of Leroy and Pessaux [17] and Blume et al. [4] on typing exceptions. Fields in a record can be absent, present at a particular type, or polymorphic in their presence. An earlier version of Links [20] was based on a slightly more refined variant of Remy's system, $\Pi ML'$ [30], in which the type of a label is independent of whether or not it is present. This system was abandoned because it seemed somewhat counterintuitive for the purposes of record typing, but (as indicated in Section 2.5) it may offer some advantages for effect typing.

Subtyping and Row Typing

> Subtyping is in fact a poor man's row polymorphism.
> — Andreas Rossberg[1]

Subtyping (or subeffecting) and row typing address similar concerns. However, they are not the same thing. Row polymorphism is more expressive than subtyping and subtyping is more expressive than row polymorphism. Row polymorphism allows part of an effect to be named and reused in several places. This is essential for typing polymorphic functions such as `map`. Row polymorphism can also be used whenever one might otherwise use subtyping in a first order manner. In terms of effects, this amounts to always keeping functions polymorphic in the effect, in order that the effect variable can be instantiated in order to simulate an upcast. On the other hand, subtyping applies at higher-order when row typing does not.

Links does not support subsumption (implicitly inferred subtyping), but it does support subtyping through explicit upcasts. However, such casts seem to be rarely needed in practice and can often be avoided altogether by using first-class polymorphism (another feature of Links) instead.

Shallow Handlers We have not covered them in much detail here, but our implementation also supports shallow handlers [13]. These are indicated by using the `shallowhandler` keyword in place of `handler`. Whereas a deep handler performs a fold over a computation, a shallow handler merely performs a case-split. This means that one must explicitly reinvoke the handler each time the continuation is implied 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 not much use without an external notion of recursion, and they are less easy to optimise than deep handlers [32]. The changes to the typing rules and operational semantics to accommodate shallow handlers are standard [13]. The modifications to the abstract machine are modest; the M-OP-HANDLE rule is adapted to drop the current handler and append the pure continuation on to its successor:

$$\begin{aligned}&\langle (\mathbf{do}\ \ell\ V)^E \mid \gamma \mid (\sigma, \chi) :: (\sigma', \chi') :: \kappa \mid \kappa' \rangle_{\mathsf{op}} \longrightarrow \\ &\qquad \langle M \mid \gamma'[x \mapsto [\![V]\!]\gamma, k \mapsto (\kappa')^B \mid (\sigma +\!\!+ \sigma', \chi') :: \kappa \rangle\end{aligned}$$

6. Related Work

Faking Row Polymorphism in Haskell Haskell provides a rather rich type system which allows one to simulate many aspects of row polymorphism. Perhaps the biggest mismatch between row polymorphism and most other typing features is that rows are inherently unordered, whereas other typing features are usually inherently ordered.

One approach is Swierstra's *data types a la carte* technique [31]. This amounts to encoding a row type as a sum, and then leveraging the type class system to automatically navigate through the sum type as if it was unordered. In practice, this encoding is a little fragile (e.g. sometimes additional type annotations are required), although recent improvements can make it somewhat more robust [23], particularly if one adds support for instance chains [24].

Another approach is to take advantage of the fact that type class constraints genuinely are unordered. Early work on monad transformers [19] uses this idea to write modular abstract computations, as do Kammar et al. [13], Kiselyov et al. [15], and Kiselyov and Ishii [14] in their effect handlers libraries. However, without some form of higher-order constraint solving (not supported by Haskell), one must still materialise ordered lists of effects when composing effect handlers. For many useful examples this is not a problem, but suppose we wish to build a list of handlers, from disparate sources,

[1] `http://lambda-the-ultimate.org/node/3711#comment-52984`

Handler

$$\texttt{handler h}\overline{(p)} \equiv \texttt{handler[m] h}\overline{(p)}, \text{ where } \texttt{m} \text{ is fresh.}$$

$$\mathcal{D}\left(\texttt{handler[m] h}\overline{(p)}\ \{\ \overline{c}\ \}\right) = \begin{cases} \texttt{fun h(m)() \{ handle(m) \{ } \mathcal{D}(\overline{c}) \texttt{ \} \}} & \text{if } |\overline{(p)}| = 0 \\ \texttt{fun h}\overline{(p)}\texttt{(m)() \{ handle(m) \{ } \mathcal{D}_{\overline{(p)}}(\overline{c}) \texttt{ \}}\overline{(p)} \texttt{ \}} & \text{otherwise} \end{cases}$$

Handler cases

$$\mathcal{D}_{\overline{(p)}}(\overline{c}) = \mathcal{D}_{\overline{(p)}}(c_1) \cdots \mathcal{D}_{\overline{(p)}}(c_n)$$

$$\mathcal{D}_{\overline{(p)}}(\texttt{case } q \texttt{ -> } M) = \begin{cases} \texttt{case } q \texttt{ -> } \mathcal{D}(M) & \text{if } |\overline{(p)}| = 0 \\ \texttt{case } q \texttt{ -> fun}\overline{(p)} \texttt{ \{ } \mathcal{D}(M) \texttt{ \}} & \text{otherwise} \end{cases}$$

Figure 11: Desugaring Handlers

then we need to carefully ensure that their effects are composed in the same order.

Orchard and Petricek [25] make some progress towards encoding unordered effect rows, by performing a sorting algorithm at the level of types, taking advantage of GHC's support for dependently-typed programming [34]. However, this approach can fail in practice as the type system cannot always infer that two types are equivalent in the presence of effect polymorphism.

Implementations 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 Many of the library implementations of effect handlers include implementations based on free monads [6, 13–15, 33].

Kammar et al. [13] provide an implementation of effect handlers using a continuation monad, which completely avoids materialising any data constructors. Wu and Schrijvers [32] explain how it works, by taking advantage of Haskell's fusion optimisations. This approach does appear to depend rather critically on the handlers being deep rather than shallow, and in Haskell it relies on them being type classes, and hence not really first class.

The Idris effects library [6] takes advantage of dependent types to provide effect handlers for a form of effects corresponding to parameterised monads [1]. In the effects library, effects are represented as lists of types.

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

- The Eff language [3] is a strict language with Hindley-Milner type inference similar in spirit to ML, but extended with effect handlers. It includes a novel feature for supporting fresh generation of effects in order to support effects such as ML-style higher-order state (which has an operation for generating new references). The original version of Eff [3] does not include an effect type system. However, an effect type system has subsequently been experimented with [2, 28]. This effect type system is considerably more complicated than ours. It makes essential use of subtyping, includes a region system, and a form of effect polymorphism, which one might reasonably cast as a form of row polymorphism.
- Frank [21] takes the idea of effect handlers to the extreme, having no primitive notion of function, only handlers. In Frank a function is but a special case of a handler. 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. Frank's effect system can be viewed as implementing a form of row polymorphism. Unlike Links, but much like Koka [16], Frank allows multiple occurrences of the same label in a row. In contrast rows in Links are based on Remy's design in which duplicates are not allowed, but negative information is.
- Shonky [22] 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 is quite similar to ours in that it uses a generalisation of the CEK machine. The main differences are that Shonky does not use an ANF representation, so has more forms of continuation to handle, and in contrast to our nested continuation structure, Shonky uses a completely flat structure.

Although OCaml itself has no support for effect handlers, a development branch, Multicore OCaml [9], 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, as continuations need never be copied, so they can simply be stored on the stack.

7. Conclusions and Future Work

We have implemented algebraic effects and handlers using row polymorphism and demonstrated that the extensibility of rows enables us to compose effectful computations seamlessly. We have formalised our system as the core calculus $\lambda^{\rho}_{\text{eff}}$ and shown a correspondence between two semantics: a small-step operational semantics and an abstract machine semantics, the latter of which is close to the actual implementation. We conclude by discussing ongoing and future work.

Effects are pervasive in modern web applications, thus we would like to extend our implementation to the client backend of Links. The client backend already produces JavaScript in continuation-passing style in order to implement concurrency. We plan to extend this representation to support handlers.

The overhead incurred by the Links interpreter is significant [12]. To improve performance, we are working on building a compiler backend with support for handlers. One performance bottleneck we envisage is the need to support copying of continuations. But, it is well-known that one-shot continuations can be implemented efficiently [7]. Links now has a linear type system. In future we will take advantage of this to track the linearity of handlers. Then during code generation we can specialise the run-time representation of handlers according to their linearity.

Links employs a message-passing concurrency model, similar to Erlang, but typed. Taking ideas from Multicore OCaml [9], we are investigating whether we can rebuild the Links concurrency implementation directly in terms of handlers.

References

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

[2] A. Bauer and M. Pretnar. An effect system for algebraic effects and handlers. *Logical Methods in Computer Science*, 10(4), 2014. . URL http://dx.doi.org/10.2168/LMCS-10(4:9)2014.

[3] 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.

[4] M. Blume, U. A. Acar, and W. Chae. Exception handlers as extensible cases. In G. Ramalingam, editor, *Programming Languages and Systems, 6th Asian Symposium, APLAS 2008, Bangalore, India, December 9-11, 2008. Proceedings*, volume 5356 of *Lecture Notes in Computer Science*, pages 273–289. Springer, 2008. URL http://dx.doi.org/10.1007/978-3-540-89330-1_20.

[5] C. L. Bouton. Nim, a game with a complete mathematical theory. *The Annals of Mathematics*, 3(1/4):35–39, 1901.

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

[7] 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.

[8] 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.

[9] S. Dolan, L. White, K. Sivaramakrishnan, J. Yallop, and A. Madhavapeddy. Effective concurrency through algebraic effects, 9 2015. OCaml Workshop.

[10] 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.

[11] 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.

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

[13] 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.

[14] 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.

[15] 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.

[16] D. Leijen. Koka: Programming with row polymorphic effect types. In P. Levy and N. Krishnaswami, editors, *Proceedings 5th Workshop on Mathematically Structured Functional Programming, MSFP 2014, Grenoble, France, 12 April 2014.*, volume 153 of *EPTCS*, pages 100–126, 2014. URL http://dx.doi.org/10.4204/EPTCS.153.8.

[17] X. Leroy and F. Pessaux. Type-based analysis of uncaught exceptions. *ACM Trans. Program. Lang. Syst.*, 22(2):340–377, 2000. URL http://doi.acm.org/10.1145/349214.349230.

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

[19] S. Liang, P. Hudak, and M. P. Jones. Monad transformers and modular interpreters. In R. K. Cytron and P. Lee, editors, *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23-25, 1995*, pages 333–343. ACM Press, 1995. URL http://doi.acm.org/10.1145/199448.199528.

[20] 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.

[21] S. Lindley and C. McBride. Do be do be do, 2014. URL http://homepages.inf.ed.ac.uk/slindley/papers/frankly-draft-march2014.pdf. Draft, March 2014.

[22] C. McBride. Shonky, 2016. https://github.com/pigworker/shonky.

[23] J. G. Morris. Variations on variants. In B. Lippmeier, editor, *Proceedings of the 8th ACM SIGPLAN Symposium on Haskell, Haskell 2015, Vancouver, BC, Canada, September 3-4, 2015*, pages 71–81. ACM, 2015. URL http://doi.acm.org/10.1145/2804302.2804320.

[24] J. G. Morris and M. P. Jones. Instance chains: type class programming without overlapping instances. In P. Hudak and S. Weirich, editors, *Proceeding of the 15th ACM SIGPLAN international conference on Functional programming, ICFP 2010, Baltimore, Maryland, USA, September 27-29, 2010*, pages 375–386. ACM, 2010. . URL http://doi.acm.org/10.1145/1863543.1863596.

[25] D. A. Orchard and T. Petricek. Embedding effect systems in haskell. In W. Swierstra, editor, *Proceedings of the 2014 ACM SIGPLAN symposium on Haskell, Gothenburg, Sweden, September 4-5, 2014*, pages 13–24. ACM, 2014. . URL http://doi.acm.org/10.1145/2633357.2633368.

[26] 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.

[27] 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.

[28] M. Pretnar. Inferring algebraic effects. *Logical Methods in Computer Science*, 10(3), 2014. URL http://dx.doi.org/10.2168/LMCS-10(3:21)2014.

[29] 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.

[30] D. Rémy. Theoretical aspects of object-oriented programming. chapter Type Inference for Records in Natural Extension of ML, pages 67–95. MIT Press, Cambridge, MA, USA, 1994. ISBN 0-262-07155-X. URL http://dl.acm.org/citation.cfm?id=186677.186689.

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

[32] 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.

[33] 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.

[34] B. A. Yorgey, S. Weirich, J. Cretin, S. L. P. Jones, D. Vytiniotis, and J. P. Magalhães. Giving haskell a promotion. 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 53–66. ACM, 2012. URL http://doi.acm.org/10.1145/2103786.2103795.