M1 Internship Report : Effect Typing for Links

2 May 2022 – 22 July 2022, The University of Edinburgh

ROBIN JOURDE, École Normale Supérieure de Lyon, France
SUPERVISED BY SAM LINDLEY and DANIEL HILLERSTRÖM, The University of Edinburgh, United Kingdom

This paper is the report of my internship at the School of Informatics of the University of Edinburgh with Sam Lindley and Daniel Hillerström from 2 May 2022 to 22 July 2022. Effect handlers are a language feature that is becoming more and more widespread. They allow the user to define and compose many functionalities from exceptions to concurrency and probabilistic programming. Links is an experimental tierless functional language for the web, mainly developed in the University of Edinburgh. Among its numerous features, Links supports effect handlers. The goal of my internship was to improve the effect typing system of Links, allowing aliases, avoiding pollution and allowing polymorphic operations.

Contents

Authors' addresses: Robin Jourde, École Normale Supérieure de Lyon, France, Robin.Jourde@ens-lyon.fr; Supervised by Sam Lindley, Sam.Lindley@ed.ac.uk; Daniel Hillerström, Daniel.Hillerstrom@ed.ac.uk, The University of Edinburgh, United Kingdom.

1 INTRODUCTION

TODO

2 LINKS

> *Links is a functional programming language designed to make web programming easier.*
>
> Links website [Webb]

Links [CLWY07] is a tierless functional programming language for the web developed at the University of Edinburgh. Typical web development involves the manipulation of several *tiers* which means several, sometimes really different, programming languages :

- the server side written in Python or Java for instance
- the client side in JavaScript and HTML/CSS
- the database queries in SQL

This is neither elegant nor really practical and prevents easy binding between different parts of the program that need to communicate data. It can cause *impedance mismatch*, ie situations where the objects required by the server do not match what the database provide and vice versa.

Links avoids these by providing the programmer one single language for each tier (HTML/CSS are still needed) that is compiled into bytecode for the server, JavaScript for the client and SQL for the queries.

Links also supports numerous useful features including

- polymorphic variants
- regular expressions
- session types
- row typing
- effect handlers (see 3)

It is used as an experimental language by researchers to try out new ideas and what could be future widespread programming languages features.

Links is strict, statically typed and use a Hindley-Milner type system as well as a Rémy style row typing system 4.1 [Ré94]. Its syntax is inspired from JavaScript and ML. Its compiler [Weba] is written in OCaml.

Remark 2.1. In appendix B, the reader could find some technical points on Links syntax and semantics that are not worth it to develop here. He is invited to look at it whenever needed.

3 EFFECT HANDLERS

In functional programming, functions are seen as black boxes that takes an input and computes an output. Though this view of *pure* functions is convenient and pleases λ-calculus fans, it fails to accurately represent what real programming is. Computations interact with the outer world by many means :

- input/output
- concurrency
- exceptions
- choice and non determinism
- ...

The "outer world" here stand for anything that is not the pure computation, from memory, other threads and user interaction to the evaluation context itself. These interactions not taken into account by the black-box model are called *effects*.

Gordon Plotkin and Matija Pretnar [PP09] introduced a construct that allow *composable* and *customisable* user-defined interpretation of effects : *effect handlers*. They give the programmer direct access to the context, in the form of a first class continuation, often called *resumption*.

Effect handlers can be seen as a generalisation of exception handlers. One can perform an effect in the same way as one could raise an exception and this will transfer the flow control to a handler. It is given the delimited continuation between the point where the effect is performed and where it is handled and can use it

0 times and act as an exception handler
once (linear handler)
twice or more (non-linear handler) to simulate a choice for instance

Effect handlers are being shown a growing industrial interest nowadays. Below are listed some of the biggest projects that uses and implement some kind of effect handlers

- GitHub code analysis library Semantic (used in more than 25 million repositories)
- Meta JavaScript UI library React (used in mire than 2 million websites)
- Uber Pyro language
- WebAssembly

TODO : refs ?

3.1 Syntax and usage

We give the Links syntax alongside with usage of effect handlers and operations. All examples in this report are written in Links.

Inside any segment of code, one can invoke an operation `Op` with payload `p` writing **`do`** `Op(p)`. This has to be scoped into a handler of this operation. It will take the payload and the resumption and decide what to do.

```
1  handle ( code ) {
2    case <Op(p) => r> -> ...
3    case x -> ...
4  }
```

Listing 1. Handler

Listing 1 give the syntax for a handler for `Op`, `code` being the code that can perform the operation. When the operation is invoked, the right hand side of line 2 is executed with `p` the payload and `r` the resumption bound in context. When `code` returns a value, line 3 is executed with `x` bound to that value.

If the handler gives the resumption a value `v`, it acts as if **`do`** `Op(p)` returned the value `v`. If the payload has type `t` and `v` has type `t'`, we say that `Op` has type `t => t'`.

The type system keeps track of what effects can be performed. A function from type `a` to `b` which can performs operations `Op1: t1 => t1'`, ..., `Opk:tk => tk'` will have type `(a) {Op1:t1 => t1', ..., Opk:tk => tk'}-> b`.

3.2 Examples

3.2.1 A first exception handler.

```
1  sig assert_positive : (Int) {Fail:() => Zero|_}-> Int
2  fun assert_positive(x) {
3    if (x>0) x else absurd(do Fail)
4  }
```

Listing 2. Fail invocation

This first example shows a function `assert_positive` that takes an integer and returns it only if it positive, else, it raises the effect `Fail` (for details on `absurd` and the `Zero` type, see B.4). This is the simplest example when one operation is invoked with no payload.

```
1  sig maybe : (() {Fail:() => Zero|e}~> a) -> () {Fail{_}|e}~> Maybe(a)
2  fun maybe(m) () {
3    handle(m()) {
4      case <Fail> -> Nothing
5      case x -> Just(x)
6    }
```

```
}
```

Listing 3. maybe handler

This operation can be handled by handler 3. This is actually an exception handler since the resumption is not used in the right hand side (and thus can be omitted in the left hand side). The *effect signature* is `{Fail: ()=> Zero}`, that means there is only one operation called `Fail` that will be handled.

When the operation `Fail` is invoked, the handler directly returns a value, here `Nothing`. When the computation returns a value, the handler will return that value, embedded in the `Maybe` type (on `Maybe` type, see B.5).

The function `maybe` will take a computation that can perform the effect `{Fail: ()=> Zero|e}` ie the operation `Fail` and possibly other effects, captured by the row variable `e`, and returns something of type `a`. It will execute this computation, performing effects from `e` that should be handled by another handler at some point. In the end, it returns a value of type `Maybe(a)`.

The `Fail{_}` in the effect row indicates a polymorphic presence for the operation. Roughly it means that the operation can or cannot be present after. This is needed since the effect type system enforces that all operations in a row are distinct. `Fail` cannot belong to `e` because of the effect row of the input computation.

```
sig h2g2 : () {Fail:() => Zero|_}-> Int
fun h2g2 () {
  assert_positive( -6 * -7 )
}
sig h2g2' : () {Fail:() => Zero|_}-> Int
fun h2g2' () {
  assert_positive( -6 ) * -7
}
```

We can then use the `maybe` function to handle the failures in functions `h2g2` and `h2g2'` :

```
links> maybe ( h2g2 ) () ;
Just(42) : Maybe (Int)
links> maybe ( h2g2' ) () ;
Nothing : Maybe (Int)
```

Listing 4. Links console

3.2.2 Drunk coin tossing. We consider now a new operation `Choose` that can produce a boolean and use it to simulate a drunk coin toss. The drunk tosses the coin and either let it fall or catch it and see heads or tail.

```
typename Toss = [| Heads | Tail |] ;

sig toss : () {Choose:Bool|_}-> Toss
fun toss () {
  if (do Choose) Heads else Tail
}

sig drunkToss : () {Choose:Bool, Fail:Zero|_}-> Toss
fun drunkToss () {
  if (do Choose)
    # catch
    toss ()
  else
    # fall
    absurd(do Fail)
}

sig drunkTosses : (Int) -> Comp([Toss], {Choose:Bool, Fail:Zero|_})
fun drunkTosses (n) () {
  if (n<=0)
    []
  else
    drunkToss() :: drunkTosses (n-1) ()
}
```

Listing 5. Drunk toss

We can define several handlers for this operation.

```
# linear handler : always true
sig h_true : (Comp(a,{Choose:Bool|e})) -> Comp(a,{Choose{_}|e})
fun h_true (m) () {
  handle (m()) {
      case <Choose => r> -> r(true)
  }
}

# non-linear handler : all choices in a list
sig h_all : (Comp(a,{Choose:Bool|e})) -> Comp([a],{Choose{_}|e})
fun h_all (m) () {
  handle (m()) {
```

```
13          case <Choose => r> -> r(true) ++ r(false)
14          case x -> [x]
15      }
16  }
```

Listing 6. Toss handlers

And then we can compose the different handlers in several ways.

```
links> maybe ( h_true ( drunkTosses(2) ) ) () ;
Just([Heads, Heads]) : Maybe ([Toss])
links> h_true ( maybe ( drunkTosses(2) ) ) () ;
Just([Heads, Heads]) : Maybe ([Toss])

links> maybe ( h_all ( drunkTosses(2) ) ) () ;
Nothing : Maybe ([[Toss]])
links> h_all ( maybe ( drunkTosses(2) ) ) () ;
[Just([Heads, Heads]), Just([Heads, Tail]), Nothing, Just([Tail, Heads]),
    Just([Tail, Tail]), Nothing, Nothing] : [Maybe ([Toss])]
```

We try different combinations for 2 coin tosses. In the two first cases, there is no failure so the order of handlers won't matter. The coin is always catch and always on heads.

In the third case, `h_all` imposes that all possibilities are explored so at some point, the coin will fall. This causes a failure, catch by `maybe` that simply returns `Nothing`.

The last case is the most complicated one. The failures are catch by `maybe` *under* `h_all`. That means `h_all` won't see failures and will take each result, whether it is a `Nothing` or a list of heads/tail and form its big list of all possibilities.

3.3 Semantics

We give the operational semantics for handlers.

$$\begin{array}{rcll} \texttt{handle (} v \texttt{) \{ } H \texttt{ \}} & \longrightarrow & N[v/\mathsf{x}] & \\ \texttt{handle (} \mathcal{E}[\texttt{do } Op_i\texttt{(} v \texttt{)}] \texttt{) \{ } H \texttt{ \}} & \longrightarrow & N_i[v/\mathsf{p}, \texttt{fun (x) \{ handle (} \mathcal{E}[\mathsf{x}] \texttt{) \{ } H \texttt{ \} \}}/\mathsf{r}] & \\ & & Op_i \notin \mathcal{E} & \end{array}$$

where $H = \begin{cases} \texttt{case x} \mapsto N \\ \texttt{case } \langle Op_i(\mathsf{p}) \Rightarrow \mathsf{r} \rangle \mapsto N_i \\ \dots \end{cases}$

and $\mathcal{E}$ is an evaluation context : $\mathcal{E} ::= [] \mid \texttt{handle (} \mathcal{E} \texttt{) \{ } H \texttt{ \}} \mid \texttt{var x = } \mathcal{E} \texttt{ ; } N$.

$Op \in \mathcal{E}$ means there is a handler for Op in $\mathcal{E}$.

If there is a value v inside the handler, we call the return case N with x bound to v. When the handler is the most inner one that can handle the operation Op_i invoke with some payload v, what is insured by $Op_i \notin \mathcal{E}$, we bind the payload and build the resumption before calling the case N_i.

The resumption is embedded in the current handler since operation Op_i could be invoked in the rest of the evaluation context $\mathcal{E}$. This is called a *deep* handler.

Shallow handlers. Links also support *shallow* handlers, ie handlers that do not automatically reuse the handler in the resumption. The programmer has to write it and can choose another handler if needed. Below is the new rule for shallow handler.

$$\texttt{shallowhandle (} \mathcal{E}[\texttt{do } Op_i\texttt{(} v \texttt{)}] \texttt{) \{ } H \texttt{ \}} \longrightarrow N_i[v/\mathsf{p}, \texttt{fun (x) \{ } E[\texttt{x}] \texttt{ \}/r}] \quad Op_i \notin \mathcal{E}$$

Parameterised handlers. Another useful feature of handlers is that they could be parameterised. The programmer can give the handler a parameter that is carried along the computation.

```
sig h_state : (a) -> (Comp(b,{ Get:()=>a, Put:(a)=>() | e })) ~> Comp((b,a),
    { Get{_}, Put{_} | e })
fun h_state (init) (m) () {
  handle (m()) (s <- init) {
    case x -> (x, s)
    case <Get => r> -> r (s, s)
    case <Put(s') => r> -> r ((), s')
  }
}
```

Listing 7. Example of parameterised handler

```
fun ex () {
  var x = do Get * 10 ;
  var x = 2*x + do Get ;
  do Put(x) ;
  ()
}

links> h_state (2) (ex) () ;
((), 42) : ((), Int)
```

A second parameter is given to the **handle** construct that binds a variable and the resumption will take an additional parameter.

In the example above, we use that feature to write a state-cell handler. There are two operations `Put` and `Get` that respectively write into the cell and read what is inside. The parameter here is just the content of the cell.

Here are the reduction rules for parameterised handlers

$$\begin{array}{rcll} \texttt{handle (} v \texttt{) (s} \leftarrow w \texttt{) \{ } H \texttt{ \}} & \longrightarrow & N[v/\texttt{x}, w/\texttt{s}] & \\ \quad \texttt{handle (} \mathcal{E}[\texttt{do } Op_i\texttt{(} v \texttt{)}] \texttt{)} & \longrightarrow & N_i[v/\texttt{p}, w/\texttt{s}, & \\ \qquad \texttt{(s} \leftarrow w \texttt{) \{ } H \texttt{ \}} & & \texttt{fun (x, s') \{ handle (} \mathcal{E}[\texttt{x}] \texttt{) (s} \leftarrow \texttt{s') \{ } H \texttt{ \} \}/r}] & \\ & & Op_i \notin \mathcal{E} & \end{array}$$

Remark 3.1. A parameterised has to be deep since the parameter needs a handler to carry it.

Relationships between handlers. Those three kinds of handlers are equivalent in the sense that it is possible to express each of them using one other.

If it is easy to see how to express a deep handler from a parameterised handler or from a shallow handler, the converse is not so obvious. See [HLA20], section 4 or [Hil22], section 6.

4 TYPING

Links is a strongly statically typed language. In this section, we describe some main features of Links type system that allow us to type effect handlers.

4.1 Rows

In order to type effects, we store in function types the effects that are *allowed* to be perform during the execution of that function. To do so we use rows. They are an unordered and labeled data and type structure. They are also used in others features of Links as records and variants.

In the most general setting, a row is defined by

$$R ::= \bullet \mid \rho \mid l : \delta; R$$

where ρ is a row variable, l is a label and δ is a data. $\bullet$ denotes the empty row.

A row is said to be *open* if it ends with a row variable and *closed* if it ends with the empty row.

For actual records, the data are actual values of the language but in the typing setting (typing effects, records and variants), δ will be a presence (see 4.2).

We require also that in one row, labels are all distinct and we consider row equal up to reordering of pairs labels/data.

Example 4.1. $Fail : Zero; Choose : Bool; \bullet$ is a valid row and is equal to $Choose : Bool; Fail : Zero; \bullet$ but $Fail : Zero; Fail : Int; e$ and $Fail : Zero; Fail : Zero; e$ are not valid. In the open row $Choose : Bool; e$, e could not be substituted by anything with label *Choose*.

4.2 Links Type System

We describe in this section and the following Links type system. In order to avoid useless complexity, we focus on the parts of the type system that are useful to understand effect typing. We omit easy and common parts as well as some parts used only for other Links features.

Below is the type and kind syntax.

Kinds	K	::=	$Type \mid Row \mid Presence$	
		$\mid$	κ	kind variable
Rows	R, E	::=	$\bullet$	
		$\mid$	$l : P; R$	
		$\mid$	ρ	row variable
Presence	P	::=	$-$	absent
		$\mid$	T	present with a type
		$\mid$	θ	presence variable
Types	T, A, B	::=	P	primitive type
		$\mid$	$T \to_E T$	function type
		$\mid$	$T \Rightarrow T$	operation type
		$\mid$	(R)	record type
		$\mid$	$[R]$	variant type
		$\mid$	$\forall \alpha :: K.T$	polymorphic type
		$\mid$	$A\ B$	type application
		$\mid$	α	type variable
Primitive types	P	::=	$Int \mid Bool \mid String \mid \ldots$	

We have three different kinds that are usual *types*, *rows* and *presence types*.

Any label that appears into a row is associated with a presence type that indicates if the label is present in this row. If it is, we have to provide a type, otherwise this is not necessary and we just indicate the absence. One may wonder why are presence useful. After all, to know if a label is present into a row, looking at the row should be enough. This is true for closed rows but not for open ones. Writing $l : -; \rho$ implies that the row variable ρ cannot stands for a row containing l since a label cannot appear twice into a row. Absence guaranties that whatever the row variable is substituted with, this label will never occur.

Remark 4.2. In *closed* rows, absence is redundant :

$$l_0 : -; l_1 : P_1; \ldots; l_k : P_k; \bullet = l_1 : P_1; \ldots; l_k : P_k; \bullet$$

Links syntax for presence and effect rows. In Links, we write effect rows between braces `{ ... }`. For a label `A`, the absence is denoted `A-`, the typed presence `A:type` and the polymorphic presence `A{p}`, with `p` a variable of presence kind. The empty row is denoted by `{}`, the closed row $l_1 : \delta_1; \ldots; l_k : \delta_k$ is denoted by `{l1:d1, ..., lk:dk}` and the open row $l_1 : \delta_1; \ldots; l_k : \delta_k; e$ is denoted by `{l1:d1, ..., lk:dk | e}`.

4.3 Typing handlers

The typing judgment has to take into account the current allowed effects. We gather them into a row R and we write $\Delta; \Gamma \vdash e : T/R$ for expression e has type T in context $\Delta; \Gamma$ and can perform effects in R.

Remark 4.3. The context contains two parts Γ and Δ. $\Gamma = x : A; \ldots$ contains typed term variables and $\Delta = \alpha :: K; \ldots$ kinded type variables.

We give below the main typing rules that allow to understand how effect handlers should be typed. The other rules are the obvious ones and can be found in appendix C. For the sake of clarity, shallow and parameterised handlers rules have also been moved to the appendix.

$$\frac{\Delta; \Gamma \vdash p : B/R \qquad R = l : B \Rightarrow A; R'}{\Delta; \Gamma \vdash do\ l(p) : A/R}\ Op$$

$$\frac{\Delta; \Gamma \vdash f : B \to_R A/R \qquad \Delta; \Gamma \vdash t : B/R}{\Delta; \Gamma \vdash ft : A/R}\ App$$

$$\frac{\Delta; \Gamma, x : B \vdash e : A/R}{\Delta; \Gamma \vdash fun(x)\{e\} : B \to_R A/R}\ Fun$$

$$\frac{\Delta; \Gamma \vdash m : B/R' \qquad \Delta; \Gamma, x : B \vdash n : A/R \qquad \begin{array}{c} R' = l_i : B_i \Rightarrow A_i; R_0 \qquad R = l_i : \theta; R_0 \\ \Delta; \Gamma, p_i : B_i, r_i : A_i \to_R A \vdash n_i : A/R \end{array}}{\Delta; \Gamma \vdash handle(m)\{case\ x \mapsto n\ case\langle l_i(p_i) \Rightarrow r_i\rangle \mapsto n_i\} : A/R}\ Handle$$

$$\frac{\Delta; \Gamma \vdash e : A/R \qquad R \subseteq R'}{\Delta; \Gamma \vdash e : A/R'}\ WeakEff$$

where $R \subseteq R'$ is defined by

$$\frac{}{R \subseteq R} \quad \frac{}{R \subseteq \rho} \quad \frac{R \subseteq R'}{R \subseteq l : P; R'} \quad \frac{R \subseteq R'}{l : P; R \subseteq l : P; R'} \quad \frac{R \subseteq R'}{l : \theta; R \subseteq l : P; R'}$$

Rule *Op* states that invoking an operation requires a ambient effect row that contains it with the correct types. Rule *App* ensure that, when applying a function, its effect row is compatible with the ambient one. The *Handle* rule types a **`handle`** construct. The important thing to notice is that the row used to type m obviously contains the operations l_i handled and that after handling, all l_is have been removed from the row. The last rule mentioned here ensures that the ambient

effect row mentions effects that are *allowed* to be performed but not necessary since an effect can be added to the row.

We say that an expression e type-checks only if no unhandled effects are allowed : $\Delta; \Gamma \vdash e : T/\bullet$, noted $\Delta; \Gamma \vdash e : T$.

Theorem 4.4. *If $\Delta; \Gamma \vdash e : T/R$, in the execution of e only effects in R are performed and not handled.*

Corollary 4.5. *If $\Delta; \Gamma \vdash e : T$, in the execution of e all effects are handled.*

See [HL16] for the proof.

4.4 Polymorphism

Links supports a System F [GTL89] style polymorphism. Types can contains kinded variables α that can be abstracted over with a $\forall \alpha :: K$. A polymorphic type can be instantiated with a type application.

For more details on Links polymorphism, one could look at [CCFL19].

5 CONTRIBUTIONS

During my internship I added some new features to Links that have been merged into the 0.0.0 release.

5.1 Effect Aliases

In Links we can define type aliases using the keyword **`typename`**.

```
# sum type via variants
typename BinTree(a) = [| Node(BinTree(a), a, BinTree(a)) | Leaf |] ;
# product type
typename IntPair = (Int, Int) ;
# record type
typename Pet = (name:String, age:Int, species:String, Fur:Bool) ;
# function type
typename Comp(a, e::Row) = () ~e~> a ;
```

Listing 8. Type alias examples

But until now, there was no way to define such thing for effects. This is really usefull since when working with effects, the same set of effects show up everywhere.

Using the new keyword **`effectname`** we can now write an alias for an effect row :

```
effectname MyEffectRow(a, ... ,e::Eff, ...) = { Op1 : type, ... | e } ; #
    open row
```

```
effectname MyEffectRow(a, ... ) = { Op1 : type, ... } ; # closed row
```

To use it in a signature or in another type or effect alias just apply it with the right arguments as for typename things. In arrows, it can be used as row variables:

```
() -MyEffectRow(args)-> ()
```

However, due to lack of kind inference, row variables and aliases have to be used carefully so that Links does not think they are of kind type. We need to write them most of the time between braces `{ | ... }`. For instance, if there is **effectname** `E(a::Eff)= {X : ... | a }` and a row variable `e::Eff`, you will have to write `E({ |e})`, and similarly for another effect alias instead of the variable. This makes the usage of several nested aliases a bit messy, it would be nice if it could be avoided.

Recursive effect aliases are not available for now. The aliases are replaced by the row they correspond to, aliases are not kept are not printed out.

The implementation is mostly inspired by what is done for type aliases.

5.2 Fresh Labels

5.2.1 The pollution problem. TODO examples + cf ref 44 Tes

We first introduce a problem that arises when playing with numerous effects and that we will be calling *effect pollution*. Let us start with an example.

```
sig safe_sqrt : ( Comp(a, {Fail:Zero, Sqrt:(Float)=>Float| e}) ) -> Comp(a ,
    {Fail:Zero, Sqrt{_} | e})
fun safe_sqrt (m) () {
  handle (m())  {
      case <Sqrt(x) => r> -> r (if (x<0.0) absurd(do Fail) else sqrt(x))
      case x -> x
  }
}

sig maybe_sqrt : ( Comp(a, {Fail:Zero, Sqrt:(Float)=>Float| e}) ) -> Comp(
    Maybe(a) , {Fail{_}, Sqrt{_} | e})
fun maybe_sqrt (m) {
  maybe ( safe_sqrt( m ) )
}
```

Let `safe_sqrt` be a handler for the operation `Sqrt` that takes a float and computes its square root. If the number is negative, the handler raises a failure via the `Fail` operation. Let `maybe_sqrt` the composition of the `maybe` handler for failures seen previously and the `safe_sqrt` handler.

```
fun shrink (x) {
```

```
    var y = do Sqrt(x) ;
    if (y < x)
      y
    else
      absurd(do Fail)
  }

  fun end (m) () {
    handle (m()) {
      case x -> [x]
      case <Fail> -> []
    }
  }

  fun f () {
    end ( maybe_sqrt ( fun () { shrink(1.2) +. shrink(0.3) } ) ) ()
  }

  links> f () ;
  [Nothing] : [Maybe(Float)]
```

Let us image that another piece of code describe above uses this `Sqrt` feature to compute its square roots. The function `shrink` takes the square root and then checks whether the number has shrunk. If it is not the case, it raises an exception catch by the handler `end`. What could happen there is that the `Fail` exception used for `Sqrt` can collided with the `Fail` exception used by this piece of code. That is what happen in function `f` : this should have given a empty list since all square roots are correct and one value is not shrunk but the `Fail` from the `shrink` is catch by `maybe_sqrt` that returns `Nothing`.

That kind of effect pollution could become really annoying when one works with a large amount of code and it reduces modularity.

5.2.2 State of the art. Until then, we could find two constructions that tackle this issue in the literature : *lexically scoped handlers* and *effect coercions*.

Effect coercions. Effect coercions, also known as *adaptators* ([CLMM20] section 3), are constructions that change how operations are interpreted and handled.

The simplest example is the *lift* or *mask* coercion that masks the operation from the nearest handler. A masked operation is then handled by the second nearest handler.

```
TODO
```

Listing 9. Mask coercion example

Coercions could also be exchanges, also called *swap*, or effect duplication in some contexts.

It turns out that, if they offer a solution to the pollution problem, they are not so practical. Moreover they seems a bit *ad hoc* and are not really satisfying.

Lexically scoped handlers. [SBMO22] introduces this kind of handlers where one can figure out statically which handler is invoked when. This is achieved by not having operations but computations that take an operation function as parameter.

```
lexhandle ( fun (op) { e } ) with { case <p => r> -> ... }
```

Remark 5.1. In this setting, the handler does not handle "by name". That's why it does only need the payload and not the label of the operation in `case <p => r>`.

Biernacki et al. [BPPS19] use another version of lexically scoped effects with names. Handlers and operations labels are coupled with names that binds them together. Handlers act as binders for theses names.

5.2.3 A fresh idea ? In [EdVP22], de Vilhena and Pottier develop the idea of lexically scoped handlers and propose, as we do, to use fresh effect label generation. Our work has be done in parallel but it turns out that they formalised a system close to what we use and describe there.

Our system has two sort of labels, noted l :

global labels labels as previously used, defined uniquely by their name

local labels labels that are restricted to a scope and are identified by their name together with an identifier $id \in \mathbb{N} \cup \{\bot\}$

$$l ::= Op \mid `Op^{\langle id \rangle}$$

To distinguish local and global labels we use a backtick for local ones. The programmer does not need to find by themselves the identifier. They do not specify any (which is represented by the "free" identifier $\bot$) and fresh unique identifiers are created automatically.

We define a new construct `fresh `A { e }` where `e` is any list of function and/or aliases definitions. This is a binding for local labels. When this is encountered a new unique identifier is generated for `` `A `` and each local occurrence of `` `A `` in the scope `e` is bound to this identifier.

Remark 5.2. When several `fresh` bindings with the same name are nested, a label is bound by the innermost `fresh`.

Going out of scope. A function, value or alias that has a local label in its type cannot escape the scope, but we do want to keep some of the declarations made in that scope. When leaving the scope, for each declaration, we try to *erase* the fresh label l from its type. If it is possible, ie if the label has a polymorphic or absent presence, the function/value/alias declared is added into the context and is available in the rest of the program, its type being cleaned from any local label. Otherwise, the function/value/alias is removed from the context.

We note $A \uparrow_l A'$ if one can erase label l from type A and this will give type A', and $A \downarrow_l$ if it is not possible to erase label l from type A, ie if there exists no A' such that $A \uparrow_l A'$. Below are the most interesting rules that define the erasure of a row type. The rest of the rules can be found in appendix D.

$$\frac{R \uparrow_l R'}{l : -; R \uparrow_l R'} \qquad \frac{R \uparrow_l R'}{l : \theta; R \uparrow_l R'} \qquad \frac{T \uparrow_l T' \qquad R \uparrow_l R' \qquad l \neq l'}{l' : T; R \uparrow_l l' : T'; R'} \qquad \frac{}{l : T; R \downarrow_l}$$

The two first rules state that absent or polymorphic instances of the label can be erased, the third one shows how the erasure propagates into the row and the last one forces the erasure of a present instance to fail.

Theorem 5.3. *For any label l and type T, exactly one of the following holds*

- *either there exists a unique T' such that $T \uparrow_l T'$. we write $e(T) = T'$*
- *or $T \downarrow_l$*

At the end of the scope, for each declaration, we type it and get its type T. If $e(T)$ exists, we replace its type by $e(T)$ in the context. Otherwise, we remove the object from the context. Thus, local labels cannot escape.

5.2.4 Implementation and usage in Links. The possibility to define local labels, ie labels that can only be used in a specific scope, has been added to Links.

The user can define one or more local labels that will be bound in the scope with

```
fresh `A, `B {
  [ scope where labels `A and `B are bound ]
}
```

He can use the local labels so defined in the scope as any other effect label but he cannot use them outside that scope.

Remaining issues. There is one issue remaining. The effect aliases being inlined before typechecking, we can get a label going out of scope. This will be identified by the typechecker and will raise a `local label ... is not bound error`. It seems that the effectname has escaped the scope.

```
fresh `A {
    effectname E = {`A:Int} ;
}
typename T = () -E-> () ;

links>
Type error: The local label `A<1> is not bound
In expression: typename T = () -E-> ().
```

This is not really satisfying but should be fixed by improving effect aliases and giving them the same behavior as type aliases. Furthermore, this could not cause code that should not type-check to type-check since it is catch.

Implementation. Briefly, here is how the code works. All labels in the code can be either global or local, have a name and the local ones have a unique id. Labels with a backtick are parsed into free local label and normal label are parsed into global ones. During desugaring, a newpass binds the local labels. During type-checking, when a `fresh ... { ... }` binding is encountered, we add the labels into the context and typecheck each declaration in the scope. Before that, the context should be clean from any label that will be shadowed by the fresh declaration (labels that have the same name). When leaving the scope, we clean the context from the local labels to avoid them to escape the scope. Moreover we check during typechecking if types do not contain unbound or shadowed labels.

Remark 5.4. For now if all labels are internally of the same type and then can be local or global, the parser only accepts local labels in effect related things. If needed, this could be a way of improvement.

5.3 Polymorphic Operations

My last contribution was to implement the support for polymorphic operations in Links. Until then, only handlers could be polymorphic but not the operations themselves. However this could be useful in a lot of situations. For instance, when writing exceptions, one would want to have a polymorphic `Fail` operation so that it could be used in any location. Without actual polymorphism this could be simulated with a Zero type and its destructor as we've done in previous examples.

This is easily added since polymorphism is already incorporated in Links. The only thing to add is to allow operations to be of the form $\forall \alpha :: K. \ldots$ and not only fat arrow types.

To make an operation polymorphic, the handler should be annotated with a polymorphic type. The type system cannot infer polymorphism and is not able to propagate the polymorphism

information in the signature down to the effect cases. This is something that could be improved in the future.

```
sig catch : (() {Fail:forall a.a |e}~> b) {Fail{_} |e}~> Maybe(b)
fun catch(m) {
  handle(m()) {
    case <Fail => k> : (forall a. () => a) -> Nothing
    case x -> Just(x)
  }
}

sig f : () {Fail:forall a.a}~> Int
fun f () {
  if (do Fail)
    42
  else
    do Fail
}

catch (f)
links> Nothing : Maybe (Int)
```

Listing 10. Polymorphism example

6 CONCLUSION

TODO

7 ACKNOWLEDGEMENT

TODO

REFERENCES

[BPPS19] Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. Binders by day, labels by night: Effect instances via lexically scoped handlers. *Proc. ACM Program. Lang.*, 4(POPL), dec 2019.

[CCFL19] Kwanghoon Choi, James Cheney, Simon Fowler, and Sam Lindley. A polymorphic RPC calculus. *CoRR*, abs/1910.10988, 2019.

[CLMM20] Lukas Convent, Sam Lindley, Conor McBride, and Craig McLaughlin. Doo bee doo bee doo. *Journal of Functional Programming*, 30:e9, 2020.

[CLWY07] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: Web programming without tiers. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *Formal Methods for Components and Objects*, pages 266–296, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[EdVP22] Paulo Emílio de Vilhena and François Pottier. A type system for effect handlers and dynamic labels. 2022.

[GTL89] Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and Types*. Cambridge University Press, USA, 1989.

[Hil22] Daniel Hillerström. *Foundations for programming and implementing effect handlers*. PhD thesis, 2022.

[HL16] Daniel Hillerström and Sam Lindley. Liberating effects with rows and handlers. In *Proceedings of the 1st International Workshop on Type-Driven Development*, TyDe 2016, page 15–27, New York, NY, USA, 2016. Association for Computing Machinery.

[HLA20] Daniel Hillerström, Sam Lindley, and Robert Atkey. Effect handlers via generalised continuations. *Journal of Functional Programming*, 30:e5, 2020.

[Joua] Robin Jourde. Links fork. https://github.com/Orbion-J/links.

[Joub] Robin Jourde. Links pull requests. https://github.com/links-lang/links/issues?q=author%3AOrbion-J+.

[LC12] Sam Lindley and James Cheney. Row-based effect types for database integration. In *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation*, TLDI '12, page 91–102, New York, NY, USA, 2012. Association for Computing Machinery.

[PP09] Gordon Plotkin and Matija Pretnar. Handlers of algebraic effects. In Giuseppe Castagna, editor, *Programming Languages and Systems*, pages 80–94, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[Ré94] Didier Rémy. Type inference for records in a natural extension of ml. *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*, 1994.

[SBMO22] Philipp Schuster, Jonathan Immanuel Brachthäuser, Marius Müller, and Klaus Ostermann. A typed continuation-passing translation for lexical effect handlers. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2022, page 566–579, New York, NY, USA, 2022. Association for Computing Machinery.

[Weba] Web. Links repository. https://github.com/links-lang/links.

[Webb] Web. Links website. https://www.links-lang.org.

APPENDIX

A ARTEFACT

The code that I have produced during this internship is available in my Links fork on GitHub [Joua]. The notable branches are the master, the `fresh-label` branch that contains the work described in section 5.2 and the `polymorphic-operations` branch that contains the work described in section 5.3.

The pull requests that I opened are available here [Joub].

The Links repository can be found at [Weba]. Set up and installation instructions are available on Links website [Webb] or you can built in from source using the Makefile provided in the repository. After having installed the dependencies with `$ opam install --deps-only links`, a simple `$ make` should be sufficient.

To replicate the examples in this paper, use preferably the branches of my fork... TODO WHEN RELEASE

B LINKS TECHNICALITIES

Here are some technical points on Links syntax and semantics that could be useful to the reader.

B.1 Comments

Everything following a `#` in a line is a comment.

B.2 Tuples

A tuple is a closed record with positive integer labels. For instance `(foo, bar)` is the same as `(1=foo, 2=bar)`.

Unit `()` is the empty record.

B.3 Operation type

An operation should always be of type `(a) => b` but one can write `Op:b` which will be desugared into `Op:() => b`.

A polymorphic operation type can also be sugared. One can write `Op:forall a. b` for `Op:forall a. ()=> b`. It is worth noting that this does *not* stand for `Op:() => forall a. b`.

B.4 Empty type

The empty variant type `[||]` is `Zero`. To destroy it, we use the following `absurd` function.

```
1  sig absurd : (Zero) -> a
2  fun absurd(x) {
```

```
    switch ( x ) {}
  }
```

B.5 Option type

Links has a builtin polymorphic option type `Maybe` defined as below. An object of type `Maybe(a)` is either `Nothing` or `Just(x)` where `x` is of type `a`

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

B.6 Function types

Function types are written `(a) {...}-> b`. The domain type is always embraced with parentheses. The braces contains an effect row that tells among others which effects can be performed by the function.

One can write `(a) -e-> b` when `e` is a row variable or an effect type and `(a) -> b` stands for `(a) -_-> b` : the function is then polymorphic in its effects.

`(a) {...}~> b`, `(a) ~e~> b` or `(a) ~> b` is a syntactic sugar that adds the special effect `wild:()` to the row. This is used to denote "wild" code, ie code that cannot be transformed into a SQL query. Recursion, IO or handlers are typically wild. See [LC12] for justifications and details on the use of row typing for database integration.

For more details on row typing, see section 4.1.

B.7 Computation and handlers

Links has a builtin type for computations

```
typename Comp(a, e::Row) = () ~e~> a
```

For better handler composability, we write handlers that take a computation and output another computation. Then we can compose them easily and we just need to launch the computation at the end :

```
# computation
handler''( handler'( handler( f ) ) )
# value
handler''( handler'( handler( f ) ) ) ()
```

C TYPING RULES

We give extra typing rules for shallow and parameterised handlers.

$$\frac{\Delta;\Gamma \vdash m : B/R' \qquad \Delta;\Gamma, x : B \vdash n : A/R \qquad \begin{array}{c} R' = l_i : B_i \Rightarrow A_i; R_0 \qquad R = l_i : \theta; R_0 \\ \Delta;\Gamma, p_i : B_i, r_i : A_i \rightarrow_{R'} A \vdash n_i : A/R \end{array}}{\Delta;\Gamma \vdash shallowhandle(m)\{case\ x \mapsto n\ case\langle l_i(p_i) \Rightarrow r_i\rangle \mapsto n_i\} : A/R}\ ShallowHandle$$

$$\frac{\begin{array}{c} \Delta;\Gamma \vdash m : B/R' \qquad \Delta;\Gamma \vdash w : \gamma/R' \qquad R' = l_i : B_i \Rightarrow A_i; R \qquad R = l_i : \theta; R_0 \\ \Delta;\Gamma, x : B, s : \gamma \vdash n : A/R \qquad \Delta;\Gamma, s : \gamma, p_i : B_i, r_i : (A_i, \gamma) \rightarrow_{R'} A \vdash n_i : A/R \end{array}}{\Delta;\Gamma \vdash handle(m)(s \leftarrow w)\{case\ x \mapsto n\ case\langle l_i(p_i) \Rightarrow r_i\rangle \mapsto n_i\} : A/R}\ ParamHandle$$

D ERASING PROCEDURE

We give the missing rules for the erasure procedure introduced in section 5.2.3.

$$\frac{x = \alpha, P, \rho, \bullet}{x \uparrow_l x} \qquad \frac{A \uparrow_l A' \qquad B \uparrow_l B' \qquad R \uparrow_l R'}{A \rightarrow_R B \uparrow_l A' \rightarrow_{R'} B'}$$

$$\frac{A \uparrow_l A' \qquad B \uparrow_l B'}{A \Rightarrow B \uparrow_l A' \Rightarrow B'} \qquad \frac{A \uparrow_l A' \qquad B \uparrow_l B'}{A\ B \uparrow_l A'\ B'} \qquad \frac{T \uparrow_l T'}{\forall \alpha :: K.T \uparrow_l \forall \alpha :: K.T'}$$

$$\frac{A \downarrow_l}{A \rightarrow_R B \downarrow_l} \qquad \frac{B \downarrow_l}{A \rightarrow_R B \downarrow_l} \qquad \frac{R \downarrow_l}{A \rightarrow_R B \downarrow_l}$$

$$\frac{A \downarrow_l}{A \Rightarrow B \downarrow_l} \qquad \frac{B \downarrow_l}{A \Rightarrow B \downarrow_l} \qquad \frac{A \downarrow_l}{A\ B \downarrow_l} \qquad \frac{B \downarrow_l}{A\ B \downarrow_l} \qquad \frac{T \downarrow_l}{\forall \alpha :: K.T \downarrow_l}$$