

作ってかんたん Algebraic Effects

びしょ〜じょ  @Nymphium

1 はじめに

こんにちは、びしょ〜じょです。みなさんは *Algebraic Effects*（または代数的効果、代数効果など）という言葉聞いたことがありますか。私はある。簡潔に述べるなら、“復帰可能な例外” だが¹⁾、これには少々誤解がある。単に例外発生位置に復帰できるだけでなく、復帰しない、復帰後の処理を複数回おこなう（復帰しないことを 0 回の処理と考えてよい）、といった選択肢の幅がある。また、復帰後の処理をおこなったあとに例外ハンドラの中に戻って別の処理をおこなうことができる。さらに、拡張された型システムを用いることで、例外にはなかった型安全性が得られる²⁾。

まあ、つまるところワンフレーズではおさまらないポテンシャルを秘めていると考えてよい。しかして複雑な言語機能なのだが、今回はそれを小さな言語 λ_{eff} への実装を通して、何が起きているのかを体感してもらう。

λ_{eff} のソースコードはこちら。今回は Haskell でインタプリタを実装する。

 <https://github.com/nymphium/lambdaeff>

実は 4 年前に実装したもののだが、せっかくなので解説しようと思い筆を執った³⁾。

-
- 1) *Algebraic Effects* ではなくドンピシャにこの一言で表せる言語機能といえば、例えば Common Lisp の `restart-case` がある
 - 2) 例外も同様に型システムを拡張するなり *Either* にエンコードするなどで型が付くとか、型安全性ってなんですかなどはここでは割愛する
 - 3) 実際に動かしてみたらバグって動かなかった。4 年も致命的なバグを放置していたわけだ

2 Algebraic Effects in λ_{eff}

本章では λ_{eff} とそれを用いた Algebraic Effects の紹介をする。該当リポジトリで `$ stack exec lambdaeff-ext` を実行すると λ_{eff} のプロンプトが実行される。`rlwrap` コマンドを併用すると親切かもしれない。詳細は本章末で述べるが、 λ_{eff} の文法を簡単に述べると、ラムダ抽象 (`fun x → e`)、let 式 (`let x = e in e'`)、整数と四則演算および Algebraic Effects 関連から成る。

2.1 “復帰可能な例外” から紐解く

おそらく多くの人にとっては、言語の定義を示されるよりも例をいくつか出したほうが直感的に分かるだろう。“復帰可能な例外” というアナロジーを足がかりに、まずは Algebraic Effects を見てみる。

`inst`関数によって新しいエフェクトを生成する。

```
λeff⇒ inst ()
Eff 1
```

`λeff ⇒` はプロンプト。2 行目が結果である。`Eff 1` というのはエフェクトの内部表現である。

`perform eff arg` で `arg` を持ったエフェクト `eff` を発生させる。

```
λeff⇒ let Exn = inst() in perform Exn 0
Abort
```

(エフェクトに対応した) ハンドラがない場合 `Abort` する。

2.1.1 シンプル例外

ハンドラは `handler` 構文、ハンドル処理は `with handle` でこなう。

```
λeff⇒ let Exn = inst() in
      let h = handler Exn (val v -> v) ((x, k) -> (0 - 1)) in
      with h handle 1 + (perform Exn 0)
Int (-1)
```

この処理系は改行を許さない。負の整数リテラルも許さない。縛りがあったほうが楽しいため。

それはともかく、この式の評価を展開すると次のようになる。

```
let Exn = inst () in
let h = handler Exn (val v -> v) ((x, k) -> (0 - 1)) in
with h handle 1 + (perform Exn 0)
(* → ハンドラを展開。便宜上タプルを書いたが言語上は無い *)
(fun v -> v) ((fun (x, k) -> -1)
  ( (* ① *) 0
    , (* ② *) fun y -> 1 + y))
(* → value handlerに適用 *)
(fun v -> v) -1
(* → *)
-1
```

handler Exn (val v -> v) ((x, k) -> (0 - 1)) というのがハンドラの定義になる。Exn エフェクトをハンドルするもので、Exn が発生したら 0 - 1 を返しますというハンドラである。with h handle 1 + (perform Exn 0) は、1 + (perform Exn 0) という式の評価を h でハンドルする、と読める。ところで皆さん継続はご存知ですか？デキるビジネスパーソンなら知っているはずである。エフェクトが発生するとき、① 引数の 0 を x に渡し、② 発生源の perform 式から見た継続 1 + □ をハンドラの k に束縛する。val v -> v の部分は value handler（日本語では値ハンドラなど）と呼ばれるもので、ハンドル処理が終わったあとの値に最終的な変更が加えられる。今は気にしないでおく。

2.1.2 復帰

では、ここで例外を復帰させてみる。ビジパならすでに察しがついているが、継続を使う。

```
λeff⇒ let Exn = inst() in
      let h = handler Exn (val v -> v) ((x, k) -> k x) in
      with h handle 1 + (perform Exn 0)
```

Int 2

ハンドラの定義が変わったことに注目してほしい。先程は `-1` を返していたが、今は `k x` を返す。つまり、`1 + □` の `□` に `x` に束縛した `0` を放り込んでいる。これは即ち例外発生位置からの継続 `1 + □` に復帰している。

`Exn` という名前のエフェクトだが、もはや例外のような動きをしていない。ハンドラによる実装の柔軟性がここに見てとれる。

2.1.3 いじり回す

ここでおもむろに継続を 2 回動かしてみる。

```
λeff⇒ let Twice = inst() in
      let h = handler Twice (val v -> v) ((x, k) -> k (k x)) in
      with h handle
        ( let v = perform Twice 3 in
          2 * v )
```

Int 12

継続に引数を渡した結果を更に継続に渡している。丁寧に紐解くと、`let v = □ in 2 * v` に 3 を放り込んだ結果を追い適用し、`let v = (let v = 3 in 2 * v) in 2 * v`、つまり 12 が返ってくる。継続の使用回数に制限がある言語、ない言語や埋め込みがある⁴⁾。`λeff` は回数無制限なのでランタイムが壊れるまで継続を使ってよい。

`value handler` にも触れてみる。つまるところ `try-catch-finally` である。

```
λeff⇒ let Twice = inst() in
      let h = handler Twice (val v -> v + 1) ((x, k) -> k (k x)) in
      with h handle
        ( let v = perform Twice 3 in
          2 * v )
```

Int 13

4) 回数制限がある理由として、パフォーマンスを考慮する場合や埋め込みに利用する言語機能自体に制限のある場合が挙げられる

ハンドルの処理の最後に value handler の処理がおこなわれるので、先述の通りの 12 に + 1 して 13 となる。

こんなところで雰囲気は掴めたでしょうか。これを使うと何ができるかは、[関連項目の章](#)にて述べる。また、Algebraic Effects が言語プリミティブや埋め込みなどによって多くのユーザが触れるようになってから日が浅く、使い方はまだ手探りなことが多い。本記事を通して Algebraic Effects がどういった動きをするか分かった読者に使い方をどんどん開拓してってもらいたい。

2.2 λ_{eff}

本章の最後に、 λ_{eff} の定義を示しておく。実装時に参照すればよいので、今じっくり読まなくても問題ない。四則演算に関する部分は省略した。

2.2.1 構文

[図 1](#) に構文を示す。

```

 $x, k \in \text{Variables}$ 
 $eff \in \text{Effects}$ 

 $v ::= x \mid h \mid \lambda x. e \mid eff$ 
 $h ::= \text{handler } v \text{ (val } x \rightarrow e) ((x, k) \rightarrow e)$ 
 $e ::= v \mid e \ e \mid \text{let } x = e \text{ in } e$ 
       $\mid \text{inst } () \mid \text{with } v \text{ handle } e$ 
       $\mid \text{perform } e \ e$ 

 $F ::= e \ \square \mid \square \ v \mid \text{let } x = \square \text{ in } e$ 
       $\mid \text{with } v \text{ handle } \square \mid \text{perform } \square \ e \mid \text{perform } v \ \square$ 
 $s ::= [] \mid F :: s$ 

```

図 1 the syntax of λ_{eff}

メタ変数 v および e が値および式を成す。2.1 節で文法は出揃っているので目新し

いところはない。特に h はハンドラの値である。

F 、 s は意味論で用いる。 F は *Frame* の頭文字を取っており、実行時の継続を表す。 s は *stack* の頭文字であり、平たく言えばコールスタックで、 F のリスト。後述するが、 F は評価順序を表している。四則演算は省略したが、例えば和を表すならば $e + \square \mid \square + v$ を足すことになる。

エフェクトを値と同じ名前空間にしたのは微妙だった。OCaml の `local exception` のように、`let effect Eff in` のようにしても良かったかもしれない。

2.2.2 意味論

次に意味論を示していく。まず、スタックを λ_{eff} 上の関数として扱うための補助関数 *flatfn* を図 2 に示す。

$$\begin{aligned} flatfn \ [] &= \lambda x.x \\ flatfn \ (F :: s) &= \lambda x.flatfn \ s \ (F \ [x]) \end{aligned}$$

図 2 utility function *flatfn*

続いて、 λ_{eff} の意味論を小ステップで図 3 に示す。評価したい式 e 、式の評価コンテキスト s およびエフェクト発生位置からの継続を表す評価コンテキスト es の三つ組 $\langle e; s; es \rangle$ の状態遷移によって表される。図 1 の F の形を見ると、関数適用は引数から、他は左から右へ部分項に着目しているのが分かる。つまり F の形が式の評価順序を表している。そして、この評価順序を回していくのが `PUSH` と `POP` 規則である。`THROW` と `HANDLE` で式の評価コンテキストから `with handle` 構文によりエフェクトハンドラを掘り出す。前者では、ハンドル処理でない *Frame* を剥がし継続となる評価コンテキストに乗せている。後者は発生するエフェクトに合致するハンドル処理を掘り当て、 es を *flatfn* で関数に変換し、継続に相当するハンドラの変数に束縛する。ここで注目すると面白いのは、ハンドル処理の *Frame* をもう一度評価コンテキストに積み直している部分だ。この積み直しにより、再度発生しうるエフェクトを同じハンドラで処理することができる。このような動作をするハンドラを “*deep handler*” と呼ぶ。関連項目にて少し言及する。

$\langle F[e]; s; es \rangle \mapsto \langle e; F :: s; es \rangle$	(PUSH)
$\langle v; F :: s; es \rangle \mapsto \langle F[v]; s; es \rangle$	(POP)
$\langle v; []; es \rangle \mapsto \langle v; []; es \rangle$	(RESULT)
$\langle \lambda x.e; (\square v) :: s; es \rangle \mapsto \langle e[x = v]; s; es \rangle$	(APPLY)
$\langle \text{inst } (); s; es \rangle \mapsto \langle \text{eff}; s; es \rangle$	(INSTANCIATE)
$\langle \text{perform } \text{eff} v; F :: s; es \rangle \mapsto \langle \text{perform } \text{eff} v; s; F :: es \rangle$	(THROW)
$\langle \text{perform } \text{eff} v; F :: s; es \rangle \mapsto \langle e_{\text{eff}}[x = v, k = \text{flatfn } es]; F :: s; [] \rangle$ where $F = \text{with } h \text{ handle } \square$	(HANDLE)
$h = \text{handler } \text{eff} (\text{val } x \rightarrow e_v) ((x, k) \rightarrow e_{\text{eff}})$	
$\langle v; F :: s; es \rangle \mapsto \langle e_v[x = v]; s; es \rangle$ where $F = \text{with } h \text{ handle } \square$	(HANDLEV)
$h = \text{handler } \text{eff} (\text{val } x \rightarrow e_v) ((x, k) \rightarrow e_{\text{eff}})$	
$\langle \text{perform } \text{eff} v; []; es \rangle \mapsto \text{abort}$	(ABORT)

図 3 the semantics of λ_{eff}

3 実装

さて、材料が揃ったので実装する。ちなむと、Nix⁵⁾とdirenv⁶⁾を採用しており、それらを使うとGHCのバージョンに対応したhaskell-language-serverが動くのでソースコードが読みやすい。

構文木はListing 1のようにした。

Listing 1 Syntax.hs

```
module Syntax where
```

5) <https://nixos.org/>

6) <https://direnv.net/>

```

type EffectP = Int

data Term
  = Var String
  | Fun String Term
  | Eff EffectP
  | Term :@: Term
  | Perform Term Term
  | Let String Term Term
  | Inst
  | WithH Term Term
  | Handler Term (String, Term) (String, String, Term)
  | Int Int
  | Term :+: Term
  | Term :-: Term
  | Term :*: Term
  | Term :/: Term
  | Abort
  deriving (Show, Eq)

type Stack = [Term -> Term]

```

2.1 節でもちらっと見たように、エフェクトの内部表現は `Int` である。Eq のインスタンスで適当に異なる値を生成できればなんでもよかった。簡単のため、 F の構造は定義せず、評価器に任せた。図 3 に示した `ABORT` を表現するにあたり、また簡単のために状態ではなく値として、`Abort` を追加した。

`flatfn` は Haskell で表現すると `Stack` を `Term -> Term` にする関数なので、やりたいことが比較的分かりやすい。定義はそのまま、初期値を `identity` で fold する (Listing 2)。

Listing 2 flatfn (part of Eval.hs)

```

flatfn :: Stack -> Term -> Term

```



```
flatfn = foldr (.) id
```

リストの先頭側ほど新しい *Frame* なので、右から fold する。

言語の定義にて雰囲気ですませた変数の置換も実装ではちゃんとする必要があるものの、今回は特に面白い部分がないので省略する。

Listing 3 subst (part of Eval.hs)

```
-- | 'subst t x t'' substitutes 'x' in the term 't' to 't''
subst :: Term -> String -> Term -> Term
subst (Var x) x' t
  | x == x' = t
  | otherwise = Var x
.....

substs :: Term -> [(String, Term)] -> Term
substs = foldl (\t' (x, u) -> subst t' x u)
```

値と式が syntactic に分かれていないため、判別関数を用意する (Listing 4)。

Listing 4 valuable (part of Eval.hs)

```
-- | is the term a value?
valuable :: Term -> Bool
valuable = \case
  Var _ -> True
  Handler {} -> True
  Fun {} -> True
  Eff _ -> True
  Int _ -> True
  Abort -> True
  _ -> False
```

さて、いよいよ評価器の実装に取り掛かる (Listing 5)。

Listing 5 eval1 (part of Eval.hs)

```

type Model = (Term, Stack, Stack)

-- small-step evaluation
eval1 :: MonadState EffectP m => Model -> m Model
-- inst
eval1 (Inst, s, es) = do
  i <- get
  let idx' = i + 1
  put idx'
  pure (Eff idx', s, es)

```

eval1 の型に関して、三つ組を Model として定義した。小ステップ意味論で三つ組が遷移していきただけなら Model -> Model でよいのだが、エフェクトの生成でそれぞれ異なるものを作りたいので初期値をインクリメントして渡す適当な実装にした。パフォーマンスなどを厳しく求めないおもしろ実装ならこれでよいのだ。

あとはだいたい定義をそのまま書き写せばよい。

Listing 6 eval1 cont'd (part of Eval.hs)

```

-- pop
eval1 (v, f : s, es) | valuable v = pure (f v, s, es)
-- result
eval1 m@(v, [], _) | valuable v = pure m
-- apply
eval1 (Fun x body :@: v, s, es) | valuable v = pure (subst body x v, s, es)
-- push
eval1 (f :@: e, s, es)
  | valuable f = pure (e, (f :@:) : s, es)
  | otherwise = pure (f, (:@: e) : s, es)

```

ただ、APPLY はちょっとケアする必要がある。*Frame* を Term -> Term という関数で表現しており、パターンマッチングでスタックトップから取り出すことができない。したがって、APPLY 規則に遷移する直前の項を考え、これを分解する。 $(\lambda x.e)v$ が $\lambda x.e$ と $\Box v$ になり (PUSH 規則)、後者がスタックに積まれることで APPLY に遷移できる。

なので、実装ではこれを決め打ちして遷移することなく評価する⁷⁾。

気になるのはエフェクト発生時である (Listing 7)。

Listing 7 eval1 cont'd. (part of Eval.hs)

```
eval1 (pf@(Perform eff e), s, es)
| valuable eff && valuable e = pure $ handleOrThrow eff e s es
| valuable eff = pure (e, Perform eff : s, es)
| otherwise = pure (eff, flip Perform e : s, es)
where
  -- take top of the stack as 'f'
  handleOrThrow eff v s@(f : s') es =
    case f hole of
      -- with (handler eff' _ ((x, k) -> e)) □
      WithH (Handler eff' _ (x, k, e)) hole'
      | eff' == eff && hole == hole' -> do
        let k' = kfun es
            e' = substs e [(x, v), (k, k')]
        -- 'f' remains in 's': it means the handler is *deep*
        (e', s, [])
      | otherwise -> throw
    _ -> throw
  where
    throw = (pf, s', f : es)
  handleOrThrow _ _ [] es = (Abort, s, es)
```

`perform eff v` という項に対し、`THROW` と `HANDLE`、そして `ABORT` の 3 通りでスタックトップの形を気にする必要がある。`handleOrThrow` ではこの 3 種類を捌くため、まずスタックトップを取り出し、`Term -> Term` という Haskell 上の関数に `hole` という特別な (λ_{eff}) 変数を渡し、`Term` にして中身を窺う (Listing 8)。

Listing 8 hole (part of Eval.hs)

```
-- the hole in evaluation context
```

7) なんかないというインタプリタの最適化手法があった気がする。ググって教えてください

```
hole = Var "□"
```

他の変数と被らなければなんでもよい。handleOrThrow の話に戻ると、hole を埋めた結果が with h handle □ かつ h が発生したエフェクトに対応するハンドラ的时候、HANDLE 処理をおこなう。継続を flatfn ではなく kfun という関数で作っているのは、ここでは Haskell 上の関数ではなく λ_{eff} 上の関数が必要なためである (Listing 9)。

Listing 9 kfun (part of Eval.hs)

```
kfun :: Stack -> Term
kfun es = do
  let var = "◇"
  Fun var $ flatfn es (Var var)
```

最後に、eval1 は 1 ステップの遷移 (\mapsto) なので、与えられた式が値になるまで評価するために (\mapsto^*) eval と run を定義する (Listing 10)。

Listing 10 eval (part of Eval.hs)

```
eval :: Term -> Stack -> Stack -> EffectP -> Term
eval t s es = go (t, s, es)
  where
    go mod idx =
      case flip runState idx $ eval1 mod of
        ((v, [], _), _) | valuable v -> v
        (mod', idx') -> go mod' idx'

run :: Term -> Term
run t = eval t [] [] 0
```

色々言ったが、Algebraic Effects を持つ言語のインタプリタは Frame を溜めて継続にすればなんかできるわけだ。実際のところ、他の継続演算子を持つ言語を実装するときと大差はない。実装言語上のデータか対象言語のデータなのかの混乱は、言語実装一般のことなので脳を鍛えるしかない。

パーザは特筆する点がないので解説は省略する。Applicative とパーザコンビネータを常用しているのでなければ、パーザコンビネータよりもパーザジェネレータのほうが保守性が高いと 4 年前に書いたパーザを見て思った（小並感）。


4 関連項目

ここでは λ_{eff} のデザインチョイスに少し触れ、Algebraic Effects 一般の話を広げていく。

4.1 持ち味は何か

2.1 節で見たように、エフェクトそれ自体は意味を持たず、ハンドラによってどういった処理なのかが決まる。これはつまり、副作用を外部から、エフェクトではなくハンドラによって、操作することができる。この抽象化、実装の分離によって、テストバリエーションの向上や依存性の注入など、柔軟な実装が可能となる。また、継続が利用可能なので、強力なコントロール抽象をユーザレベルで定義することができる。

OCaml 5.0 を利用した Algebraic Effects の example 集を見るともう少し雰囲気が出るかもしれない。

 <https://github.com/ocaml-multicore/effects-examples>

並行計算、非同期処理、パイプ、動変数などをユーザレベルで実装する例が示されている。

4.2 Type-and-Effect System

ここまで型の話がなかった。理由は筆者が型の話に弱いから。ところで Algebraic Effects には、Monad と同様に、副作用を（型）安全に扱いたいというモチベーションがある。

2.1 節の Exn エフェクトを例に取る。 $1 + (\text{perform Exn } 0)$ という式があったとき、型は `int` のようになってほしい。一方で最初の例のようにハンドラで継続が破棄されるような場合は？分かりようがないですね。でもエフェクトが発生することは分かる。なので、`int!Exn` のようにマークを付ける。じゃあ $1 + (\text{perform Exn } 0) * (\text{perform Twice } 3)$ はどうするか？ `int!{Exn, Twice}` のように集合として表してみる。このように型に加えてエフェクト情報を不要する体系を “Type-and-Effect system” と呼ぶ。またこのエフェクトの扱いについてはいくつか流

儀があり、当然そこに面白い話がある [1] [4] [5]。

4.3 エフェクトの生成と定義

λ_{eff} はエフェクトを値として持つが、この手法は既存の言語に Algebraic Effects を埋め込むときに取り回しが良い。Kiselyov らは multi-prompt shift/reset を用いた埋め込みで、prompt を内部表現として、エフェクトを値として扱った [3]。Kawahara らは Lua への埋め込みには table を、Ruby への埋め込みには Object を内部表現とした。被らないものを手軽に生成でき、等価性が判別できればなんでもよい。Algebraic Effects をネイティブに持つ言語や、型を用いた埋め込みをする場合は、型による等価性を用いることで、静的にその辺が解決されることになる。

4.4 Shallow and Deep

2.2.2 項で触れたように、ハンドラには deep だの何だのがある。Listing 7 のコメントをよく見ると、

```
.....
-- 'f' remains in 's': it means the handler is *deep*
.....
```

とある。スタックに with handle が残ることで、エフェクトがハンドルされた後の継続の中で起きうるエフェクトも（ついてこれてますか？）、同じハンドラによって捕捉することができる。この深追いするような動きを以て、こういったハンドラを Deep handler と呼ぶ。そうでないものは “shallow handler” と呼ばれる。他の継続演算子を知っているなら、shift/reset と $\text{shift}_0/\text{reset}_0$ の関係と考えればよい。

一瞬「後続の処理にもハンドラついてくるなら deep handler のほうがお得じゃん!」と思うかもしれない。しかし、例えば再帰的にハンドラを定義することによって、deep handler は shallow handler でシミュレーションできる。Hillerström らは [2] にて deep handler では表現できない Unix pipe のエミュレーション、shallow による deep の実装、またその逆を示している。

最後に、我田引水で恐縮だが、筆者のブログでも Algebraic Effects に関する話題を多く取り上げている。こちらは日本語なのでカジュアルに読めると思う。よかったら目を通してほしい。

5 おわりに

本記事では、Haskell を用いて Algebraic Effects を持つ言語を実装してみることで、どういう言語機能なのかを見た。理解不能で使い所がわからないアカデミアやプログラミングをうまぶりたい人のためだけの機能ではなく、汎用性が高く色々なことに使えそうな楽しい機能であることが分かってもらえたら幸いである。OCaml 5.0 にも Algebraic Effects が導入される⁸⁾。その他様々な言語にも埋め込む方法が提案され、Algebraic Effects が使える範囲はあなたの手のすぐ届く場所、またはもう手の中まで来ている。今一度、Algebraic Effects がどういう動きを見せるのか分かったところで、あなたも Algebraic Effects で遊んでみませんか。

References

- [1]Andrej Bauer and Matija Pretnar. “Programming with Algebraic Effects and Handlers”. In: *Journal of Logical and Algebraic Methods in Programming* 84 (Mar. 2012), doi: [10.1016/j.jlamp.2014.02.001](https://doi.org/10.1016/j.jlamp.2014.02.001).
- [2]Daniel Hillerström and Sam Lindley. “Shallow Effect Handlers”. In: *Asian Symposium on Programming Languages and Systems*. Springer. 2018, 415–435.
- [3]Oleg Kiselyov and KC Sivaramakrishnan. “Eff Directly in OCaml”. In: *Electronic Proceedings in Theoretical Computer Science* 285 (Dec. 2018), 23–58. doi: [10.4204/EPTCS.285.2](https://doi.org/10.4204/EPTCS.285.2).
- [4]Daan Leijen. “Type Directed Compilation of Row-Typed Algebraic Effects”. In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. POPL ’17. Paris, France: Association for Computing Machinery, 2017, pp. 486–499. ISBN: 9781450346603. doi: [10.1145/3009837.3009872](https://doi.org/10.1145/3009837.3009872). URL: <https://doi.org/10.1145/3009837.3009872>.

8) 2022 年 4 月にリリース予定だったが大幅に延期され、次のリリース予定日である 9 月 4 日（本記事を赤入れした日！）も milestone の進捗率が 77%だった。頑張ってほしい

- [5]Taro Sekiyama, Takeshi Tsukada, and Atsushi Igarashi. “Signature Restriction for Polymorphic Algebraic Effects”. In: *Proc. ACM Program. Lang.* 4.ICFP (Aug. 2020). DOI: [10.1145/3408999](https://doi.org/10.1145/3408999). URL: <https://doi.org/10.1145/3408999>.