代数的エフェクトを活用するための プログラミングの支援に関する考察

山崎 陽介[1,a)]　川端 英之[1,b)]　弘中 哲夫[1,c)]

概要：代数的エフェクトとそのハンドラは，プログラム中のエフェクトの発生とその処理を分離する言語機構である．この分離により，インターフェースと実装が明確に区別されるため，プログラムの可読性や再利用性が大幅に向上する．近年，OCaml をはじめとする多数のプログラミング言語でこの概念が取り入れられている．しかし，プログラマがその機構を利用するにはいくつかの課題が伴うと考える．具体的には，インターフェースと実装が明確に区別されていることで，かえってコードのフローが直感的で無くなる可能性や，抽象度を上げていくにつれて，ハンドラがネストしエフェクトの流れを追いづらくなる点が挙げられる．本研究では，ALGO Basic という二人用ゲームの実装を通じて，代数的エフェクトの実際の応用時の課題と，それをサポートするための環境についての考察を行う．

キーワード：代数的エフェクト，プログラミング支援，OCaml

1. はじめに

近年, 代数的エフェクトとそのハンドラはOCamlをはじめとした多数のプログラミング言語で導入され注目を集めている．この言語機構は，プログラム中のエフェクトの発生とその処理を分離する言語機構である．

エフェクトとはプログラムがその中で閉じておらず，外部とのやり取りを行う必要のある処理のことを指す．例えば，例外の発生や，I/O を伴う処理，並行処理などである．これらの処理は純粋でないため，ほぼ全ての計算を純粋に行う関数型言語のパラダイムでは扱い方が非常に重要である．ここでの純粋は副作用がないことを指す．エフェクトを扱う処理の代表例としてモナド [8] があり，現在も Haskell をはじめとする多くの言語で用いられている．しかし，モナドは「インターフェースに対してプログラムを書くべきで，実装に対してではない」という根本的なカプセル化原則を破っている [5]．具体的には，モナドは型を提供するがその具体的な実装は外に露出する．代数的エフェクトとそのハンドラはインターフェースと実装を分離しながら，エフェクトを抽象化して扱えるためプログラムの可読性や再利用性が大幅に向上する．

代数的エフェクトとそのハンドラは，優れた抽象化の反面，プログラマがその機構を利用するにはいくつかの課題が伴う．具体的には，インターフェースと実装が分離することによって，かえってコードのフローが直感的でなくなる可能性や，抽象度を上げていくにつれてハンドラがネストしエフェクトの流れを追いづらくなる点などが挙げら

[1] 広島市立大学大学院
[a)] mi66019@e.hiroshima-cu.ac.jp
[b)] kawabata@hiroshima-cu.ac.jp
[c)] hironaka@hiroshima-cu.ac.jp

れる．エフェクトの流れを正しく追うことができない場合，予期せぬバグが発生することがエフェクトを伴うプログラミングでは多くなる．プログラマは，エフェクトを発生させた際にその後どのようなエフェクトがどのような順序で発生するかを確認できることで代数的エフェクトを用いたプログラミングの有用性がさらに増すと考える．本論文では，代数的エフェクトを用いた実装の実例を紹介し，代数的エフェクトの実際の応用時の課題と，それをサポートするための環境についての考察を行う．

本論文の構成は次のとおりである．2 節では，代数的エフェクトハンドラの概要と実用されている環境を述べる．3 節では，代数的エフェクトハンドラを用いた実装例をカードゲーム ALGO Basic[*1] を用いて紹介する．3.4 節では，3 節での実装からみる課題を紹介する．4 節では，それらの課題を解決するための方法の提案と，その実現可能性を述べる．5 節では，関連研究を述べる．最後に 6 節では，まとめと今後の課題を述べる．

2. 代数的エフェクトとそのハンドラ

代数的エフェクトハンドラを持つ言語では，図 1 のようにエフェクトを発生させるとその時点での継続 (その時点での残りの計算)[2] を切り取って，ハンドラへ処理が渡される．ハンドラは，渡された継続を用いて処理を行うことができる．まとめると以下のような処理の流れである．

(1) エフェクトが発生
(2) 対応するハンドラが捕捉
(3) (1) の続きの計算 (継続) から再開

また，継続は再開することができる回数に応じてそれぞれ種類がある．高々一回しか継続を再開できないものをワンショットと呼び，複数回再開できるものをマルチショットと呼ぶ．代数的エフェクトをサポートしている言語でもワンショットに限定しているものも多くある．

代数的エフェクトハンドラは言語機構の一つである例外ハンドラの一般化と見ることができ，例外ハンドラとの違いはエフェクトが発生地点での継続を再開できる点である．代数的エフェクトには，deep と shallow[5] の 2 種類のハンドラがあり，それぞれ等価なことが知られている [4]．しかし，それぞれに特徴があり適切に使用することが望ましい．deep ハンドラと shallow ハンドラはハンドラ内でエフェクトの発生が一回のみの場合は，どちらも同じ結果になるが，複数回起こるときにハンドルの有無が異なる．

ソースコード 1: 代数的エフェクトとそのハンドラの例

```
1  (* effect definition *)
2  type _ Effect.t +=
3            Tick: unit -> int Effect.t
4
5  (* effect perform *)
6  let rec loop x () =
7    if x <= 0 then ()
8    else loop (x - perform (Tick ())) ()
9
10 (* handler definition *)
11 let step f () =
12   match_with f ()
13     { retc = (fun _ -> ());
14       exnc = (fun e -> raise e);
15       effc = (fun (type b) (eff: b t) ->
16         match eff with
17         | Tick () -> Some (fun (k: (b,_)
               continuation) ->
18           Printf.printf "tick\n";
19           continue k 1)
20       )}
```

ソースコード 1 のようにエフェクト (2 行目) は型だけを定義しており，そのエフェクトの具体的な処理はハンドラ (10 から 19 行目) が行っている．このことから，実装とインターフェースが分離されていることがわかる．

2.1 Deep ハンドラ

deep ハンドラは，ハンドラの内側で発生したエフェクトを捕捉する．また，継続の再開後もハンドラの内側で発生したエフェクトを捕捉する．ソースコード 1 のハンドラが deep であった場合を説明する．ソースコード 1 でのハンドラは，エフェクト *Tick* を捕捉 (16 行目) して再開 (18 行目) した

[*1] https://www.sansu-olympic.gr.jp/algo/

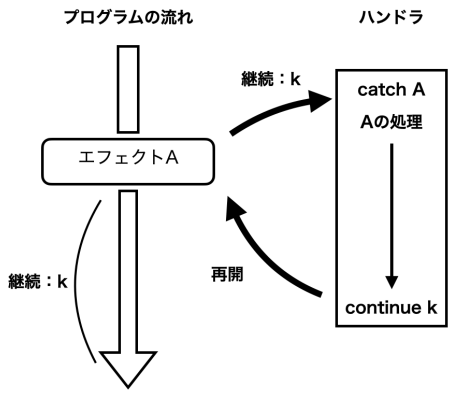

図 1: エフェクトとハンドラの関係

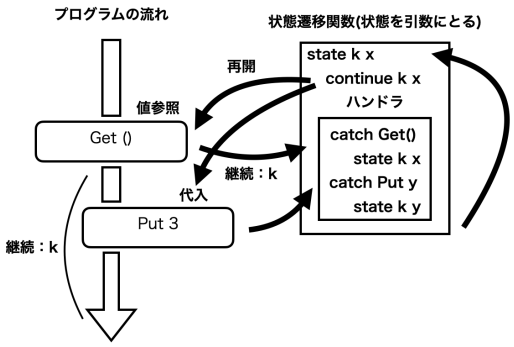

図 2: 代数的エフェクトによる状態管理

後にエフェクト *Tick* が発生したらこのハンドラが捕捉する．そのため，`step (loop 2) ()` と実行すると，tick が 2 回出力されて正常終了する．

2.2 Shallow ハンドラ

shallow ハンドラは，ハンドラの内側で発生したエフェクトを捕捉する．しかし，継続の再開後のエフェクトは捕捉しない．ソースコード 1 のハンドラが shallow であった場合を説明する．このハンドラはエフェクト *Tick* を捕捉 (16 行目) して再開 (18 行目) した後は，どのエフェクトが発生しても捕捉しない．そのため，ソースコード 1 で `step (loop 2) ()` と実行すると，tick が 1 回出力された後，エフェクトがハンドルされずに終了する．

代数的エフェクトで大域変数の参照を純粋に扱う方法を紹介する．純粋に扱うとは，破壊的代入なしで変数の状態を管理することを指す．これには図 2 のように，状態遷移関数 (再帰関数) の内側に shallow ハンドラを定義して実現する．エフェクトとして値を参照する Get と値を代入する Put を用意する．ハンドラは状態遷移関数の引数を変化させて再帰することによって状態を変化させている．このように状態操作の実装を行うハンドラがインターフェースと分離することで，実装がわかりやすくなる利点もある．

2.3 代数的エフェクトが使用されている言語

現在までに，代数的エフェクトが採用されているプログラミング言語が増えている．ここでは，その一部を紹介する．

OCaml： OCaml は昨年バージョン 5.0.0 が公開され，代数的エフェクトがプリミティブとして組み込まれた．また，これによって並行処理がサポートされた．OCaml は継続の利用を一回に限定するようなワンショットの代数的エフェクトをサポートしている．本研究の実装は全て OCaml で行なっている．

Eff[1]： 代数的エフェクトハンドラの研究用に作られた最初の言語である．ML 風の構文で OCaml で実装されている．Eff は継続の利用を一回に限定しないマルチショットの代数的エフェクトをサポートしている．

Koka[7]： JavaScript や.NET にコンパイルされて使用される代数的エフェクトを使うことができる web 指向のプログラミング言語である．

3. 代数的エフェクトを用いた実装例

代数的エフェクトを用いてカードゲーム ALGO Basic を実装した．ALGO Basic は代数的エフェクトを用いることで効率的に実装することができる．ALGO Basic をプロセス間通信を用いて実装するには以下のような要素が必要である．

- カードの状態管理
- 通信の同期
- I/O が絡む処理

- チップの状態管理

これらの処理は代数的エフェクトを用いることで効率的に実装することができる．

3.1 ALGO Basic のルール

ALGO Basic は最大 4 人で遊べるカードゲームであり，獲得したチップの量の多さを競う．基本的なルールは，交互に相手の伏せられたカードに書かれた数字と色を当てていき，当てるとチップが獲得できる．どちらかのカードが全て表になればゲームを終了する．カードを当てる行為をアタックと呼び，当てた場合は成功，当たらないかった場合は失敗という．24 枚のアルゴカードとチップから構成されるゲームであり，カードは 0 ～ 11 までの数字がついた黒と白のカードからなる．最初に各プレイヤにはカードが 4 枚ずつ配られ相手に見せないように横に並べて伏せる．ただし，カードの並べ方には制約があり，左から右に向かって，数字の小さい順に並べる．また，同じ数字の場合は黒の方が小さいとする．この並べ方は．ゲーム中に新しく引いたカードに対しても適用される．

以下がゲームの大まかな流れである．

(1) 山札からカードを一枚引く
(2) 相手のカードの中から当てるカードを決めてそのカードにアタックする
(3) アタックが成功したら，相手はチップを支払いそのカードを表にする．その後もう一度アタックするかアタックをやめて (1) で引いたカードを伏せたまま自分の列に入れるかを選択する．
(4) 相手のカードが全て表になっていたら勝利
(5) アタックが失敗したら，(1) で引いたカードを表にして自分の列に入れる
(6) 相手が (1) を行う

3.2 実装の概要

代数的エフェクトを扱う上での課題を議論するために実装した ALGO Basic の概要について説明する．今回の実装は 2 人プレイに限定している．ハンドラとそれに対応するエフェクトは以下である．

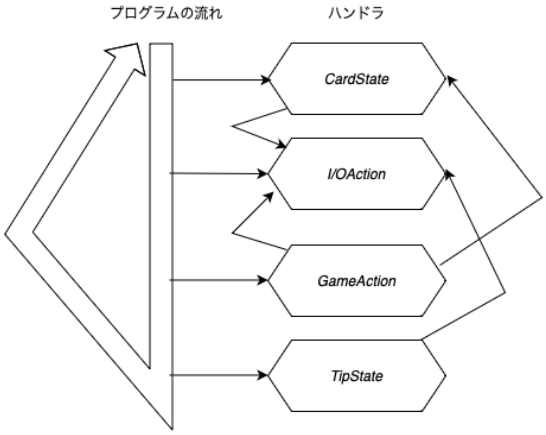

図 3: プログラム全体の構成図

- カードの状態を管理するハンドラ ($CardState$)
  - カードを山札から一枚取得するエフェクト (Get)
  - プレイヤの手札を取得するエフェクト ($Hand$)
  - プレイヤのカードの状態 (表・裏) を変更するエフェクト ($Change$)
  - 最後に取得したカード情報を取得するエフェクト ($Last$)
- I/O 処理を適切に行うハンドラ ($I/OAction$)
  - 入力を受け付けるエフェクト ($Input$)
  - 出力するエフェクト ($Output$)
- ゲームのアクションを管理するハンドラ ($GameAction$)
  - アタックを行うエフェクト (Try)
  - 相手のアクションを待つエフェクト ($Wait$)
  - 勝ち・負け・ゲームを続けるを判定するエフェクト (Now)
- チップを管理するハンドラ ($TipState$)
  - チップを渡すエフェクト ($Serve$)

それぞれのハンドラは $CardState$, $I/OAction$, $TipState$ が shallow ハンドラ，$GameAction$ が deep ハンドラで実装されている．

ALGO Basic のルールで説明したようなゲームに関するアクションを $GameAction$ は行い，その中でカードの状態が変わる (カードが表になるなど) の操作は $CardState$ が行っている．また，それらを各ユーザに伝える処理を $I/OAction$ が行っ

ソースコード 2: Now のエフェクトの発生と処理

```
1  let hand player = perform (Hand player)
2  ...
3  let printbuf player msg = perform (
       Output (player, msg))
4  ...
5  (* handler definition *)
6  let game_action_handlerf v () =
7    | Now player -> Some (fun (k: (b,_)
         continuation) ->
8      let hand = Card_state.hand player in
9      let msg = ... in
10     Io_Action.printbuf player msg;
11     let hand2 = Card_state.hand (
           Card_state.another_player
           player) in
12     let msg2 = ... in
13     Io_Action.printbuf player msg2;
14     let result = Card_state.is_win hand
           in
15     ...
16 (* effect perform *)
17 let ((result, _), player_hand) = perform
        (Now (Card_state.another_player
       player)) in
18 ...
19 (* runner *)
20 let rec run_both a b =
21   ...
22   run_both (fun () -> continue k1
         return_value) (fun () -> continue
         k2 return_value)
23   ...
```

ており，*TipState* はチップの処理に関する動作を行っている．

これらのハンドラの全体像は図 3 のようになっている．ここでの矢印はエフェクトの発生を表している．

3.3 代数的エフェクトを用いた制御の具体的な流れ

3.1 節で述べた ALGO Basic のルールの (4) の処理について説明する．この部分は，勝敗が決まるかゲームを続けるかの判定を行う必要がある．その判定のエフェクト *Now* の実装についてソースコードと共に説明する．*Now* の実装はお互いの手札を取得したのちに，片方のプレイヤの手札が全て表になっているかを確かめるというものである．その際に，お互いの手札の情報を標準出力に出力することも行なっている．

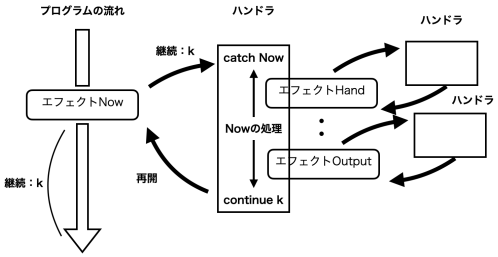

図 4: エフェクト *Now* とハンドラの関係

図 4 のように，エフェクトの流れは以下である．

(1) エフェクト *Now* が発生
(2) プレイヤの手札を取得するためにエフェクト *Hand* が発生
(3) その手札情報を標準出力に出力するエフェクト *Output* が発生
(4) 相手プレイヤの手札を取得するためにエフェクト *Hand* が発生
(5) その手札情報を標準出力に出力するエフェクト *Output* が発生
(6) (2)，(4) をもとに勝敗が決まるかゲームを続けるかの判定を行う

エフェクト *Now* は *GameAction* ハンドラに捕捉されて，そのハンドラ内で，エフェクト *Hand*, *Output* が発生する．

3.4 代数的エフェクトを用いた実装の課題

代数的エフェクトハンドラはエフェクトの実装とインターフェースが分離することで，プログラムの流れがわかりやすくなる一方で，エフェクトの発生が起こった際に再開されるまでのエフェクトの順序を把握しづらいという問題がある．

プログラマは設計段階でエフェクトの発生順序を決めることで，全体像が把握して実装を行うことができる．実装時にエフェクトの発生順序を把握することで，想定通りの動作が行われるかを確認しながら実装できる．3.2 節の具体例では，エフェクトの発生順序は $Now \rightarrow Hand \rightarrow Output \rightarrow Hand \rightarrow$

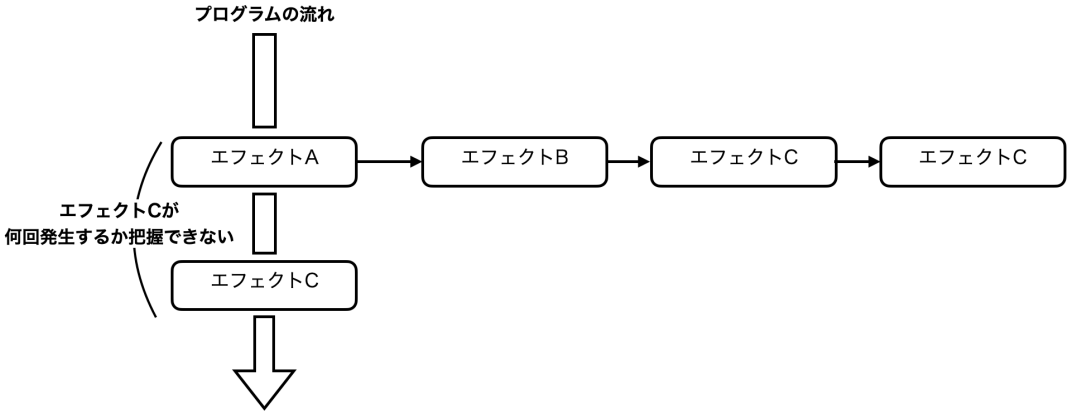

図 5: ハンドラがネストする時

Output の順になっており *Hand* と *Output* が 2 回ずつ現れていることから，それぞれのプレイヤの手札を出力していることがエフェクトの列を見ればおおよそ確認できる．

現在はエフェクト *Now* が発生した地点で，再開するまでにどのようなエフェクトが発生するかを把握することができる言語は存在しない．よって，プログラマはエフェクトの順序が異なるか否かに基づくデバッグができない．

今回の実装では，図 4 のようにハンドラのネストが 3 段以上にはならないためエフェクト *Now* が発生してから再開するまでのでエフェクトは，ソースコード 2 の 7 行目のハンドラの実装部分で全てのエフェクトが発生する．しかし，ネストが深くなるにつれてエフェクトの発生がさまざまな場所で発生するため追いづらくなっていく．例えば，図 5 のようにネストが多段になった場合は，最初のエフェクト A が発生した地点からは，継続が再開されるまでにどのようなエフェクトが発生するかを追うことは難しい．実際に，図 5 のプログラムの流れを見るとエフェクト C が 1 回しか発生しないように見えるが，実際にはエフェクト A の発生によってエフェクト C が 2 回発生している．エフェクト C が副作用を伴うものだった場合に，想定していた動作でなくなっている可能性がある．また，ワンショットの継続ではなくマルチショットの場合には，エフェクトの発生回数がより把握しづらくなる．

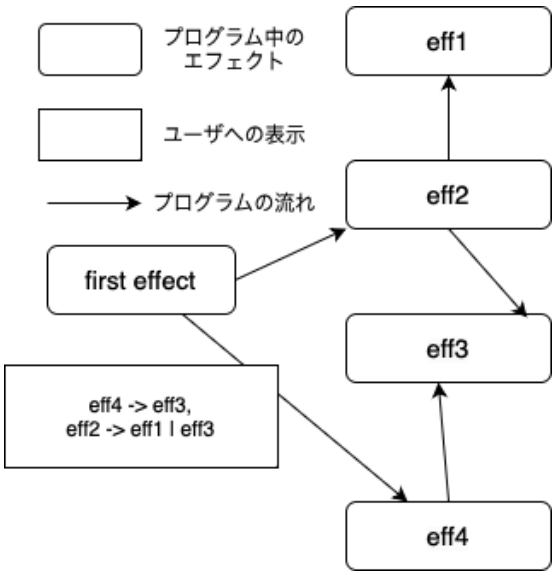

図 6: 提案するツールで提供する情報

純粋な関数は，引数にとる値が等しい場合は，常に同じ値を返すものである．しかし，エフェクトは常に同じとは限らない．エフェクトは，状態を変化させるものをはじめとする副作用を伴うものが多く存在するため，通常の純粋な関数に比べてバグの原因になりやすい．これらの発生順序をプログラマが正しく把握することが，バグのないコードを記述することにつながると考えている．

4. エフェクトの列を正しく把握するためのシステムに関する考察

継続再開までのエフェクトの可視化

我々は，エフェクトの発生からその継続の再開までのエフェクトの系列を静的解析によって抽出し，ユーザに提供することで，プログラマが代数的エフェクトを扱いやすくすることができると考えている．図 6 は我々が提案するツールで提供する情報のイメージ図である．対象となるエフェクト (first effect) を見ると，継続が再開するまでにどのようなエフェクトの列が発生する可能性があるかを表示するものである．この例では，eff4 → eff3 のパターンと eff2 → (eff1 or eff3) のパターンがある．プログラマに図 6 のユーザへの表示を提供することで，プログラマは対象となるエフェクト (first effect) の発生によって，どのようなエフェクトが発生するかを把握することができる．このシステムによって，図 5 のような多段なネストが起こった際にもエフェクトの流れを正確に把握することができる．これはワンショットとマルチショットのいずれでも有益であるが，特にマルチショットの複雑さは大幅に改善する．

我々はこのシステムを VSCode 拡張で提供したいと考えている．従来の LSP を用いた型の表示のようにエフェクト部分にホバー表示でエフェクトの列を表示することができれば非常に便利である．この機能によって，代数的エフェクトがより扱いやすくなり，プログラマの助けになればと考えている．

エフェクトの列の導出

現在の代数的エフェクトに対するユーザ支援は，そのエフェクトの型を LSP から表示させるもの以外には有望なものは我々の知る限りでは存在しない．また，エフェクトが実際にハンドルされるかを保証されることも困難であったが，Vilhena ら [3] がハンドルされるかを保証するようなエフェクトシステムを提案している．しかし，プログラマがそのエフェクトが意図しない挙動 (エフェクトの発生) を行わないことを確かめるためには，そのエフェクトが発生したのちのエフェクトの列情報を把握できることが必要である．

これらを実現するに当たって課題になる点は以下の通りである．

- エフェクトの列を把握するためのエフェクトシステムを導出する必要がある
- deep ハンドラがネストした際の対応
- ハンドルされないエフェクトが存在した場合の対応
- (マルチショットが許されている環境でのエフェクトシステム)

エフェクトの列を把握するためには，そのためのエフェクトシステムが必要である．また，型付けが shallow ハンドラと比べて難しくなる deep ハンドラのネストはより難しくなる [12]．ハンドルされないエフェクトが存在した場合には，再開されることはないため特別な処理が必要である．マルチショットの場合は，継続が複数回再開される可能性があるため難しい．その場合は，エフェクトの発生がハンドラの外側の部分に現れる可能性があるためである．

5. 関連研究

エフェクトの順序を導出することに関連した研究を紹介する．

時間的振る舞いの検証: エフェクトの発生順序を検証する手法としては様々な手法が提案されている [6][9]．ただしこれらは代数的エフェクトハンドラを含まない体系である．また，代数的エフェクトを含む体系として，川俣ら [12] はトレースエフェクトを導出し，shallow ハンドラに対して時間的な振る舞いを検証している．Song ら [11] はホーア論理と項書き換えシステムを拡張し，ワンショットの継続だけでなくマルチショットの継続に対しても時間的な振る舞いを検証している．

型・エフェクトシステム: Vilhena ら [3] は，OCaml の effect system などで用いられている弱い健全性では保証しなかった，エフェクトの発生は必ずハンドルされることを保証するようなエフェクトシステムを提案している．これによって，エフェクトがハンドルされないために停止することを検証することが可能になる．

6. まとめと今後の課題

本研究では，代数的エフェクトを用いた ALGO Basic の実装例から，この機構を扱う難しさとそれを解決するための方法についての考察を行った．ALGO Basic の実装は GitHub[*2]で公開している．また，この実装は [10] の state と message passing を参考に実装している．

今後の課題としては，エフェクトの列を把握するためのエフェクトシステムの導出と，それを用いた支援システムの実装を行いたいと考えている．

参考文献

[1] Bauer, A. and Pretnar, M.: Programming with algebraic effects and handlers, *Journal of logical and algebraic methods in programming*, Vol. 84, No. 1, pp. 108–123 (2015).

[2] Danvy, O. and Filinski, A.: Abstracting control, *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, pp. 151–160 (1990).

[3] de Vilhena, P. E. and Pottier, F.: A type system for effect handlers and dynamic labels, *European Symposium on Programming*, Springer Nature Switzerland Cham, pp. 225–252 (2023).

[4] Hillerström, D. and Lindley, S.: Shallow effect handlers, *Programming Languages and Systems: 16th Asian Symposium, APLAS 2018, Wellington, New Zealand, December 2–6, 2018, Proceedings 16*, Springer, pp. 415–435 (2018).

[5] Kammar, O., Lindley, S. and Oury, N.: Handlers in action, *ACM SIGPLAN Notices*, Vol. 48, No. 9, pp. 145–158 (2013).

[6] Koskinen, E. and Terauchi, T.: Local temporal reasoning, *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pp. 1–10 (2014).

[7] Leijen, D.: Type directed compilation of row-typed algebraic effects, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, pp. 486–499 (2017).

[8] Moggi, E.: *Computational lambda-calculus and monads*, University of Edinburgh, Department of Computer Science, Laboratory for . . . (1988).

[9] Nanjo, Y., Unno, H., Koskinen, E. and Terauchi, T.: A fixpoint logic and dependent effects for temporal property verification, *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, pp. 759–768 (2018).

[10] Sivaramakrishnan, K., Dolan, S., White, L., Kelly, T., Jaffer, S. and Madhavapeddy, A.: Retrofitting effect handlers onto OCaml, *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pp. 206–221 (2021).

[11] Song, Y., Foo, D. and Chin, W.-N.: Automated Temporal Verification for Algebraic Effects, *Asian Symposium on Programming Languages and Systems*, Springer, pp. 88–109 (2022).

[12] 川俣楓河，寺内多智弘：代数的エフェクトハンドラを持つ言語のためのトレースエフェクト，コンピュータソフトウェア， Vol. 40, No. 2, pp. 19–48 (2023).

[*2] `https://github.com/zaki5m/algo_basic`