JavaScript における multi-shot Effect Handler のジェネレータベース実装の試み

武樋 一樹 山崎 徹郎 千葉 滋

Effect handler は代数的効果を扱うための言語機構であり, 副作用の分離と制御フローの柔軟な操作を可能にする. 本研究では, JavaScript に effect handler を導入するトランスパイラを Babel を使って実装する. 従来の CPS 変換による実装と比較して実行速度の向上を図るため, ジェネレータ関数を基盤とする方式を採用する. 特に, while 文と switch 文を組み合わせた制御構造により, 従来のジェネレータを用いる実装では対応していない multi-shot 継続を実現する. これにより, 同一継続の複数回実行が可能となる. さらに, 制御フロー, ネストしたハンドラ, 外部からの継続実行, 複数効果の組み合わせなど, 多岐にわたるエッジケースを網羅する包括的なユニットテストを構築し, 実装の正確性を検証する.

1 はじめに

Effect handler は副作用の分離と柔軟な制御フローを両立させる言語機構として提案され, 例外処理や状態管理から非決定性計算まで多様な効果を統一的に扱える点で注目を集めてきた [15]. 近年は OCaml 5.0 [14] や Koka [12] など主要言語が採用を進めている.

JavaScript で effect handler を実装する既存の手法として, 主に CPS 変換とジェネレータを利用するそれぞれ 2 つのアプローチが提案されてきた. 前者は特に機能の制限はないが, 実行時間が著しく増大する. 後者は CPS 変換を利用するアプローチと比較すると高速だが, multi-shot に対応していないため機能が制限される. 例えばバックトラッキングといったプログラムを実装できない.

本研究は効率の良い multi-shot に対応した effect handler を実装することが目的である. 具体的には, 既にあるジェネレータを利用するアプローチに対して `while` 文と `for` 文を組み合わせた制御構造を加え, multi-shot effect handler を Babel [2] を使って実装する. 実行速度については, 予備実験ではあるものの, CPS 変換と比べて 1.6〜1.7 倍の高速化を確認している.

本論文は次の構成で議論を進める. 第 2 章で既存実装の利点と限界を整理し, 第 3 章で提案手法の設計と実装を詳述する. 第 4 章では予備的な性能評価と包括的ユニットテストの結果を示し, 第 5 章で関連研究を述べる. 最後に第 6 章で結論と今後の課題を述べる.

2 JavaScript における effect handler

Effect handler は代数的効果（algebraic effects）を扱うための言語機構であり, 副作用の分離と制御フローの柔軟な操作を可能にする. 効果の発生を表す `perform` と, 効果を処理するハンドラ, 効果の発生元に処理を戻す `resume` がある. この機構により, 例えば `try` 節中の `perform` の後続処理を, 異なる値で複数回実行することができる.

以下に, 非決定的選択を用いて複数の値から選択し, それぞれを 3 倍した結果をすべて収集する effect handler プログラムの例を示す. このプログラムは `[3, 6]` を返す：

このプログラムの実行フローを詳細に説明する. まず 1 行目から 4 行目までの `try` 節に入り, 2 行

A Generator-based Implementation of Multi-shot Effect Handlers in JavaScript

Kazuki Takehi, Tetsuro Yamazaki, Shigeru Chiba, 東京大学大学院情報理工学系研究科, Graduate School of Information Science and Technology, The University of Tokyo.

```
1  try {
2    let choice = perform({type: "choose",
         opt1: 1, opt2: 2});
3    // ここで制御がchoose ハンドラへ
4    return choice * 3;  // choice は 1 回目
         は 1, 2 回目は 2
5  } handle "choose" with {
6    let r1 = resume(arg.opt1);
7    // 継続を値 1 で実行, 結果 [3] を取得
8    let r2 = resume(arg.opt2);  // 継続を値
         2 で実行, 結果 [6] を取得
9    return r1.concat(r2);
10   // [3, 6] を返す
11 } handle "return" with {
12   return [arg.returnValue];  // 返り値を配
         列でラップ
13 }
```

図 1　非決定的選択の effect handler プログラム例

目で perform 式が実行される．perform は type: "choose"で指定された choose ハンドラのインスタンスを呼び出し，制御をハンドラ（4-7 行目）に移す．このとき，2 行目の perform 式以降の 3 行目の return 文までの継続（continuation）がハンドラに渡される．

4 行目で choose ハンドラが開始され，5 行目で resume(arg.opt1) が呼び出される．arg には perform に渡されたオブジェクト全体が格納されており，arg.opt1 は 1 となる．resume は渡された継続を引数 1 で実行する．これは perform 式の値となる．これにより 2 行目の choice に 1 が代入され，3 行目で choice * 3（つまり 3）を try 節の実行結果として返し，try 節が終了する．

try 節が終了すると，return ハンドラ（11-12 行目）が呼び出される．return ハンドラは try 節で返した値である 3 を配列 [3] として返す．return ハンドラが値を返すと，それを実行結果として 5 行目の継続の実行が終わる．配列 [3] は r1 に代入され，choose ハンドラの続きが実行される．

次に，8 行目で resume(arg.opt2) が実行される．すると，前回同様 resume は渡された継続を実行し，2 行目で choice に 2 が代入される．3 行目で 6 を返し，try 節が再び終了する．すると，return ハンドラ内部で今度は [6] が返される．これで，8 行目の resume(arg.opt2) による継続の実行が終了し 8 行目の r2 に [6] が格納される．

最後に 7 行目で r1.concat(r2) により [3, 6] が返され，1 行目から 13 行目までの try-handle 全体の実行結果となる．このように，同じ継続（2-3 行目）が異なる値（1 と 2）で 2 回実行されている．

この例では，choose ハンドラから resume が 2 回呼び出された．このように，あるハンドラから resume を 2 回以上呼ぶことで，同一の継続を異なる引数で複数回実行できる effect handler を，以降は multi-shot 対応の effect handler，あるいは単に multi-shot effect handler と呼ぶ．逆に最大 1 回しか継続を実行できない effect handler を，single-shot effect handler と呼ぶ．multi-shot 対応の effect handler では，非決定性計算や全探索アルゴリズムを簡潔に記述できる．

JavaScript で effect handler を実装する主要なアプローチとして，CPS 変換を用いた実装とジェネレータを用いた実装が提案されてきた．CPS 変換による実装では，すべての関数が継続を引数として受け取るスタイルに変換される．継続はファーストクラスの関数オブジェクトとして表現されるため，同じ継続を複数回呼び出すことで multi-shot effect handler を実現できる．しかしながら，CPS 変換による実装には実行効率の面で課題がある．例えば関数呼び出しのたびに，その継続が独立した関数に変換される．その関数の生成，呼び出しの実行時オーバーヘッドは無視できない．

ジェネレータを用いた実装は，CPS 変換による実装と比較して高い実行性能を示す傾向にある．しかしジェネレータの性質上，一度進めた実行状態を巻き戻すことができないため，同じ継続を 2 回以上呼び出すことが一般に困難である．そのため，既存のジェネレータを用いた実装では single-shot しか実現されてこなかった．

3　Multi-shot effect handler のジェネレータによる実装

本論文では，ジェネレータと while-switch 制御構造を組み合わせた multi-shot effect handler の実装手法を提案する．本手法の核心は，ジェネレータの中

断・再開機能を活用しつつ，ラベル付き while 文により同一継続の複数回実行を実現する点にある．以下基本変換規則と，その multi-shot への対応，関数呼び出しや条件分岐の変換について説明する．

3.1 基本変換規則

提案する実装手法では，try 節と各 handle 節を別々のコルーチンとし，ランタイムシステムと呼ぶ関数によって実行を再開するコルーチンを適切に切り替えることで，try/handle, perform, resume の機能を実現する．このコルーチンの実装にジェネレータ関数を用いる．

ジェネレータ関数とは，呼び出すとジェネレータというオブジェクトを返す関数である．ジェネレータは .next(value) メソッドを呼び出すと，初めての呼び出し時は関数の先頭から，2 回目以降は前回 yield で停止した位置から，実行を再開する．

引数 value は直前の yield 式の評価結果としてジェネレータ内部に渡され，次の yield まで処理が進み，その時点で値を呼び出し側に返して一時停止する．このようにジェネレータと呼び出し側の間では，値と制御の双方向の受け渡しが可能である．ゆえに，perform や resume や return などのように try 節や handle 節を値と共に行き来するものは，yield に置き換えて実装することができる．ここで行き来する際，適切な try 節や handle 節を見つけて実行するのがランタイムシステムである．

.next の引数 value は直前の yield の実行結果としてジェネレータに渡される．再開したジェネレータは次の yield まで実行を進めて再び一時停止し，yield の引数が .next の戻り値として返される．コルーチンはこの仕組みを利用すれば実装できる．コルーチン化したい try 節と handle 節をジェネレータ関数とし，.next を呼ぶ側の関数が適切なジェネレータを選んで .next を呼ぶようにすればよい．選ばれたジェネレータに対応するコルーチンの実行が再開される．本論文では以下，このジェネレータを選んで .next を呼ぶ側の関数のことをランタイムシステムと呼ぶ．この実装では，perform や resume, return のように try 節と handle 節の間を行き来するものは，yield によって実現される．

```
1  try {
2    // 処理 A
3    let result1 = perform({type: "T1"});
4    // 処理 B
5  } handle "T1" with {
6    // 処理 C
7    resume(arg1);
8    // 処理 D
9    resume(arg2);
10   // 処理 E
11 } handle "T2" with {
12   // 処理 F
13 }
```

図 2　try/perform/handle の基本例

図 2 のような単純なプログラム例と，その変換後のプログラムである図 3 を用いて，具体的な変換方法を説明する．ただし，図 2 と図 3 における「処理 A」や「処理 B」などは，プログラムの任意の処理を表す抽象的な記述である．全体のプログラムはまず，main というジェネレータ関数になっている．また，try 節は 2-21 行目の_tryGen というジェネレータ関数に置き換わっている．handle 節は T1 と T2 の 2 つあり，それぞれ別々のジェネレータ関数に置き換わる．一つ目は 28-43 行目，2 つ目は 47-50 行目のジェネレータ関数である．プログラム全体は，56 行目のランタイムシステム, runGenerator の呼び出しによって実行が始まる．ランタイムシステムの実装は，図 4 で示す．

変換後の各ジェネレータは，yield の際に，その実行状態を表すオブジェクトを返す．それぞれ，RUN_TRYGEN, PERFORM, RESUME, TRY_DONE の 4 つである．ランタイムシステムは，それらのオブジェクトを受け取って対応する処理を行う．まず，ランタイムシステムはプログラム全体を表すジェネレータを受け取るので，図 4 の 2 行目で next を呼んで，図 3 の 23 行目の最初の yield まで実行を進める．

図 3 の 23 行目の yield は，RUN_TRYGEN オブジェクトを返すので，ランタイムシステムは図 4 の 6 行目から 10 行目までの対応する処理を実行する．ここで渡されたハンドラを登録し，さらに try 節のジェネ

```
function* main() {
  function* _tryGen() {
    // 処理 A
    let {RESUME_ARG: result} = yield PERFORM({
      type: "T1",
      label: "l1",
      arg: null,
    });
    l1: while (true) {
      // 処理 B
      // try ブロックの実行が完了したことをランタイムに通知
      const { LABEL, RESUME_ARG } =
        yield TRY_DONE();
      switch (LABEL) {
        case "l1":
          result = RESUME_ARG;
          continue l1;
        // 他の perform がある場合は追加の case 節
      }
    }
  }

  yield RUN_TRYGEN({
    tryGenerator: _tryGen(),
    handlers: [
      {
        handlerType: "T1",
        handlerFunc: function* () {
          // 処理 C
          let value1 = yield RESUME({
            arg: arg1,
          });
          // 処理 D
          let value2 = yield RESUME({
            arg: arg2,
          });
          // 処理 E
          const { WHICH_LABEL, RESUME_ARG } =
            yield TRY_DONE();
          switch (WHICH_LABEL) {
            // perform がある場合は case 節
          }
        }
      },
      {
        handlerType: "T2",
        handlerFunc: function* () {
          // 処理 F
          // TRY_DONE + switch 文
        }
      }
    ]
  });
}
// ランタイムシステムの呼び出し
runGenerator(main(), null);
```

図 3　変換後のジェネレータベース実装

```
function runGenerator(generator, arg) {
  let {value} = generator.next(arg);

  switch (value.MESSAGE) {
    // try ブロックの実行を開始する
    case "RUN_TRYGEN":
      let {tryGenerator, handlers} = value;
      tryGenerator.PARENT = generator;
      SETHANDLERS(tryGenerator, handlers);
      return runGenerator(tryGenerator, null);

    // perform 式が実行されたときの処理
    case "PERFORM":
      let handler = LOOKUP_HANDLER(
        value.type, generator
      );
      let handlerGen = handler.handlerFunc()
      handlerGen.PERFORMER = generator;
      handlerGen.label = value.label
      return runGenerator(
        handlerGen, value.arg);

  // resume 式が実行されたときの処理
    case "RESUME":
      let handler = generator;
      return runGenerator(
        handler.PERFORMER, {
          WHICH_LABEL: handler.label,
          RESUME_ARG: value.arg});

    // try ブロックが完了したときの処理
    case "TRY_DONE":
      let lastHandler =
        LOOKUP_LAST_RESUMED(generator);
      return runGenerator(lastHandler, value.arg);

    // 関数が終了したときの処理
    case "FUNC_DONE":
      return runGenerator(generator.PARENT, {
        WHICH_LABEL: generator.label,
        RESUME_ARG: value});

    // 関数呼び出しの処理
    case "RUN_FUNC":
      value.generator.PARENT = generator;
      return runGenerator(value.generator, null);
  }
}
```

図 4　ランタイムシステムの実装

レータを，runGenerator を使って再帰的に呼び出す．

図 2 で try 節の中で perform が実行されたら，図 3 の 4 行目で PERFORM オブジェクトを yield する．図 4 の 13 行目から 22 行目で，適切なハンドラを見つけて handler.handlerFunc() により新しいハンドラのジェネレータを生成し，再び runGenerator を使って再帰的に呼び出す．図 3 の例では，28 行目から 43 行目のジェネレータ関数から生成されたジェネレータが呼ばれる．

ハンドラが resume を実行すると，図 3 の 30 行目のように RESUME オブジェクトが yield で返される．図 4 では，perform を実行した際に，try 節に対応するジェネレータが PERFORMER に記録してあるので，それを引数に runGenerator を使って再帰的に呼び出す．これにより，中断していた try 節の実行が再開される．またこの時，図 3 の 4 行目の yield の戻り値として RESUME の引数である図 3 の 31 行目の arg1 が渡され，結果 result に代入される

次に，try 節の最後までたどり着いたら図 3 の 13 行目で TRY_DONE オブジェクトを yield で返す．ランタイムシステムは，まだ終わっていないハンドラを探して runGenerator を使って実行を再開する．終わっていないハンドラが無い場合，try 節のジェネレータを呼び出した側であるジェネレータの実行を，runGenerator を使って再開する．各ハンドラの最後でも TRY_DONE オブジェクトを yield で返し，同様の処理を行う．

3.2 multi-shot への対応

multi-shot へ対応するためには，一度実行を再開した yield に再び戻って実行を再開できなければならないが，そのような機能はジェネレータにはないためできない．そこで，perform に対応する yield 以降の処理は while 文で囲み，繰り返し実行できるようにする．yield が入れ子になっている場合に備えて，while 文のボディの最後で TRY_DONE を yield で返した後，switch 文を使って適切な while 文の先頭に continue 文で戻るようにする．

図 3 の例では 9 行目から 20 行目までの while 文が multi-shot 対応のための while 文である．図 3 の 13 行目の TRY_DONE を返す yield で一時停止した後，このジェネレータが再び .next で再開された場合は，17 行目の continue 文で 9 行目に戻り，再び処理 B から実行する．図 2 の 7 行目の resume の後，9 行目の 2 回目の resume を実行すると，そのような再開がおこる．前者の resume に対応するのは図 3 の 30 行目の yield である．

T1 ハンドラに対応するジェネレータが，この yield による一時停止から再開するときは，try 節に対応するジェネレータは図 3 の 12 行目の yield で一時停止している．T1 ハンドラに対応するジェネレータが実行再開後，2 回目の resume に対応する図 3 の 34 行目の yield で一時停止すると，ランタイムシステムにより try 節に対応するジェネレータが 12 行目の一時停止から再開される．continue 文で 9 行目に戻るので，perform 直後の処理 B が再度実行される．

while 文中の switch 文は try 節中で perform が複数回実行される場合に，適切な perform からの継続を実行するためのものである．図 5 の例を用いて詳しい変換方法を説明する．図 5 は，try 節内で n 回 perform を実行するプログラムであり，図 6 がその変換後のプログラムである．ただし，図 5 における「処理 Ai」は，関数呼び出しや perform がないような，プログラムの任意の処理を表す抽象的な記述である．try 節は，_tryGen というジェネレータ関数に置き換わっている．なお，_tryGen にハンドラを紐づけた上で実行する RUN_TRYGEN の部分は 3.1 節と同様であるので省略する．

まず，図 5 で n 個のうち 1 つ目の perform が実行されたら，図 6 の 3 行目で PERFORM オブジェクトを yield する．ここで，どのハンドラを呼び出すかを指定する type と，それを呼び出した perform を表す label の，2 つのプロパティを渡す．図 4 の 14 行目から 15 行目でランタイムシステムは，適切なハンドラのジェネレータを新しく生成し，runGenerator を使って再帰的に呼び出す．

try 節で n 回全ての perform を実行した後は，図 6 の 22 行目にたどり着き，TRY_DONE オブジェクトを yield したとする．すると図 4 の 34 行目でランタイムシステムは，最後に resume したハンドラに対応す

るジェネレータを探し出し，runGenerator を使って再帰的に呼び出す．これは図 5 の 7 行目の perform により実行されているハンドラに対応するジェネレータで，図 6 の 15 行目の yield で一時停止している．これを呼び出すと，図 5 の 11 行目にあるように，ハンドラはそのジェネレータにおいて二回目の resume を実行するので，図 3 の 34 行目から 36 行目と同様にして，arg2 を含んだ RESUME オブジェクトが yield でランタイムシステムに渡される．

続いて，図 4 の 24 行目から 29 行目で yield の戻り値として RESUME オブジェクトを通じて渡された arg2 の値と label が渡される．この label は 2 回目の resume を実行したハンドラを呼び出した perform に付加された label である．これは図 5 の 7 行目の n 回目の perform であるので図 6 の 22 行目の yield に渡される ln である．再開される try 節に対応するジェネレータは図 6 の 22 行目の yield で一時停止しているので，再開すると続く switch 文を実行する．戻り値として返される label は ln であるので，33 行目の continue で 19 行目に戻り，処理 $A_(n+1)$ が実行される．

このように perform が自身を表す label をランタイムシステムに渡すことにより，resume により継続を実行するときに，try 節のどこから先の継続を実行すればよいかを制御している．図 5 の n 回目以外の perform によって呼ばれたハンドラが 2 回目の resume を実行した際も，上記と同様の仕組みによって try 節の正しい位置から継続を実行できる．

3.3 関数呼び出しの変換

関数定義と関数呼び出しはハンドラおよび perform の実装と同様である．すべての関数はジェネレータ関数として定義する．関数呼び出しは yield に置き換え，RUN_FUNC オブジェクトを引数に渡す．RUN_FUNC オブジェクトには，呼び出される関数から生成したジェネレータや，その関数呼び出しの位置を label として含める．関数呼び出し以降の継続は，perform の場合と同様，対応する label 付き while 文で囲む．while 文のボディの最後は yield 終わり，continue で適切な while 文の先頭に戻る．関数定義のボディの

```
1  try {
2    // n 回 perform するような処理
3    // 処理 A1;
4    let result_1 = perform({type: "T1" });
5    // ...
6    // 処理 An
7    let result_n = perform({type: "T1" });
8    // 処理An+1
9  } handle "T1" with {
10   resume(arg1);
11   resume(arg2);
12 }
```

図 5　try/perform/handle の基本例

中の return 文は，yield に置き換え，FUNC_DONE オブジェクトを渡す．具体的な変換例を図 7 と 8 に示す．

4 実験

4.1 実行速度

本論文で提案する実装と，CPS 変換に基づく実装（以下，CPS 版と呼ぶ）の性能を比較するために，予備的な実験を行う．具体的には，ループ内で perform を n 回呼び出すシンプルなベンチマークを使用する．測定環境は，CPU: Apple M1, OS: macOS 15.3.1, Node.js: v21.7.3 である．ベンチマークコードは図 9 の通りである．

表 1 に，ループ回数 n を変化させた際の実行時間を示す．まず，小規模な処理（n = 10〜25）では両実装の性能差は小さく，ほぼ同等の実行時間を示している．しかし処理量が増加するにつれて性能差が拡大し，n = 100 以上では本論文の手法が CPS を用いる手法より約 1.6〜1.7 倍高速に動作することが確認される．

4.2 包括的なユニットテスト

本論文で提案する実装の正しさを検証するため，unit001 から unit100 までの 100 個のユニットテストを作成した．本論文で提案する実装は作成したこれらのユニットテストをすべてパスする．ユニットテストの作成にあたっては，まず大規模言語モデルを用いてテストプログラムとその期待出力のペアを生成し，それらを手動で検証・修正した．具体的にはまず，本論文で示す effect handler の仕様書を読み込ませた

表 1　perform 呼び出し回数による実行時間の比較（ミリ秒）

実装方式	n=10	n=25	n=50	n=100	n=150
ジェネレータ版	0.785	1.072	1.113	1.530	1.917
CPS 版	0.790	0.962	1.192	2.439	3.217
性能比	1.0 倍	0.9 倍	1.1 倍	1.6 倍	1.7 倍

```
function* _tryGen() {
  // 処理 A1
  let {RESUME_ARG: result1} = yield
      PERFORM({
    type: "T1",
    label: "l1"
  });
  l1: while (true) {
    // 処理 A2
    let {RESUME_ARG: result_2} = yield
        PERFORM({
      type: "T1",
      label: "l2"
    });
    l2: while (true) {
      ...
      let {RESUME_ARG: result_n} =
          yield PERFORM({
        type: "T1",
        label: "ln"
      });
      ln: while (true) {
        // 処理 An+1
        const { WHICH_LABEL, RESUME_ARG
            } =
          yield TRY_DONE();
        switch (WHICH_LABEL) {
          case "l1":
            result_1 = RESUME_ARG;
            continue l1;
          case "l2":
            result_2 = RESUME_ARG;
            continue l2;
          ...
          case "ln":
            result_n = RESUME_ARG;
            continue ln;
        }
      }
    }
  }
}
```

図 6　変換後のジェネレータベース実装

```
function foo() {
  // 処理A
  return B;
}
// 処理 C
let result = foo();
// 処理D
```

図 7　関数定義と呼び出しの例

```
function* foo() {
  // 処理 A

  let {WHICH_LABEL, RESUME_ARG} =
    yield FUNC_DONE(C);
  switch (WHICH_LABEL) {
    // 処理 A に perform や関数呼び出しが
    // ある場合は，それらに対応した case 節
  }
}
// 処理 C
let result = yield RUN_FUNC({
  generator: foo(),
  label: "L1"
});
L1: while (true) {
  // 処理 D
}
```

図 8　変換後の関数定義と呼び出し

上で, 以下のようなプロンプトで作成させた.

> 作成した effect handler の仕様書を参考に, 本実装 をテストする 100 個のユニットテストを作成してください. 以下のカテゴリで作成してください：
>
> - unit001-042: 基本エフェクト（State, Exception, Reader, Writer, Nondeterminism, Asynchronous）とその組み合わせ

```
try {
  for (let i = 0; i < n; i++) {
    perform({type: "a"});
  }
} handle "a" with {
  resume();
}
```

図 9　性能測定用ベンチマークコード

```
let globalResume = null;

try {
  console.log("Handler called");
  let value = perform({type: "store"});
  console.log("Resumed from global:",
      value);
} handle "store" with {
  globalResume = resume;
  console.log("Global resume exists");
}

// ハンドラ外部から resume を呼び出す
globalResume("hello");
```

図 10　resume をグローバル変数に格納するテスト（unit063）

- unit043-062: 各制御フロー（`if`, `for`, `while`, `try-catch` 等）と multi-shot effect handler の組み合わせ
- unit063-100: エッジケース（グローバル変数への resume 保存, ネストしたハンドラ, resume を呼ばないケース, try 節内での resume 呼び出し等）

各テストは, テストコードと期待される出力のペアで構成してください.

例えば図 10 は, `resume` をグローバル変数に格納して後で実行するテスト（unit063）である. プログラムの出力は以下.

```
Handler called
Global resume exists
Resumed from global: hello
```

このプログラムでは, まず `try` 節に入り「Handler called」が出力される（4 行目）. 次に `perform` が実行されて store ハンドラに制御が移り, `resume` 関数が globalResume に格納されて「Global resume exists」が出力される（8-9 行目）. ハンドラ内で `resume` は呼び出されないので, try-handle 構文全体の実行も完了する. その後, グローバル変数 `globalResume` に格納された `resume` を引数 `"hello"` で呼び出すと（13 行目）, 中断されていた `try` 節の継続が再開され, `perform` の値として `"hello"` が返されて「`Resumed from global: hello`」が出力される（6 行目）.

5　関連研究

言語レベルで effect handler が実装されている代表例として, Koka, Eff[3], Effekt などがある. これらの言語では, effect handler が第一級の言語機能として設計されており, 基本的に multi-shot をサポートしている. ライブラリとして effect handler を実装する試みは, JavaScript, C, Java, Scala, Haskell など多くの言語で行われている. JavaScript では effectsjs[8] , algebraic-effects-ts[16] などが実装されており, C では libhandler[11] , libmpeff[13] , libseff[1] が提案されている. Java では Java-Effekt[6] , Haskell では extensible-effects[10] , freer-simple[9] , effect-handlers[4] , Scala では Scala-Effekt[7] , ZIO[17] などが開発されている. これらの実装の多くは, 言語の制約により single-shot に限定されている. 例外として Scala では, モナドを用いたライブラリ実装で multi-shot effect handler を実現している. 本研究では, JavaScript でジェネレータを用いた上で multi-shot の対応を試みている.

6　まとめ

本論文では, JavaScript における multi-shot effect handler の実装を, ジェネレータと while-switch 制御構造を組み合わせることで実現した. multi-shot をサポートし, CPS 変換を用いる従来手法と比較して 1.6〜1.7 倍の高速化を達成している. また, 100 個のユニットテストを作成し, 基本的な正確さを検証した. 今後の課題として, ユニットテストの拡充と, Koka など既存実装への移植によるユニットテスト自体の正しさの検証が挙げられる. なお本論文中には大規模言語

モデル Claude Opus 4 が生成した文章が含まれる．

参 考 文 献

[1] Alvarez-Picallo, M., Freund, T., Ghica, D.R., and Lindley, S.: “Effect Handlers for C via Coroutines,” *Proc. ACM Program. Lang.* 8(OOPSLA2), 2024.

[2] Babel Team: “Babel: The compiler for next generation JavaScript, ” `https://babeljs.io/`.

[3] Bauer, A.and Pretnar, M.: “The Eff programming language, ” `https://www.eff-lang.org/`, 2020.

[4] Bauer, A. and Pretnar, M.: “effect-handlers: A library for writing extensible algebraic effects and handlers,” Hackage, `https://hackage.haskell.org/package/effect-handlers`.

[5] Brachthäuser, J.I., Schuster, P., and Ostermann, K.: “Effekt: Algebraic Effect Handlers for All, ” *Proc.PLDI 2020*, pp.1–15, 2020.

[6] Brachthäuser, J.I., Schuster, P., and Ostermann, K.: “Effect Handlers for the Masses,” *Proc. ACM Program. Lang.* 2(OOPSLA), 2018.

[7] Brachthäuser, J.I., Schuster, P., and Ostermann, K.: “Effekt: Capability-Passing Style for Type- and Effect-Safe, Extensible Effect Handlers in Scala,” *Journal of Functional Programming* 30, 2020.

[8] effectsjs Team: “effectsjs: JavaScript library for algebraic effects, ” GitHub Repository, `https://github.com/effectsjs/effectsjs`.

[9] King, A.: “freer-simple: A friendly effect system for Haskell,” GitHub Repository, `https://github.com/lexi-lambda/freer-simple`.

[10] Kiselyov, O., Sabry, A., and Swords, C.: “Extensible Effects: An Alternative to Monad Transformers,” *Proc. Haskell Symposium*, pp.59–70, 2013.

[11] Koka Team: “libhandler: C library for algebraic effect handlers,” GitHub Repository, `https://github.com/koka-lang/libhandler`.

[12] Leijen, D.: “Koka: Programming with Row-Polymorphic Effect Types, ” *Proc.MSFP 2014*, EPTCS 153, pp.100–126, 2014.

[13] Leijen, D.: “Implementing Algebraic Effects in C ”Monads for Free in C”,” Microsoft Research Technical Report MSR-TR-2017-23, 2017.

[14] OCaml Development Team: “OCaml 5.0.0 Release Notes, ” `https://ocaml.org/releases/5.0.0`, 2022-12-16.

[15] Plotkin, G.and Pretnar, M.: “Handlers of Algebraic Effects, ” *Proc.ESOP 2009*, LNCS 5502, pp.80–94, 2009.

[16] van Hecke, A.(3Shain): “algebraic-effects-ts: TypeScript library for algebraic effects, ” GitHub Repository, `https://github.com/3Shain/algebraic-effects-ts`.

[17] ZIO Team: “ZIO: Type-safe, composable asynchronous and concurrent programming for Scala,” `https://zio.dev/`.