東京大学
情報理工学系研究科 創造情報学専攻
修士論文

JavaScript のジェネレータ関数を活用したマルチショット・エフェクトハンドラの実装

An Implementation of Multi-shot Effect Handlers Exploiting JavaScript's Generator Functions

武樋 一樹
Kazuki Takehi

指導教員　千葉 教授

2026 年 1 月

概要

エフェクトハンドラは，例外処理を一般化した言語機構である．近年注目を集めている言語機構であり， Ocaml，Scala，Koka など様々な言語において実装手法が提案されている．

特に，マルチショット・エフェクトハンドラは，バックトラッキングや確率的プログラミングなどを簡潔に表現できるという利点を持つ．一方，シングルショット・エフェクトハンドラは実行効率が一般的に良い．

JavaScript においてエフェクトハンドラを実装する既存手法としては，主に CPS 変換を利用したものと，ジェネレータ関数を利用したものが挙げられるが，前者は実行効率が比較的悪い一方で，後者はシングルショットに制限される．

本研究では，JavaScript のジェネレータ関数を活用し，限定的なケースを除いてマルチショットに対応するエフェクトハンドラの実装手法を提案する．提案手法では，エフェクトの発生を `yield` 操作として表現し，関数をジェネレータ関数へ変換することで，限定継続を中断状態のジェネレータとして表現する．これらのジェネレータはスケジューラによって一元的に管理され，適切な再開処理が行われる．さらに，各 `yield` 直後の処理を `while` 文で包み，`return` 文を `yield` 文に変換することで，限定継続の再実行時に制御を擬似的に巻き戻すことを可能にする．

提案手法を評価するため，Babel を基盤とした処理系 `geneffjs` を実装した．実験では，マルチショット・エフェクトハンドラを言語機構として備える Koka との動作比較を行い，一部のコーナーケースを除いて等しい出力が得られることを確認した．また，CPS 変換に基づく実装との実行速度比較を行い，どのようなケースで実行効率が相対的に良く，どのようなケースで低下するかについて，実験結果に基づいて考察した．

Abstract

Effect handlers are a language mechanism that generalizes exception handling and have attracted increasing attention in recent years. Various implementation techniques have been proposed in languages such as OCaml, Scala, and Koka. In particular, multi-shot effect handlers enable concise representations of computations involving backtracking and probabilistic programming, while single-shot effect handlers are generally more efficient in terms of execution performance.

In JavaScript, existing approaches to implementing effect handlers are mainly based on continuation-passing style (CPS) transformation or generator functions. However, CPS-based approaches tend to suffer from performance overhead, whereas generator-based approaches are restricted to single-shot.

This thesis proposes an implementation technique for effect handlers in JavaScript that supports multi-shot resumption in a wide range of cases by leveraging generator functions. In the proposed approach, effect invocations are represented as `yield` operations, and functions are transformed into generator functions, allowing delimited continuations to be represented as suspended generators. These generators are centrally managed by a scheduler, which performs appropriate resumption of generators. Furthermore, by enclosing the computation following each `yield` in a `while` loop and transforming `return` statements into `yield` statements, the proposed transformation enables pseudo rollback of control flow when the same continuation is re-executed.

To evaluate the proposed approach, we implemented a prototype system called `geneffjs` based on Babel. We compared its behavior with that of Koka, which provides multi-shot effect handlers as a language feature, and confirmed that equivalent outputs are obtained in most cases, except for a limited number of corner cases. We also conducted performance comparisons with a CPS-based implementation and analyzed, based on experimental results, the types of programs in which the proposed approach performs relatively well or poorly.

目次

第 1 章　**はじめに** 1

第 2 章　**JavaScript におけるエフェクトハンドラ** 3

2.1　CPS 変換 3

2.2　限定継続 4

2.3　コルーチンと JavaScript のジェネレータ関数 5

2.4　エフェクトハンドラ 6

2.5　エフェクトハンドラの実装方法 12

第 3 章　**JavaScript のジェネレータ関数を活用したマルチショット・エフェクトハンドラの提案** 13

3.1　基本変換規則 13

3.2　マルチショットへの対応 19

3.3　関数呼び出しの変換 23

3.4　制御構造の変換 27

3.5　最適化 32

3.6　図 2.8 の変換結果と実行追跡に基づくケーススタディ 39

第 4 章　**実験** 43

4.1　`geneffjs` の実装 43

4.2　Koka の動作との比較 43

4.3　CPS 変換による実装との実行速度の比較 53

第 5 章　**まとめと今後の課題** 58

発表文献と研究活動 61

参考文献 62

第 1 章

はじめに

エフェクトハンドラ [1, 2, 3] とは，例外処理を一般化した言語機構として提案され，近年プログラムング言語研究の分野において注目を集めている．具体的には Ocaml[4, 5, 6], Scala[7]，Koka[8] など，様々な言語において実装手法が提案されている．

エフェクトハンドラは，計算途中で発生したエフェクトに対して，その振る舞いを外部から定義できる点に特徴がある．一般的な例外処理とは異なり，エフェクトを発生させてハンドラへ制御が移る際，発生地点からの限定継続がハンドラに渡される．ハンドラはこの限定継続を捕捉し，任意の時点で実行を再開できるという特徴を持つ．同一の限定継続を複数回実行可能なものはマルチショット・エフェクトハンドラ，実行が一度のみに制限されるものはシングルショット・エフェクトハンドラと呼ばれる．

マルチショット・エフェクトハンドラは，非決定性計算やバックトラッキング，確率的プログラミングなどを簡潔に記述できるという利点を持つ．一方で，一般に限定継続の複製や管理が必要となるため，シングルショットに比べて実行効率の低下を招きやすいことが知られており，表現力と実行効率の間には大きなトレードオフが存在する．

JavaScript におけるエフェクトハンドラの実装として，主に CPS 変換を利用したものと，ジェネレータ関数を利用したものが提案されてきた．CPS 変換を用いる場合は，マルチショット・エフェクトハンドラを実装できる一方で，継続の生成や関数呼び出しの増加に伴うオーバーヘッドが大きく，実行効率に課題がある．一方，ジェネレータ関数を利用する手法は，JavaScript の組み込み機能を活用できるため比較的高い実行性能が期待できるが，一般にはシングルショットに制限されるという問題がある．

本研究の動機は，このような既存手法の長所と短所を踏まえ，JavaScript のエフェクトハンドラにおいて実行効率と表現力の両立を目指す点にある．特に，CPS 変換によるマルチショット・エフェクトハンドラは実行効率が低いという課題がある一方で，ジェネレータ関数は JavaScript の言語仕様として広く最適化が施されている機構である．そのため，ジェネレータ関数を基盤として可能な限りマルチショットに対応するようなエフェクトハンドラを実装することで，CPS 変換に比べて高い実行性能が得られることが期待される．

本研究では，JavaScript のジェネレータ関数 [9] を活用し，限定的なケースを除いてマルチショットに対応するエフェクトハンドラの実装手法を提案する．具体的には，エフェクトの発

生を yield 操作に対応付け，エフェクトを内部で発生させる関数をジェネレータ関数に変換することで，限定継続を中断状態のジェネレータとして表現する．これらのジェネレータはスケジューラによって一元的に管理され，適切な順序で再開される．さらに，各 yield 直後の処理をラベル付き while 文で包み，ジェネレータの末尾に制御用の yield を挿入するコード変換を導入する．これにより，同一の限定継続を再度実行する際に，while 文を通じて実行位置まで制御を擬似的に巻き戻すことが可能となり，ジェネレータが本来備えていないマルチショット再開を可能な限り実現する．

本研究では，提案手法を評価するため，Babel[10] を基盤とした処理系 geneffjs を実装した．geneffjs は，try-handle や perform といった独自構文を含む JavaScript 風のコードを入力として受け取り，提案手法に基づいて同値な動作を行うネイティブな JavaScript コードへ変換し，その変換後のコードを実行する．

また，geneffjs を用いて二つの実験を行った．一つ目は，マルチショット・エフェクトハンドラを言語機構として備える Koka と動作を比較し，提案手法が意図した意味論を満たすかを確認する正当性検証である．二つ目は，CPS 変換に基づく既存実装と実行時間を比較し，提案手法の性能特性を評価する実行性能評価である．

実験の結果，正当性については，多くのサンプルプログラムで Koka と同一の出力が得られることを確認した一方，一部のプログラムでは不一致が生じ，その原因を三つのカテゴリに分類して整理した．また実行性能については，提案手法が常に CPS 変換を上回るわけではなく，プログラムの構造に依存して高速化・低速化が生じることが分かった．具体的には，ネイティブな関数呼び出しや分岐・ループとして実行できる部分が支配的な場合や，同一の限定継続を多数回再開する場合には有利になりやすい一方，継続分割を要する制御構造が支配的な場合には不利になり得る．

本研究の貢献は，次の三点にまとめられる．

- JavaScript において，ジェネレータ関数を利用し，広範なケースでマルチショットに対応するエフェクトハンドラの実装手法を提案したこと．
- Babel を活用し，提案手法を実現する処理系 geneffjs を実装したこと．
- 実装した処理系を用いて，Koka との動作比較による正当性の確認と，CPS 変換手法との実行速度比較を行い，提案手法の性能特性を実験的に明らかにしたこと．

以下，本論文の構成を示す．第 2 章では，CPS 変換，限定継続，JavaScript のジェネレータ関数，エフェクトハンドラ，および既存のエフェクトハンドラ実装について概説する．第 3 章では，ジェネレータ関数と while 文を用いたマルチショット・エフェクトハンドラの具体的な変換規則を提案する．第 4 章では，geneffjs の実装および実験結果について述べ，提案手法の正当性と性能特性を評価する．最後に第 5 章では，本研究のまとめと今後の課題について述べる．

第 2 章

JavaScript におけるエフェクトハンドラ

本章では，本研究を理解する上で必要となる前提知識について説明する．まず，CPS 変換について概説する．次に，限定継続について述べる．さらに，コルーチンと JavaScript におけるジェネレータ関数について述べる．また，エフェクトハンドラの定義とその動作について述べる．最後に，既存研究において提案されてきたエフェクトハンドラの実装方法を整理した上で，JavaScript においてどのような実装手法が提案されてきたかを概観し，それぞれの長所と短所について述べる．

2.1 CPS 変換

継続とは，プログラムのある時点における，その後に実行される計算を表す概念である．ある計算が現在の評価を終えた後，どのような処理が続くかを抽象的に捉えたものと考えることができる．

継続の概念を明示的に扱う代表的な手法が CPS (Continuation Passing Style) 変換である．あるプログラムを CPS 変換するとは，計算結果を直接返すような関数を，次に実行すべき処理を表す継続を引数として受け取り計算結果をその継続に渡す形式へと書き換えることを指す．

例えば，図 2.1 に示すような通常の JavaScript プログラムでは，関数 `add` や `multiply` は，計算結果を呼び出し元に直接返す．一方，図 2.1 のプログラムを CPS 変換したものを図 2.2 に示す．CPS 変換後のプログラムでは，各関数は計算結果を返す代わりに，その後の処理を表す継続を引数として受け取り，計算結果をその継続に渡す形で処理を進める．

このように，CPS 変換では「その後の計算」を関数として明示的に受け渡すことで，プログラムの制御の流れを操作する．この意味で CPS 変換は，継続という概念をプログラム上で直接扱える形に書き下す手法であるといえる．

```
1 function add(a, b) {
2   return a + b;
3 }
4 function multiply(a, b) {
5   return a * b;
6 }
7 let sum = add(2, 3);
8 let result = multiply(sum, sum);
9 console.log(result);
```

図 2.1. 通常の関数呼び出しによる逐次的な計算の例

```
1 function addCPS(a, b, k) {
2   k(a + b);
3 }
4 function multiplyCPS(a, b, k) {
5   k(a * b);
6 }
7 addCPS(1, 2, function (sum) {
8   multiplyCPS(sum, sum, function (result) {
9     console.log(result);
10   })
11 })
```

図 2.2. 図 2.1 のプログラムを CPS 変換した例

2.2　限定継続

CPS 変換において扱われる継続は，プログラム全体の「残りの計算」を表す完全な継続である．これに対し，実際の言語機構や制御構造では，プログラム全体ではなく，特定の範囲に限定された「その後の計算」のみを操作したい場面が多い．

限定継続 [11] とは，このように，プログラムのある時点における継続のうち，その適用範囲を特定の区間に制限したものである．限定継続は，関数型言語において `reset` によって継続の範囲を区切り，`shift` によってその範囲内の継続を捕捉・操作する，といった限定継続演算子によって扱われることが多い．

限定継続を用いることで，プログラム全体の制御を破壊することなく，特定の計算区間のみを中断・再開することが可能となる．この性質は，例外処理やエフェクトハンドラといった高

```
1 function* myCoroutine() {
2   console.log("start");
3   yield 1;
4   yield 2;
5   return "done";
6 }
7
8 const gen = myCoroutine()
```

図 2.3. ジェネレータ関数の定義と，ジェネレータオブジェクトの生成

度な制御構造を実現する上で重要な役割を果たす．

2.3 コルーチンと JavaScript のジェネレータ関数

コルーチンは，実行を一時的に中断し，後に中断した箇所から再開できる制御構造である．また，その制御の方式に基づき，「非対称コルーチン」と「対称コルーチン」に分類することができる．非対称コルーチンは，呼び出し側と呼び出される側の間に明確な親子関係が存在する点に特徴がある．親は `resume` 操作によって子のコルーチンを再開し，実行権を子に譲渡する．一方，子は `yield` 操作によって実行を一時停止し，呼び出し元である親に実行権を返却する．これに対し，対称コルーチンでは特定の呼び出し元に戻る必要はなく，任意のコルーチン間で対等に実行権を移譲し合うことができる．

JavaScript のジェネレータ関数を呼び出すことで生成されるジェネレータは，非対称コルーチンを言語レベルで実装したものである．ジェネレータ関数は，図 2.3 に示すように，通常の関数定義に `*` を付与することで定義でき，関数本体では `yield` キーワードを使用することが可能である．通常の関数とは異なり，ジェネレータ関数を呼び出しただけでは関数本体の処理は実行されない．呼び出しの結果として返されるのは，実行状態を管理するためのジェネレータオブジェクトであり，このオブジェクトを非対称コルーチンとしてみなすことができる．

非対称コルーチンにおいて，親である呼び出し元が子のコルーチンを再開させる操作は，ジェネレータの `next()` メソッドに対応する．`next()` を呼び出すことで，親は子に実行権を譲渡し，子は次の `yield` 文，あるいは関数の終端に達するまで処理を進める．`yield` に到達すると，子は実行状態を保存したまま，親に実行権と値を返す．`next()` を呼び出すたびに，ジェネレータは値だけでなく，実行状態を表すオブジェクトを返す．このオブジェクトにおける `value` は，`yield` または `return` によって親に渡された値を表し，`done` はジェネレータが完全に終了したかどうかを示すフラグである．

図 2.4 は，図 2.3 で生成したジェネレータに対して `next()` メソッドを順に実行し，その返り値を出力するプログラムである．`yield` または `return` によって返却された値が，呼び出し元に返されることが分かる．

```
1 console.log(gen.next());
2 // { value: 1, done: false }
3
4 console.log(gen.next());
5 // { value: 2, done: false }
6
7 console.log(gen.next());
8 // { value: "done", done: true }
```

図 2.4. next() メソッドによるジェネレータの再開と返り値の遷移

2.4 エフェクトハンドラ

2.4.1 エフェクトハンドラの基本概念

エフェクトハンドラとは，例外処理を一般化した言語機構であり，計算の途中で発生したエフェクトに対して，その振る舞いを外部からハンドラとして定義できる仕組みである．通常の例外処理と同様に，エフェクトが発生すると対応するハンドラに制御が移るが，エフェクトハンドラでは，発生地点以降の計算を再開可能な形で扱える点に特徴がある．

エフェクトハンドラは，`try {e} handle eff with {H}` のような構文で記述される．ここで，`e` はエフェクトを発生させる可能性のある計算であり，`H` は特定のエフェクト `eff` が発生した際の処理を定義する．計算中で `perform` によってエフェクトが発生すると，囲われたハンドラが探索され，対応するハンドラに制御が移る．その際ハンドラにはエフェクトが発生した地点から `try` 節の終わりまでの限定継続が渡され，任意の時点でその限定継続を呼ぶことができる．この機能により，エフェクトハンドラは単なる例外処理を超えた柔軟な制御構造を実現する．

2.4.2 エフェクトの発生と計算の再開

本節では，エフェクトハンドラにおける基本的な実行の流れとして，エフェクトの発生と計算の再開について説明する．エフェクトハンドラを用いたプログラムでは，計算の途中で `perform` によってエフェクトが発生すると，対応するハンドラに制御が移る．ハンドラは `resume` を呼ぶことで，`perform` 以降の計算を再開できる．

図 2.5 は，`getNum` というエフェクトの発生，`getNum` エフェクトを処理する `getNum` ハンドラ，計算の正常終了を処理する `return` ハンドラ，を含むプログラム例である．`try` 節にはエフェクトを発生させる可能性のある計算が記述され，二つの `handle` 節には，それぞれ `getNum` エフェクトを捕捉するハンドラおよび `return` ハンドラが定義されている．また以後，あるエフェクト `A` を捕捉するようなハンドラを，`A` ハンドラと呼ぶ．

```
1 try {
2   return 8 + perform("getNum")
3 } handle "getNum" with {
4   console.log(resume(0));
5 } handle "return" with (x) {
6   return x * 2;
7 }
```

図 2.5. getNum エフェクトとそのハンドラが記述されたプログラム

このプログラムが評価されると，まず try 節の計算が実行される．2 行目で perform("getNum") が呼び出されると，現在の計算は中断され，getNum ハンドラである 3 行目から 5 行目までの handle 節に制御が移る．このとき，perform が呼び出された位置から try 節の末尾までの計算を示す限定継続がハンドラに渡される．

ハンドラ内では，resume を用いて限定継続を実行することができる．すなわち，中断された計算を再開することができる．例えば，図 2.5 の例では，4 行目の resume(0) によって限定継続が実行され，中断された計算が再開される．このとき，resume に渡した引数が perform 呼び出しの評価値となる．perform 以降の計算が再開されると，perform の評価値は 0 となり，return 8 + 0 が評価される．その結果，try 節の計算が終了し，制御は計算終了時の処理が定義された，5 行目から 7 行目までの return ハンドラに移る．

return ハンドラでは，try 節が返した値が引数として渡される．図 2.5 の例では，return ハンドラの引数 x に 8 が束縛され，return x * 2 が評価される．ここで return ハンドラが値を返すと限定継続の実行は終わり，またそこで返された値が 4 行目の resume の評価値となる．したがって，4 行目の console.log によって 16 が出力される．handle 節を抜けると，プログラム全体が終了する．

2.4.3 perform によるエフェクトの発生と resume による計算の再開を示した例

図 2.5 に示したようなエフェクトハンドラの利用は，実用的には，図 2.6 のようにハンドラを関数として定義し，ハンドルしたい計算を関数として受け取る形式で記述されることが多い．このように記述することで，エフェクトを発生させる側の計算と，それを処理する側のハンドラを明確に分離することが可能となる．ハンドラとエフェクトを発生する計算を分離することで，同一のエフェクトの発生に対して，囲うハンドラを切り替えるだけで異なる振る舞いを与えることができる．すなわち，計算の意味や実行結果を外部から制御できるようになる．

例えば図 2.7 に示すように，getNum ハンドラの実装を変更した関数 DivByZero2 を用いると，ハンドルする関数 handledFunc は同一であるにもかかわらず，10 行目の DivByZero2(handledFunc) は 図 2.6 における DivByZero(handledFunc) とは異なる値

```
1  function DivByZero(action) {
2    try {
3      return action();
4    } handle "getNum" with {
5      console.log(resume(0));
6    } handle "return" with {
7      return x * 2;
8    }
9  }
10 function handledFunc() {
11   return 8 + perform("getNum");
12 }
13 DivByZero(handledFunc);
```

図 2.6. ハンドラとハンドルする計算を関数として分離した例

```
1  function DivByZero2(action) {
2    try {
3      return action();
4    } handle "getNum" with {
5      console.log(resume(10));
6    } handle "return" with {
7      return x * 2;
8    }
9  }
10 DivByZero2(handledFunc);
```

図 2.7. ハンドラの実装を変更することで計算結果が変化する例

を出力する．この例では，5 行目の `resume(10)` によって再開された `handledFunc` の返り値 `8 + 10` が `return` ハンドラによって変換され，最終的に 36 が出力される．

2.4.4　マルチショット・エフェクトハンドラ

渡された限定継続を二回以上実行することが可能なエフェクトハンドラを，マルチショット再開に対応したエフェクトハンドラと呼ぶ．以後，そのようなエフェクトハンドラを，マルチショット・エフェクトハンドラと呼ぶ．マルチショット・エフェクトハンドラでは，同一のエフェクト発生地点以降の計算を，異なる値を用いて複数回再開することができる．また，同一

の限定継続を高々一回のみ実行できるようなエフェクトハンドラを，シングルショット・エフェクトハンドラと呼ぶ．

例えば，図 2.8 に示すハンドラ `DivByZeroMultiShot` を考える．これは，図 2.6 に示した `DivByZero` 関数における `getNum` ハンドラを，`resume` を二回呼び出すよう変更したものである．また，説明を簡単にするため，`return` ハンドラは `try` 節の評価結果をそのまま返すものとする．14 行目の `DivByZeroMultiShot(handledFunc)` が評価されると，まず `handledFunc` が実行され，12 行目で `perform("getNum")` が呼び出される．これにより，計算は中断され，限定継続 `return 8 + □` が `getNum` ハンドラに渡され，制御がハンドラに移る．

`handle` 節内では，まず 5 行目の `resume(0)` が実行され，限定継続に値 0 が渡される．これにより，中断された計算が再開され，`return 8 + 0` が評価される．再開された計算が終了すると，その評価結果である 8 が `resume(0)` の評価値となり，`console.log` によって出力される．次に，6 行目の `resume(10)` が実行され，同一の限定継続が再び呼び出される．このとき，12 行目の `perform` の評価値は 10 となり，`return 8 + 10` が評価される．その結果，限定継続の評価結果は 18 となり，`resume(10)` の返り値として 18 が得られ，これが出力される．

このように，マルチショット・エフェクトハンドラでは，同一の限定継続を複数回実行することが可能であり，シングルショット・エフェクトハンドラと比べて表現力が高い．例えば，複数の選択肢を分岐して扱う計算を簡潔に記述することができる．一方で，一般に限定継続の複製が必要となるため，実行効率が低下することが知られており，表現力と実行効率の間には大きなトレードオフが存在する [12].

2.4.5 マルチショット・エフェクトハンドラによるプログラミングの例

モンティ・ホール問題を解くプログラムを，マルチショット・エフェクトハンドラを用いて簡単に記述する．モンティ・ホール問題は，条件付き確率に関する古典的な問題である．3 つのドアのうち 1 つには当たりがあり，残りの 2 つには外れがあるとする．プレイヤーはまず 1 つのドアを選択する．その後，出題者はプレイヤーが選ばなかったドアのうち外れであるものを 1 つ開示する．このとき，プレイヤーは最初の選択を維持するか，あるいは残された別のドアに変更するかを選択できる．問題は，ドアを変更した場合としない場合，どちらが当たりを得る確率を最大化するかを問うものである [13].

この問題は，複数の選択肢を分岐として扱う非決定性計算として捉えることができる．ここでは，非決定的な選択を表すエフェクト `choose` を導入し，マルチショット・エフェクトハンドラによってすべての選択肢を列挙することで，モンティ・ホール問題を記述する．

図 2.9 の関数 `choose` は，候補の集合 `options` を受け取り，その中から 1 つを選択する非決定的な計算を表す．関数 `choose` 自体は具体的な選択を行わず，対応するエフェクトハンドラによって選択の処理が定義される．図 2.10 において，`openCandidates` は，プレイヤーの初期選択に応じて出題者が開け得るドアの集合を返す関数である．候補が 2 つ存在する場合

```
1  function DivByZeroMultiShot(action) {
2    try {
3      return action();
4    } handle "getNum" with {
5      console.log(resume(0));
6      console.log(resume(10));
7    } handle "return" with (x) {
8      return x;
9    }
10 }
11 function handledFunc() {
12   return 8 + perform("getNum");
13 }
14 DivByZeroMultiShot(handledFunc)
```

図 2.8. 限定継続を複数回再開するマルチショット・エフェクトハンドラの例

```
1  function choose(options) {
2    return perform({ type: "choose", options });
3  }
```

図 2.9. 非決定的な選択を表すエフェクト

はそれらをそのまま要素とした配列を返すが，候補が 1 つしか存在しない場合には同一要素を 2 個含む配列を返す．これは，choose による分岐の列挙によって確率を計算する際に，場合によって候補集合の要素数が変化すると，各分岐の重みあるいは確率が，暗黙に変化してしまうためである．分岐数を揃えることで，「2 択なら各 1/2，1 択なら確率 1」という重み付けを表現している．

図 2.10 の関数 montyHall は switchDoor を受け取ると，まず 11 行目でプレイヤーの初期選択を choose によって非決定的に選択する．ただし switchDoor は最終的にドアを変更する戦略を採る場合に真となる真偽値である． choose は図 2.9 で定義された関数である．また，当たりのドアは定数 PRIZE によって固定されている．次に，12 行目でどのドアが出題者によって開示されたかが非決定的に選択される．最後に switchDoor に応じてプレイヤーが最終的な選択を行い，戻り値として当たりを得られたかどうかを表す真偽値 finalChoice === PRIZE を 17 行目で返す．

図 2.11 の関数 allChoices には，choose エフェクトを捕捉するハンドラが定義されている．エフェクトを捕捉した際に，候補集合 options の各要素 option に対して，渡された限定継続を resume(option) によって再開する．これにより，すべての分岐を探索することが

```
1  const PRIZE = 0, DOORS = [0,1,2];
2
3  function openCandidates(firstChoice) {
4    let candidates = DOORS.filter(d => d !== PRIZE && d !==
        firstChoice);
5    return candidates.length == 1
6      ? [candidates[0], candidates[0]]
7      : candidates;
8  }
9
10 function montyHall(switchDoor) {
11   const firstChoice = choose(DOORS);
12   const opened = choose(openCandidates(firstChoice));
13   const finalChoice = switchDoor
14     ? DOORS.filter(d => d !== firstChoice && d !== opened)
         [0]
15     : firstChoice;
16
17   return finalChoice === PRIZE;
18 }
```

図 2.10. モンティ・ホール問題を表す非決定的な計算

```
1 function allChoices(action) {
2   try {
3     return action();
4   } handle "choose" with ({options}) {
5     return options.map(option => resume(option)).flat();
6   }
7 }
```

図 2.11. すべての分岐した結果を列挙するマルチショット・エフェクトハンドラ

可能となる．最終的に allChoices 関数は，全ての分岐先における結果が列挙された配列を返す．

図 2.12 のプログラムを実行すると，stayResults には，ドアを変更しない戦略を採る場合におけるすべての可能な分岐での，真偽値で表された結果が入った配列が代入される．同様に，switchResults は，ドアを変更する戦略を採る場合におけるすべての可能な結果を示し

```
1 const stayResults = allChoices(() => montyHall(false));
2 const switchResults = allChoices(() => montyHall(true));
3 console.log(stayResults.filter(x => x).length /
      stayResults.length) // 0.3333...
4 console.log(switchResults.filter(x => x).length /
      switchResults.length) // 0.6666...
```

図 2.12. 戦略ごとの当たる確率の計算

た配列となる．ドアを変更しない戦略では当たりを得る確率が 0.3333 . . .，変更する戦略では 0.6666 . . . となる．結果として，マルチショット・エフェクトハンドラを用いることで，モンティ・ホール問題を解くプログラムを比較的簡潔に記述できることを示した．特に，図 2.10 に示した `montyHall` 関数は，問題の構造と計算の流れが直接対応しており，問題設定とプログラムとの対応関係を容易に読み取ることができる．

2.5　エフェクトハンドラの実装方法

エフェクトハンドラの実装方法としては，代表的には次のような系統が知られている．すなわち，限定継続演算子の利用，スタック操作，モナドによる埋め込み，コルーチンの利用，および CPS 変換である．限定継続演算子を利用した実装としては，OCaml における実装 [6] などが挙げられる．スタック操作を伴う実装としては，C[14]，Java[15]，multicore OCaml[5] などが例として挙げられる．また，Haskell[16, 17] や Scala[7] などでは，モナドに基づく実装が議論されてきた．さらに，コルーチンを用いた実装も提案されており [18, 19]，Lua や Ruby における実装例が報告されている [20]．最後に，CPS 変換を用いた実装手法も提案されている [21, 22]．

これらのうち JavaScript では，主に CPS 変換を用いる手法とジェネレータ関数を用いる手法に集約される．CPS 変換による実装はマルチショット・エフェクトハンドラに対応する一方で，クロージャの生成や関数呼び出しの増加に伴うオーバーヘッドにより実行効率が低下しやすい．一方，ジェネレータ関数を用いた実装は，言語組み込みの中断・再開機構を活用できるため実行効率の面での利点が期待できるが，既存の代表的手法ではシングルショットに留まることが多い．CPS 変換を利用した実装としては，js_of_ocaml[23] や Links[24] などが挙げられる．ジェネレータ関数を利用した実装としては，effectsjs[25] などが挙げられる．

第 3 章

JavaScript のジェネレータ関数を活用したマルチショット・エフェクトハンドラの提案

本章では，JavaScript のジェネレータ関数と `while` 文を組み合わせることにより，マルチショット・エフェクトハンドラを実現する実装手法を提案する．本手法の核心は，ジェネレータがもつ中断・再開機構を活用しつつ，ラベル付き `while` 文を用いることで，同一の限定継続を複数回再開可能とする点にある．

本章の構成は次の通りである．まず 3.1 節で，`perform`，`try-handle`，`resume` をジェネレータとスケジューラで実現する基本枠組みを説明する．次に 3.2 節で，ラベル付き `while` 文による巻き戻しを導入し，同一の限定継続を複数回実行可能とする方法を述べる．続いて 3.3 節で，関数呼び出しに対する変換を示す．3.4 節では，`if` 文，ループ文などのに対する変換を示す．最後に 3.5 節で，実行効率を改善するための最適化について議論する．

3.1 基本変換規則

本節では，提案手法の基本枠組みとして，エフェクトの発生・捕捉・再開をジェネレータ関数とスケジューラの協調によって実現する方法を述べる．提案手法では，`perform` 式を `yield` に対応付け，`try` 節および `handle` 節をそれぞれジェネレータとして表現する．以後，`try` 節に対応するジェネレータを try ジェネレータ，ハンドラ本体に対応するジェネレータをハンドラ・ジェネレータと呼ぶ．ジェネレータが `yield` により実行を中断すると制御はスケジューラに戻る．スケジューラは `yield` された値に応じて適切な処理を行った後，次に実行すべきジェネレータを決定する．

以下では，図 3.1 のプログラム例と，その変換結果を示した図 3.2 のプログラムおよびスケジューラを示した 図 3.3, 3.4 を用いて，具体的な実行の流れを説明する．図中の「処理 A」「処理 B」などは，`perform` や関数呼び出しを含まない任意の処理を表す抽象的な記述である．図 3.1 における `try` 節は，図 3.2 の 2 行目から 20 行目に示す try ジェネレータ `_tryGen` に

```
1  try {
2    // 処理A
3    let result1 = perform("T1");
4    // 処理B
5  } handle "T1" with {
6    // 処理C
7    let r1 = resume(arg1);
8    // 処理D
9    let r2 = resume(arg2);
10   // 処理E
11 }
```

図 3.1. try - handle 構文によるエフェクトハンドラを利用したプログラムの例

置き換えられる．また，エフェクト T1 を処理する handle 節は，図 3.2 の 21 行目から 29 行目に示す関数 handler_T1 として表現される．プログラム全体は，ジェネレータ関数 main によって囲まれる．プログラムの実行は，図 3.2 の 35 行目に示す runScheduler(nextGen: main()) によって開始される．このときスケジューラには，最初に実行すべきジェネレータである main() が渡される．

変換後のジェネレータは，yield の際に実行状態を表すオブジェクトを返す．本実装では主に，RUN_TRYGEN, PERFORM, TRY_DONE の 3 種類のオブジェクトを用いる．これらはそれぞれ対応する関数呼び出しによって生成されるのではなく，引数として渡された値（オブジェクトまたは値）に対してプロパティ "MESSAGE" を付与し，そのまま返却するための補助関数である．具体的には，RUN_TRYGEN(obj) は，引数 obj に obj["MESSAGE"] = "RUN_TRYGEN" を設定して返す．同様に，PERFORM(obj) は obj["MESSAGE"] = "PERFORM" を設定して返す．したがって，例えば PERFORM に {type: "getNum", label: "l1"} を渡すと，{MESSAGE: "PERFORM", type: "getNum", label: "l1"} が返る．一方，TRY_DONE(result) は，MESSAGE に "TRY_DONE" を設定したオブジェクトを返し，併せて，当該 TRY_DONE を生成したジェネレータに関する情報を付与する．具体的には，当該ジェネレータが try 節に対応する場合は isHandler=false とし，handle 節に対応する場合は isHandler=true とした上で，同一の try ...handle ... として定義された try 節のジェネレータを tryGenerator に格納する．

まず，main は図 3.2 の 30 行目において RUN_TRYGEN オブジェクトを yield する．これを受け取ったスケジューラは，図 3.3 の 12 行目から 19 行目に示す処理により，_tryGen に対してハンドラ handler_T1 を登録した上で，次に実行するジェネレータとして _tryGen を選択する．

次に，図 3.1 の 3 行目において perform("T1") が実行されると，図 3.2 の 4 行目で

PERFORM オブジェクトが yield される．すると制御がスケジューラに戻り，図 3.4 の 1 行目から 19 行目の処理が実行される．ここでは，yield を行ったジェネレータ（_tryGen）を起点として，PARENT を辿りながら，対応するハンドラが登録された try ジェネレータを探索する．ハンドラ handler_T1 が見つかると，図 3.4 の 17 行目で handler が呼び出され，ハンドラ・ジェネレータ handlerGen が生成される．この際，ハンドラが限定継続を再開するために必要な resume オブジェクトを handler 関数に渡す．resume には，nextGen に PERFORM を yield したジェネレータ，tryGenerator に対応する try ジェネレータ，および performLabel に再開位置を示すラベル情報が格納される．その後，図 3.4 の 18 行目で，次に実行するジェネレータとして gen に handlerGen が代入され，ハンドラ側の処理へ制御が移る．

ハンドラが resume を実行すると，図 3.2 の 23 行目の runScheduler(resume, arg1) に示すように，スケジューラが再度呼び出される．このときスケジューラは，図 3.3 の 5 行目から 8 行目において，resume オブジェクトに格納された tryGenerator.afterTryDone をインクリメントし，図 3.3 の 10 行目で nextGen，すなわち PERFORM を行ったジェネレータを，再開ラベルと引数を与えて実行する．

try 節側の実行が末尾まで進むと，図 3.2 の 12 行目で TRY_DONE が yield され，スケジューラは図 3.4 の 21 行目から 35 行目の処理を行う．ここで _tryGen.afterTryDone が 0 より大きい場合，この TRY_DONE は runScheduler(resume, ...) によって起動された実行の終端であるため，図 3.4 の 30 行目の return によりスケジューラ自身の実行を終了し，制御をハンドラへ戻す．同時に afterTryDone をデクリメントする．

以上により，ハンドラは resume を呼び出すたびに，スケジューラを介して try ジェネレータの実行を進め，try 節の終端到達時に呼び出し元へ復帰できる．ただし，同一の限定継続を複数回再開するための巻き戻しは，次節で導入するラベル付き while 文により実現される．

```
1  function* main() {
2      function* _tryGen() {
3        // 処理 A
4        let {RESUME_ARG: result} = yield PERFORM({
5          type: "T1",
6          label: "l1",
7          arg: null,
8        });
9        l1: while (true) {
10         // 処理 B
11         // try 節の終端にたどり着いたことを通知
12         const { LABEL, RESUME_ARG } =  yield TRY_DONE();
13         switch (LABEL) {
14             case "l1":
15                 result = RESUME_ARG;
16                 continue l1;
17           // 他の perform がある場合は追加の case 節
18         }
19       }
20     }
21     function* handler_T1(resume) {
22         // 処理 C
23         let r1 = runScheduler(resume, arg1);
24         // 処理 D
25         let r2 = runScheduler(resume, arg2);
26         // 処理 E
27         const { LABEL, RESUME_ARG } = yield TRY_DONE();
28         switch (LABEL) {...}
29     }
30     yield RUN_TRYEGEN({
31         tryGenerator: _tryGen(),
32         handlers: [{T1: handler_T1}]
33     })
34 }
35 runScheduler({nextGen: main()});
```

図 3.2. 図 3.1 のプログラムを提案手法に基づいて変換した例

```
1  function runScheduler(info, ...args) {
2    let gen = info.nextGen, arg;
3
4    // resume により再開する場合の準備
5    if (info.isResume) {
6      info.tryGenerator.afterTryDone++;
7      arg = {WHICH_LABEL: info.performLabel, RESUME_ARG:
         args}
8    }
9    for (;;) {
10     const { value, done } = gen.next(arg);
11     switch (value.MESSAGE) {
12       case "RUN_TRYGEN":
13         // try ジェネレータの実行を開始する
14         let {tryGenerator, handlers} = value;
15         tryGenerator.PARENT = gen;
16         tryGenerator.afterTryDone = 0;
17         tryGenerator.handlers = handlers;
18         gen = tryGenerator, arg = null;
19         continue;
20
21       case "PERFORM":
22         // 図\ref{fig:3.4} を参照
23         /* ... */
24         continue;
25
26       case "TRY_DONE":
27          // 図\ref{fig:3.4} を参照
28          /* ... */
29          continue;
30         }
31     }
32 }
```

図 3.3. ジェネレータの実行を管理するスケジューラの基本構造

```
1  case "PERFORM":
2    let handler, tryGen, parent = gen;
3    while (parent) {
4        if (isTryGen(parent) && parent.handlers[value.type])
             {
5            tryGen = parent;
6            handler = parent.handlers[value.type];
7            break;
8        }
9        parent = parent.PARENT;
10   }
11   const resume = {
12       isResume: true,
13       performLabel: value.label,
14       tryGenerator: tryGen
15       nextGen: gen
16   }
17   const handlerGen = handler(resume);
18   gen = handlerGen, arg = null;
19   continue;
20
21 case "TRY_DONE":
22   let tryGen = value.tryGenerator;
23   if (tryGen.returnHandler && !value.isHandler) {
24       gen = tryGen.returnHandler(value.arg), arg = null;
25       continue;
26   }
27   if (tryGen.afterTryDone > 0) {
28       tryGen.afterTryDone -= 1;
29       return value.arg;
30   } else {
31       gen = tryGen.PARENT, arg = {
32           WHICH_LABEL: tryGen.label,
33           RESUME_ARG: value.arg
34       };
35       continue;
36   }
```

図 3.4. PERFORM および TRY_DONE に対するスケジューラの処理

3.2　マルチショットへの対応

マルチショット・エフェクトハンドラに対応するためには，一度 resume により再開した限定継続を，さらに再び同じ地点から再開できなければならない．しかし，JavaScript のジェネレータは，同一の yield 位置から複数回 next によって再開する機構を備えていない．そのため，単純に perform を yield に置き換えるだけでは，resume を複数回呼び出すハンドラを実装できない．

そこで本提案では，perform に対応する yield 以降の処理である限定継続を，ラベル付き while 文で囲むことにより，同一の限定継続を繰り返し実行可能とする．具体的には，perform 直後から try 節末尾までの区間を while で囲み，その区間の末尾に到達したことをスケジューラへ通知するため，末尾に return を挿入した上でこれを yield TRY_DONE に変換する．さらに，yield TRY_DONE によって一時停止したジェネレータをスケジューラが再開する際に，どの perform 直後へ巻き戻るべきかを示すラベルをジェネレータへ渡し，そのラベルに応じてラベル付き continue によって対応する while の先頭へ制御を戻す．

まず，図 3.1 に示した resume を二回呼び出す例を再考する．図 3.2 では，perform 直後の処理を示した図 3.1 の処理 B を繰り返し実行可能とするため，_tryGen 内にラベル付き while 文が挿入されている．具体的には，図 3.2 の _tryGen において，yield PERFORM の直後から try 節の末尾までに相当する区間が，図 3.2 の 9 行目から 18 行目においてラベル l1 を持つ while 文で囲まれる．この while 文の本体末尾では，ジェネレータが終了しないように一旦停止し，かつ末尾に到達したことをスケジューラへ通知する必要がある．そのため，本提案では図 3.2 の 12 行目にあるように， try 節の末尾に yield TRY_DONE を挿入する．スケジューラが _tryGen を再開する際には，再開先ラベルである l1 をジェネレータへ渡し，switch 文を介して対応するラベル付き continue を実行する．これにより，図 3.2 の 16 行目の continue l1 によって 10 行目の while 文の先頭へ制御が戻り，perform 直後の処理が再び実行される．

次に，resume が二回呼び出された場合の実行の流れを，図 3.1 と図 3.2 の行を参照しながら説明する．図 3.1 のハンドラ本体では，7 行目で一回目の resume(arg1) を呼び出し，続いて 9 行目で二回目の resume(arg2) を呼び出す．変換後の図 3.2 では，これらはそれぞれ 23 行目および 25 行目の runScheduler(resume，...) に対応する．

まず一回目の再開を考える．図 3.2 の 23 行目で runScheduler(resume，arg1) が呼ばれると，スケジューラは resume オブジェクトに含まれる，PERFORM を yield したジェネレータを示す nextGen を再開し，perform 直後から try 節末尾までを実行する．この実行が try 節末尾に到達すると，_tryGen は図 3.2 の 12 行目で yield TRY_DONE により停止する．スケジューラ側では，この TRY_DONE を受け取ると afterTryDone を用いた分岐によって runScheduler(resume，arg1) の呼び出し元であるハンドラへ制御を戻す．ただし，afterTryDone の役割は 3.1 節で述べた．

続いて二回目の再開を考える．ハンドラは図 3.2 の 25 行目で runScheduler(resume，

arg2) を呼び出す．このとき resume.performLabel には当該 perform を識別するラベル l1 が格納されているため，スケジューラは再開引数として LABEL = l1 と RESUME_ARG = arg2 を構成し，_tryGen を再開する．_tryGen は図 3.2 の 12 行目の yield で停止していたので，再開直後に switch 文を実行し，case "l1" に一致する．その結果，16 行目の continue l1 によりラベル l1 を持つ while 文の先頭へ制御が戻り，perform 直後の処理である「処理 B」が再び実行される．このようにして，同一の perform が生成した限定継続を二回実行できる．

次に，try 節内で複数回 perform が実行される場合を考える．図 3.5 は try 節内で perform を n 回実行する例であり，図 3.6 はその変換後の概要である．図 3.5 の「処理 A_i」は, 関数呼び出しや perform を含まない任意の処理を表す抽象記述である．図 3.6 の _tryGen では，各 perform に対してラベル l1, l2, ..., ln を付与し，それぞれの perform 直後から try 節末尾までの区間を対応するラベル付き while 文で囲む．例えば，1 回目の perform は図 3.6 の 3 行目で yield PERFORM(...label: "l1" ...) に変換され，その直後の区間は l1: while (true) で囲まれる．同様に，2 回目の perform は label: "l2" を持ち，l2: while (true)（9 行目以降）で囲まれる．n 回目の perform も label: "ln" を持ち，ln: while (true) の内側に配置される．

ここで重要なのは, try 節末尾の停止点を一箇所に集約し, そこからどの perform 直後へ巻き戻るかを switch 文で選択できるようにする点である．図 3.6 では，最内側の ln: while 内において 15 行目の yield TRY_DONE() により一時停止し, 直後の switch で WHICH_LABEL に応じて適切な continue を行う．例えば WHICH_LABEL = "l1" であれば continue l1 により最外側の while の先頭へ戻り，WHICH_LABEL = "ln" であれば continue ln により最内側の while の先頭へ戻る．

以下では，図 3.5 のハンドラが二回 resume を呼び出す際の挙動が，図 3.6 においてどのように対応するかを説明する．まず，図 3.5 において，n 回目 の perform が実行されると，スケジューラは対応するハンドラ・ジェネレータである 図 3.6 の handler_T1 へ実行を切り替える．ハンドラ内の 1 回目の再開は，30 行目の runScheduler(resume, arg1) に対応する．この呼び出しにより _tryGen が再開され，try 節末尾まで実行が進むと，_tryGen は図 3.6 の 15 行目で yield TRY_DONE により停止する．スケジューラは afterTryDone > 0 を満たすため，runScheduler(resume, arg1) の呼び出し元であるハンドラへ制御を戻す．

次に，ハンドラ内の 2 回目の再開は，図 3.6 の 31 行目の runScheduler(resume, arg2) に対応する．このとき resume.performLabel は，当該ハンドラを呼び出した perform の label を保持している．例えば n 回目の perform によってハンドラが起動されたなら，performLabel = "ln" である．スケジューラはこのラベルを WHICH_LABEL として _tryGen に渡し，_tryGen は図 3.6 の 15 行目の停止位置から再開される．再開後は直後の switch を実行し，case "ln" に一致するため，23 行目の continue ln により ln: while の先頭へ巻き戻る．その結果，n 回目の perform 直後の処理 A_n が再び実行される．すなわち，n 回目の perform によって生成された限定継続が二回実行された．

このように，各 perform が自身を識別する label をスケジューラへ渡し，resume を通

```
1  try {
2    // n 回 perform するような処理
3    // 処理 A0;
4    let result_1 = perform("T1");
5    // 処理 A1;
6    let result_2 = perform("T1");
7    // ...
8    // 処理 An-1
9    let result_n = perform("T1");
10   // 処理An+1
11 } handle "T1" with {
12   resume(arg1);
13   resume(arg2);
14 }
```

図 3.5. `try` 節内で n 回 `perform` を実行するプログラム例

じてその `label` を再開時に利用することで，`try` 節中のどの位置から限定継続を再実行すべきかを決定できる．したがって，図 3.5 における n 回目以外の `perform` によって起動されたハンドラが二回目の `resume` を実行した場合であっても，同様に `label` に基づいて適切な `while` の先頭へ巻き戻すことで，対応する限定継続を正しい位置から再実行できる．

```
function* _tryGen() {
  // 処理 A0
  let {RESUME_ARG: result1} = yield PERFORM({
        type: "T1", label: "l1" });
  l1: while (true) {
    // 処理 A1
    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
        const {WHICH_LABEL,RESUME_ARG} = yield TRY_DONE();
        switch (WHICH_LABEL) {
          case "l1":
            result_1 = RESUME_ARG;
            continue l1;
          ...
          case "ln":
            result_n = RESUME_ARG;
            continue ln;
        }
      }
    }
  }
}
function* handler_T1(resume) {
  runScheduler(resume, arg1);
  runScheduler(resume, arg2);
  yield TRY_DONE();
}
yield RUN_TRYGEN({
  tryGenerator: _tryGen(),
  handlers: [{ T1: handler_T1 }]
})
```

図 3.6. 図 3.5 のプログラムを提案手法に基づいて変換した例

3.3 関数呼び出しの変換

エフェクトハンドラでは，`try` 節において，`perform` を内部で実行するような関数を呼び出すことができる．関数内部で `perform` が実行された場合，`perform` から関数の末尾までの処理を A，関数呼び出しから `try` 節の末尾までの処理を B とすると，`perform` によってハンドラに渡される限定継続は [A, B] となる．

前節までは，`perform` を `yield` に変換し，`yield` 以後の限定継続全体をラベル付き `while` 文で囲むことにより，同一の限定継続の複数回実行を実現していた．しかし，関数内部に `perform` が存在する場合，関数呼び出しによって限定継続が A と B に分断されるため，限定継続全体を 1 つの `while` 文で囲むことはできない．

そこで本提案では，`perform` から関数の末尾までの処理 A と，関数呼び出しから `try` 節の末尾までの処理 B を，それぞれ個別にラベル付き `while` 文で囲む．そして，限定継続を実行する際には，スケジューラが関数に対応するジェネレータと `try` 節に対応するジェネレータを，この順に呼び出す．これにより，同一の限定継続の 2 回目以降の実行についても，処理 A および処理 B をそれぞれ `while` 文によって順に巻き戻すことで実現可能となる．ある継続を分割して実行しているため，これは CPS 変換を応用した手法と捉えることもできる．

以下では，内部で `perform` を実行する関数 `foo` が定義され，`try` 節の中で `foo` が呼び出されているプログラムである図 3.7 と，その提案手法により変換されたプログラムである図 3.8 を用いて，具体的な変換方法を説明する．また図 3.9 は，スケジューラにおける，関数に対する処理に該当する．

以下では，内部で `perform` を実行する関数 `foo` が定義され，`try` 節の中で `foo` が呼び出されているプログラムである図 3.7 と，その提案手法により変換されたプログラムである図 3.8 を用いて，具体的な変換方法を説明する．また図 3.9 は，スケジューラにおける，関数に対する処理に該当する．

ここで，関数呼び出しに関するスケジューリングは，`RUN_FUNC` および `FUNC_DONE` という 2 種類の状態オブジェクトを介して表現する．`RUN_FUNC(obj)` は，引数として受け取ったオブジェクト `obj` に対し `obj["MESSAGE"] = "RUN_FUNC"` を設定して返す補助関数である．また `FUNC_DONE(result)` は，{`MESSAGE: "FUNC_DONE", result: result`} を返却する補助関数であり，スケジューラはこれらを手がかりに関数の起動と完了を識別する．

まず，図 3.7 の関数 `foo` は，図 3.8 に示すように，ジェネレータ関数 `fooGen` に置き換えられる．`fooGen` では，`perform` に対応する `yield PERFORM` 以降の処理全体が，ラベル付き `while` 文で囲まれている．また，図 3.7 における `return` 文は，図 3.8 では 9 ～ 14 行目の `FUNC_DONE` オブジェクトを返す `yield` とそれに続く `switch` 文へと置き換えられている．

図 3.7 の `try` 節に入り，3 行目で `foo` が呼ばれると，図 3.8 の 19 行目で `RUN_FUNC` オブジェクトが `yield` される．図 3.9 の 9 ～ 12 行目がスケジューラ内部で処理され，次に実行するジェネレータを示す `gen` に，渡されたジェネレータ `fooGen` を代入する．関数 `foo` に入り，図 3.7 の 3 行目で `perform` が実行されると，図 3.8 の 3 行目で `PERFORM` オブジェクト

```
1  function foo() {
2      // 処理 A0;
3      let result1 = perform("T1");
4      // 処理 A1;
5      return;
6  }
7  try {
8      // 処理 B0
9      foo();
10     // 処理 B1
11 } handle with "T1" {
12     resume(arg1);
13     resume(arg2);
14 }
```

図 3.7. 関数内部で perform するプログラム例

が yield される．これまでと同様，制御はスケジューラに移り，対応するハンドラが呼び出される．

図 3.7 の 12 行目でハンドラが resume を実行すると，スケジューラは fooGen の実行を再開する．その後，図 3.7 の 5 行目で return すると，図 3.8 の 9 行目で FUNC_DONE オブジェクトが yield される．これにより，スケジューラは，図 3.9 の 2 ～ 7 行目において，次に実行するジェネレータを， foo の呼び出し元であるジェネレータ _tryGen とする．try 節に制御が戻ると，図 3.7 の 10 行目で処理 B_1 が実行されて try 節が終了する．するとハンドラに制御が移り，2 回目の resume が呼び出され，perform 地点から try 節の末尾までの処理が再度実行される．

変換後のプログラムでは，まずスケジューラが図 3.8 の 9 行目の yield により中断している fooGen を再開する．fooGen が再開されると，13 行目の continue l1 によって，yield PERFORM 直後の処理 A_1 が実行され，最後に FUNC_DONE が yield される．するとスケジューラは，25 行目の yield により中断している _tryGen を再開する．_tryGen が再開されると，29 行目の continue L1 によって巻き戻り，処理 B_1 が実行され，TRY_DONE が yield される．

このように，関数内部で呼ばれた perform 以降の限定継続を実行する場合，変換後のプログラムでは，中断された関数に対応するジェネレータと try 節に対応するジェネレータを順に呼び出す．その結果，末尾で中断されたジェネレータがそれぞれ while 文によって巻き戻されて実行され，限定継続全体が実行される．

```
1  function* fooGen() {
2      // 処理 A0;
3      let {RESUME_ARG: result1} = yield PERFORM({
4        type: "T1",
5        label: "l1"
6      });
7      l1: while (true) {
8          // 処理 A1;
9          const { WHICH_LABEL, RESUME_ARG } = yield
               FUNC_DONE();
10         switch (WHICH_LABEL) {
11             case "l1":
12                 result_1 = RESUME_ARG;
13                 continue l1;
14         }
15     }
16 }
17 function* _tryGen() {
18     // 処理 B0
19     let result = yield RUN_FUNC({
20         generator: fooGen(),
21         label: "L1"
22     });
23     L1: while (true) {
24         // 処理 B1
25         const { WHICH_LABEL, RESUME_ARG } = yield TRY_DONE
               ();
26         switch (WHICH_LABEL) {
27             case "L1":
28                 result = RESUME_ARG;
29                 continue L1;
30         }
31     }
32 }
```

図 3.8. 図 3.7 のプログラムを提案手法に基づいて変換した例

```
1  // 関数が終了したときの処理
2  case "FUNC_DONE":
3    gen = gen.PARENT, arg = {
4        WHICH_LABEL: gen.label,
5        RESUME_ARG: value.result
6    }
7    continue;
8  // 関数呼び出しの処理
9  case "RUN_FUNC":
10   value.generator.PARENT = gen;
11   gen = value.generator, arg = null;
12   continue;
```

図 3.9. RUN_FUNC および FUNC_DONE に対するスケジューラの処理

3.4 制御構造の変換

ループ文や if 文の内部でも perform を実行できることから，適切な変換が必要となる．具体的には，ループ文は再帰的な関数呼び出しとして扱い，if 文は各分岐の本体を独立したジェネレータ関数に分解してスケジューラに実行させる．限定継続を分割し，対応するジェネレータを順に実行する点で，関数呼び出しの変換と同様，CPS 変換を応用した手法とも捉えられる．

3.4.1 ループ文

ループ文は反復のたびに同様の本体を実行するため，再帰呼び出しとして表現できる．そのため，本研究ではループ本体をジェネレータ関数として切り出し，反復をスケジューラによる再帰的な実行として実現する．以下では，内部で perform を呼び出す while 文を例として説明する．便宜上，変換前のプログラムにおける while 文を，本節ではループ文と呼ぶ．

ループ文の本体における perform 以降の処理を A，ループ文の終了後に続く処理を B とすると，perform によって生成される限定継続は，[A，それ以後のループ反復，B] という構造を持つ．関数呼び出しの場合と同様に，この限定継続全体を単一のラベル付き while 文で囲むことはできない．そこで，本研究ではループ文の実行を再帰的な関数呼び出しとみなし，関数呼び出しの場合と同様に，ループ本体を表すジェネレータ関数 loopGen に変換する．loopGen の各反復の冒頭でループ継続条件が偽となれば，FUNC_DONE を yield する．また，各反復の末尾では，新たに loopGen を生成し，スケジューラによって再帰的に実行する．これにより，スケジューラは二回目以降の限定継続の実行についても，関数呼び出しの場合と同様に，[A]，[各反復に対応するジェネレータ]，[B] として分割された形で実行することが可能となる．なお，do-while 文や for 文についても，制御構造の違いを吸収した上で，同様の変換を適用することができる．

図 3.10 は，ループ内部で perform を呼び出すプログラムであり，図 3.11 はその変換後のプログラムを示す．図 3.10 の 3 行目から 7 行目のループ文は，図 3.11 では 3 行目から 20 行目において，ジェネレータ関数 loopGen の定義と，yield RUN_FUNC に置き換わっている．また，図 3.10 におけるループ文以後の処理 B1 は，図 3.11 では 21 ～ 28 行にて while 文により囲まれている．loopGen 内部では，まず各反復の冒頭でループ継続条件 test が偽であれば，FUNC_DONE が yield される．これはループの終了を意味するため，制御は親である _tryGen に戻る．また，図 3.10 の 5 行目の perform は，図 3.11 の 6 行目の yield PERFORM に対応し，perform 以後の処理 A1 は while 文により囲まれる．またループ反復を進めるために，loopGen の末尾では 10 行目において，yield RUN_FUNC で次の反復に対応する loopGen() をスケジューラに渡している．これにより，loopGen を再帰的に呼び出すことが可能となり，ループの反復が擬似的に達成される．

次に，図 3.10 において i 番目の反復で perform されたエフェクトを捕捉したハンドラにお

```
 1 try {
 2     // 処理 B0
 3     while (test) {
 4         // 処理 A0
 5         perform("T1");
 6         // 処理 A1
 7     }
 8     // 処理 B1
 9 } handle with "T1" {
10     resume(arg1);
11     resume(arg2);
12 }
```

図 3.10. ループ内部で perform を呼び出すプログラム例

ける，2 回目の resume 呼び出しを考える．本変換では，resume はスケジューラの呼び出しに対応し，スケジューラは i 番目の反復に対応する loopGen ジェネレータを呼び出す．このとき，loopGen は図 3.11 の 10 行目で yield RUN_FUNC によって中断している．再開後，13 行目の continue l1 によって while 文の先頭へ巻き戻り，処理 A1 が実行される．続いて 10 行目の RUN_FUNC において，(i+1) 番目の反復に対応するジェネレータが生成され，スケジューラに渡されて中断する．以後も同様に (i+1) 番目以降の反復が順に実行され，最後に制御が _tryGen に戻る．

一方，_tryGen は 23 行目の TRY_DONE により中断しているため，二回目の実行ではそこから再開される．26 行目の continu L1 によって while 文の先頭に戻り，処理 B1 を実行した後，再び TRY_DONE で中断する．以上により，i 番目に perform が生成した限定継続の二回目の実行が完了する．

```
1  function* _tryGen() {
2      // 処理 B0
3      function* loopGen() {
4          if (!test) { yield FUNC_DONE; }
5          // 処理 A0
6          yield PERFORM({ type: "T1", label: "l1"});
7          l1: while (true) {
8              // 処理 A1
9              const { WHICH_LABEL, RESUME_ARG } = \\
10                 yield RUN_FUNC({ generator: loopGen()});
11             switch (WHICH_LABEL) {
12                 case "l1":
13                         continue l1;
14             }
15         }
16     }
17     yield RUN_FUNC({
18         generator: loopGen(),
19         label: "L1"
20     });
21     L1: while (true) {
22         // 処理 B1
23         const { WHICH_LABEL, RESUME_ARG } = yield TRY_DONE
               ();
24         switch (WHICH_LABEL) {
25             case "L1":
26                     continue L1;
27         }
28     }
29 }
```

図 3.11. 図 3.10 を提案手法に基づいて変換したプログラム

```
1  try {
2      // 処理 A0
3      if (test1) {
4          // 処理 A1
5          perform("T1");
6          // 処理 A2
7      } else {
8          // 処理 A3
9      }
10     // 処理 A4
11     return a
12 } handle with "T1" {
13     resume(arg1);
14     resume(arg2);
15 }
```

図 3.12. if 文内部で perform を呼び出すプログラム例

3.4.2 if 文

if 文や else 節の内部で perform が呼び出される場合も，ループ文と同様に，限定継続全体を単一の while 文で囲んで巻き戻すことはできない．そのため，本研究では，各分岐の本体をそれぞれ独立したジェネレータ関数として定義し，if 文の分岐内からそれらを呼び出す形に変換する．

図 3.12 は if 文の内部で perform を呼び出すプログラム例であり，図 3.13 はその変換後のプログラムを示している．各分岐の処理は，対応するジェネレータ関数 $\texttt{ifGen}_i$ として定義され，各分岐の内部から yield RUN_FUNC によって呼び出される．$\texttt{ifGen}_i$ が内部の yield FUNC_DONE によって終了すると，スケジューラは親である _tryGen を再開する．ここで，分岐内の処理が return や continue などのジャンプ文によって終了した場合には，対応する情報が親のジェネレータへと伝搬される．

また，図 3.13 において，7 行目の yield PERFORM によって生成された限定継続が二回目に実行される場合を考える．このとき，ifGen1 および _tryGen は図中の 10 行，31 行の yield でそれぞれ中断された状態にある．スケジューラは，これらのジェネレータを順に実行することで，分割された限定継続である処理 A2，処理 B1 を正しい順序で実行することができる．

```
1  function* _tryGen() {
2      // 処理 A0
3      let ifRes;
4      if (test1) {
5          function* ifGen1() {
6              // 処理 A1
7              yield PERFORM({ type: "T1", label: "l1" });
8              l1: while (true) {
9                  // 処理 A2
10                 const { WHICH_LABEL, RESUME_ARG } = yield
                       FUNC_DONE();
11                 switch (WHICH_LABEL) {
12                     ...
13                 }
14             }
15         }
16         ifRes = yield RUN_FUNC({
17             generator: ifGen1(),
18             label: "L1" });
19     } else if (test2) {
20         function* ifGen2() { /* 処理 A3 */ }
21         ifRes = yield RUN_FUNC({...});
22     }
23     L1: while (true) {
24         if (ifRes.jumpStatement == "return") {
25             const { WHICH_LABEL, RESUME_ARG } = yield
                   TRY_DONE();
26             switch (WHICH_LABEL) { ... }
27         } else if (ifRes.jumpStatement == "continue") {
28           ...
29         } else if ...
30         // 処理 B1
31         const {WHICH_LABEL,RESUME_ARG} = yield TRY_DONE();
32         switch (WHICH_LABEL) { ... }
33     }
34 }
```

図 3.13. 図 3.12 を提案手法に基づいて変換したプログラム

3.5 最適化

本研究で提案した変換では，perform や関数呼び出しなどの各段階で yield を用いたスケジューリングを行い，また関数やループ文，if 文といった制御構造を原則としてジェネレータ関数に変換している．しかし，yield の実行やジェネレータ生成は実行効率の悪化につながると考えられるため，yield やジェネレータ生成の回数をできるだけ削減することが望ましいと思われる．

本節では，ループ文・関数・if 文について，可能な限りネイティブな構文をそのまま実行することで，yield やジェネレータ生成の回数を削減する最適化を導入する．さらに，ハンドラについても可能な限りジェネレータではなく関数として実行できるようにする．以後，外部のハンドラに捕捉される可能性のある，A 内部で発生するエフェクトを簡単に「A が発生し得るエフェクト」と呼ぶことにする．

3.5.1 ループ文

本研究では，ループ文に入る直前に，静的変換で得られた情報に基づいて，ネイティブに実行するかジェネレータ関数として実行するかを判定する最適化を行う．3.4.1 節で述べたように，本研究ではループ文を再帰的なジェネレータ関数として実装している．この方法では，各反復ごとに yield をし，新たに生成されたジェネレータをスケジューラが呼び出すため，ネイティブなループ文を直接実行する場合と比べて大きなオーバーヘッドが生じる．

しかし，ループ文をジェネレータ関数に変換する必要がある場合は限定的である．具体的には，ループ文が発生するエフェクトによって渡された限定継続が，二回以上呼び出される可能性がある場合に限られる．そもそもループ文をジェネレータ関数に変換する理由は，ループ内の perform 式の地点から try 節の終端までに対応する限定継続全体を，ループ文のままでは巻き戻すことができないためである．しかし，その限定継続が高々一度しか実行されないことが保証される場合，制御を巻き戻す必要はない．それゆえ，本最適化が適用されたループ内の perform 式については，それ以後の処理を while 文で囲むといった変換が不要となる．

そこで，静的変換では図 3.14 に示すようなループ文を，図 3.15 のように変換する．具体的には，yield SingleShot で一度スケジューラに制御を戻す．スケジューラは，ループ文がマルチショット・エフェクトを発生し得るかどうかを判定し，その結果を真偽値を返す．返された真偽値について，真であればネイティブなループ文，偽であればループ文に対応するジェネレータ関数 loopGen が実行される．スケジューラは，静的変換において得られた二つの情報，(1) ループ文が発生し得るエフェクトの集合，(2) 各ハンドラがシングルショットであるか，に基づいて判定を行う．ただし，ループ文を囲うハンドラの配列 handlerArray も用いる．handlerArray を用いて，(1) の各エフェクトを捕捉するハンドラを取得し，それらのハンドラが全てシングルショットであるかどうかを，(2) の情報に基づいて判定する．全てシングルショットであれば中断されたジェネレータに真を，そうでなければ偽を渡して再開する．以下

```
1  while (test) {
2    body
3  }
```

図 3.14. ループ文を実行する単純なプログラム

```
 1  if (yield isSingleShot) {
 2    while (test) {
 3      body
 4    }
 5  } else {
 6    function* loopGen() {
 7      ...
 8    }
 9    yield RUN_FUNC({ generator: loopGen() })
10  }
```

図 3.15. 図 3.14 を提案手法により変換したプログラム

では，静的変換において (1) と (2)，それぞれどのように得るかについて説明する．

(1) ループ文が発生し得るエフェクトの集合

静的変換の際に得たい一つ目の情報は，任意のループ文における発生し得るエフェクトの集合である．この集合には，ループ内部で直接 `perform` されるエフェクトに加え，ループ内部で呼び出される関数を通じて発生し得るエフェクトも含まれる．ただし，ループ内部で確実に捕捉されるエフェクトは外部に伝播しないため，ここでは除外する．

図 3.16 を用いて，ループ文が発生し得るエフェクトとはなにかを示す．この例では，`for` 文の内部で `T1`，`T2`，`T3` の三種類のエフェクトが発生し得るが，`T1` はループ内のハンドラによって確実に捕捉されるため，当該ループが外部に対して発生し得るエフェクト集合は `T2` と `T3` からなる．

ループ文が発生し得るエフェクト集合は，変換前の構文木に対する静的解析によって求める．まず第一段階として，各関数定義およびループ文について，外部へとエフェクトを発生し得るような `perform` 式および関数呼び出しの集合を得る．具体的に，各関数定義およびループ文について，その外部へとエフェクトを発生し得る `perform` 式および関数呼び出しを収集するアルゴリズムを図 3.17 に示す．この疑似コードでは，各 `perform` 式および関数呼び出しから構文木上で親ノードを辿り，それらが属する関数定義およびループ文を記録する．ただし `perform` 式が，対応する `try` 節によって捕捉される場合には，それ以上外側には伝播しないため，そこで探索を打ち切る．このようにして，各関数およびループ文が直接含む `perform`

```
1  function ff() {
2      perform("T1")
3      perform("T3");
4  }
5  for (;;) {
6      try {
7          perform("T1");
8          ff();
9      } handle "T1" with {
10         resume();
11     }
12     perform("T2");
13 }
```

図 3.16. ループ内部で発生し得るエフェクトのうち，一部が内側のハンドラで捕捉される例

式および関数呼び出しの集合を得る．

次に，各関数について，その関数を呼び出した際に最終的に発生し得るエフェクトの集合を求める．関数内部で呼び出される他の関数が発生し得るエフェクトを再帰的に展開することで求める．この処理は深さ優先探索として実装でき，図 3.18 に示す疑似コードで表される．ここでは，エフェクト集合は重複を除くため集合として扱う．最終的に `effectsMap` と呼ばれる対応関係に，各関数が発生し得るエフェクト集合が記録される．

最後に，各ループ文 l について，内部で呼び出される関数が発生し得るエフェクト集合を `effectsMap` により展開することで，l が発生し得る全てのエフェクト集合 `loopEffects`$[l]$ を得る．この処理を図 3.19 に示す．

(2) シングルショット・ハンドラの保証

二つ目に得たい情報は，任意のハンドラが確実にシングルショットであるかどうかである．本研究では，変換前のコードに対する静的解析により，ハンドラがシングルショットであると判断できる十分条件を定める．

具体的には，ハンドラ内で `resume` が複数回呼び出されている場合や，`resume` が関数呼び出しの被呼び出し位置以外で使用されている場合には，限定継続が複数回実行される可能性があると判断し，保守的にマルチショットとみなす．また，`resume` がループ文や関数内部で呼び出される場合も同様にマルチショットとみなす．さらに，ハンドラ内に `perform` 式，あるいはエフェクトを発生し得る関数の呼び出しが含まれ，かつそれが `resume` 呼び出しよりも手前に存在する場合，そのハンドラはシングルショットでないとみなす．これは，当該 `perform` 式または関数呼び出し以降の処理が，二回以上実行される可能性があるためである．ここで，関数呼び出しがエフェクトを発生し得るかどうかは，後述する 3.5.2 節に基づき，その関数が

```
1  for (f of allFuncs) f.performingEffects = [];
2  for (l of allLoops) l.performingEffects = [];
3
4  for (x of allPerformsAndFunctionCalls) {
5      let p = parent(x);
6      while (p) {
7          if (isTry(p) && isPerform(x) && p.handles(x.effect
               )) {
8              break;
9          } else if (isFunc(p)) {
10             p.performingEffects.push(x);
11             break;
12         } else if (isLoop(p)) {
13             p.performingEffects.push(x);
14         }
15         p = parent(p);
16     }
17 }
```

図 3.17. 構文木を上向きに走査し，各関数定義およびループ文について，外部へとエフェクトし得る perform 式あるいは関数呼び出しを収集するプログラム

変換後のプログラムにおいてジェネレータ関数として定義されるかどうかと同値である．

if--else 文における resume 呼び出しについては，各分岐における最大呼び出し回数を用いて判定する．図 3.20 に示す例では，実行時にいずれか一方の分岐しか通らないため，resume は高々一度しか呼ばれず，シングルショットであると判断できる．これに対し，図 3.21 の例では，条件によっては resume が二回呼び出され得るため，マルチショットとみなす．

3.5.2 関数・if 文・ハンドラ

現在の変換では，全ての関数をジェネレータ関数に変換し，関数呼び出しおよび関数内部の return 文を yield に置き換えている．しかし，関数内部で perform が実行されず，かつエフェクトを発生し得る関数を呼び出さない場合には，その関数は計算の中断・再開を必要としない．

そこで本研究では，発生し得るエフェクト集合が空である関数については，ジェネレータ関数へ変換せず，ネイティブな関数としてそのまま定義・呼び出す最適化を行った．この判定には前述の effectsMap を用いる．effectsMap[f] が空である関数 f は，スケジューラを介

```
1  let effectsMap = new Map();
2  let visited = new Map();
3
4  for (f of allFunctions) {
5      effectsMap[f] = new Set();
6      visited[f] = new Set();
7      let stack = [...f.performingEffects];
8
9      while (stack.length > 0) {
10         let v = stack.pop();
11         if (isFunctionCall(v)) {
12             if (visited[f].has(v)) continue;
13             visited[f].add(v);
14             let next = effectsMap[v] ?? v.
                   performingEffects;
15             stack.push(...next);
16         } else if (isEffect(v)) {
17             effectsMap[f].add(v);
18         }
19     }
20 }
```

図 3.18. 関数呼び出しの依存関係を展開し，各関数が発生し得るエフェクト集合を計算する手続き

さず直接呼び出すことができ，yield やジェネレータ生成の回数を削減できる．ただし，変数に束縛された関数呼び出しについては，その関数がジェネレータ関数かネイティブな関数かを静的に判定できない場合がある．この場合には，関数呼び出しの直前に，その関数がジェネレータとして変換されたかどうかを動的に判別し，適切な呼び出し方法を選択する．

if 文・ハンドラにおける最適化も簡潔に述べる．これまでのアルゴリズムを用いれば，ある if 文・ハンドラがエフェクトを発生し得るかどうかは静的に容易にわかる．エフェクトを発生し得ない if 文・ハンドラは変換せず，それ以外は 3.4.2 節と同様の変換を行えばよい．

```
let loopEffects = new Map();

for (l of allLoops) {
    loopEffects[l] = new Set();
    for (v of l.performingEffects) {
        if (isFunctionCall(v)) {
            for (e of effectsMap[v]) {
                loopEffects[l].add(e);
            }
        } else if (isEffect(v)) {
            loopEffects[l].add(v);
        }
    }
}
```

図 3.19. 関数呼び出しを展開し，各ループ文が発生し得るエフェクト集合を計算する手続き

```
try {
    // ...
} handle "T1" with {
    if (test1) {
        resume()
    } else {
        resume()
    }
}
```

図 3.20. 分岐ごとに `resume` が一度しか実行されないためシングルショットと判定できる例

```
1  try {
2      // ...
3  } handle "T2" with {
4      if (test1) {
5          //
6      } else if (test2) {
7          resume();
8          resume();
9      }
10 }
```

図 3.21. 条件によって resume が複数回実行され得るためマルチショットと判定される例

3.6　図 2.8 の変換結果と実行追跡に基づくケーススタディ

本節では，図 2.8 のマルチショット・エフェクトハンドラの例を対象として，提案手法による変換結果を示し，その実行を追跡する．対象プログラムでは，`perform("getNum")` により捕捉された限定継続が，`resume(0)` と `resume(10)` によって 2 回再開される．その結果，`handledFunc` の計算 `8 + perform("getNum")` が 2 回評価され，8 と 18 がこの順に出力されることが期待される．

変換後のプログラムを図 3.22 に示す．ただし行数の都合上，`DivByZeroMultiShot` の定義は図 3.23 で参照するものとする．まず，図 3.22 の末尾にある `runScheduler({nextGen: main()})` により，スケジューラは `main` の実行を開始する．`main` は，変換後の関数 `DivByZeroMultiShot` と `handledFunc` を定義した後，17 行目で `RUN_FUNC` を `yield` して `DivByZeroMultiShot(handledFunc)` の実行を要求する．ここで `label: "L1"` を付与しているため，`main` 側でも `L1: while(true){...}` と `switch` により，`RUN_FUNC` の直後への巻き戻しが可能となる．この `while/switch` による構造が，同一の限定継続を複数回実行するための基盤となる（第 3.2 節）．

次に，図 3.23 の `DivByZeroMultiShot` の内部では，`try` 節に対応するジェネレータ `_tryGen` を定義し，23 行目で `RUN_TRYGEN` を `yield` することで，ハンドラ getNum_handler と `return_handler` をスケジューラに登録した上で `_tryGen` の中身を実行する（第 3.1 節）．`_tryGen` は 3 行目で `action()` を `RUN_FUNC` により起動し，その結果を `TRY_DONE(result)` として `yield` する．ただし，今回 `action` は `handledFunc` を示す．このときも `label: "l1"` を用い，`l1: while(true){...}` により `TRY_DONE` の直前地点へ巻き戻せるようにしている．

図 3.22 で `handledFunc` の本体に注目すると，`perform("getNum")` は `PERFORM({type: "getNum", label: "l1"})` を `yield` する形に変換される（第 3.1 節）．スケジューラはこの `PERFORM` を受け取り，図 3.23 の 16 ～ 19 行目で定義された，対応するハンドラ `getNum_handler` を選択して実行する．ただし，`getNum_handler` はエフェクトを発生しないことから，最適化により通常の関数として定義されている（第 3.5 節）．`getNum_handler` は `resume` を引数に取り，`runScheduler(resume, 0)` および `runScheduler(resume, 10)` を順に呼び出す．これにより，`perform` 地点から末尾までの限定継続が 2 回再開される．以下では，2 回の再開がどのように実現されるかを，1 回目の `resume(0)` と 2 回目 の `resume(10)` の順に追跡する．

1 回目の再開では，`getNum_handler` から `runScheduler(resume, 0)` が呼ばれると，スケジューラは `handledFunc` の `PERFORM` 直後へ制御を戻し，`result` に `0` を渡して再開する．その後，`handledFunc` は 図 3.23 の 9 行目で `FUNC_DONE(8 + result)` を `yield` するため，`8 + 0` が計算され，`FUNC_DONE(8)` が発生する．すると，スケジューラにより図 3.23 の `_tryGen` が再開され，3 行目の `result` に `8` が渡される．8 行目の `yiel TRY_DONE` によりそのまま値 8 がスケジューラに渡され，`runScheduler` の返り値として `getNum_handler`

の console.log により 8 が出力される．ここで，handledFunc における yield PERFORM 以降の限定継続は，handledFunc と _tryGen を順に呼ぶことで実行されたことがわかる（第 3.3 節）.

2 回目の再開では, 図 3.23 の 18 行目で getNum_handler が続けて runScheduler(resume, 10）を呼ぶ．ここで，handledFunc は既に 1 回目の再開でジェネレータの末尾で中断している．提案手法では，handledFunc 内に導入された l1: while(true){...} と switch(LABEL）により，再開ラベル "l1" が渡された場合に result = RESUME_ARG を代入して continue l1 することで，図 3.22 の 9 行目における FUNC_DONE(8 + result) へと制御を巻き戻す．その結果，result に 10 が設定された状態で 8 + 10 が再評価され，yield FUNC_DONE(18) が発生し，console.log により 18 が出力される.

以上より，本ケーススタディでは，perform により捕捉された同一の限定継続が，resume によって複数回再開される状況において，提案手法が while/switch による巻き戻し機構を用いて同一地点からの再実行を可能にすることを具体例で確認した．さらに，この巻き戻し機構により，ジェネレータを複製することなくマルチショット再開を実現できる点が明確になった.

```
1  function* main() {
2    function* DivByZeroMultiShot(action) { /* 図.3.23 を参照
         */ }
3    function* handledFunc() {
4      let {RESUME_ARG: result} = yield PERFORM({
5        type: "getNum",
6        label: "l1"
7      });
8      l1: while (true) {
9        const { LABEL, RESUME_ARG } = yield FUNC_DONE(8 +
           result);
10       switch (LABEL) {
11         case "l1":
12           result = RESUME_ARG;
13           continue l1;
14       }
15     }
16   }
17   let {RESUME_ARG: result} = yield RUN_FUNC({
18     generator: DivByZeroMultiShot(handledFunc),
19     label: "L1"
20   });
21   L1: while (true) {
22     const { LABEL, RESUME_ARG } = yield FUNC_DONE();
23     switch (LABEL) {
24       case "L1":
25         result = RESUME_ARG;
26         continue L1;
27     }
28   }
29 }
30 runScheduler({nextGen: main()});
```

図 3.22. 図 2.8 を変換したプログラムの全体像

```
1  function* DivByZeroMultiShot(action) {
2    function* _tryGen() {
3      let {RESUME_ARG: result} = yield RUN_FUNC({
4        generator: action(),
5        label: "l1"
6      });
7      l1: while (true) {
8        const { LABEL, RESUME_ARG } = yield TRY_DONE(result)
           ;
9        switch (LABEL) {
10         case "l1":
11           result = RESUME_ARG;
12           continue l1;
13       }
14     }
15   }
16   function getNum_handler(resume) {
17     console.log(runScheduler(resume, 0));
18     console.log(runScheduler(resume, 10));
19   }
20   function return_handler(resume, x) {
21     return x;
22   }
23   let {RESUME_ARG: result} = yield RUN_TRYGEN({
24     tryGenerator: _tryGen(),
25     handlers: [{getNum: getNum_handler, return:
          return_handler}],
26     label: "L1"
27   });
28   L1: while (true) {
29     const { LABEL, RESUME_ARG } = yield FUNC_DONE(result);
30     switch (LABEL) {
31       case "L1":
32         result = RESUME_ARG;
33         continue L1;
34     }
35   }
36 }
```

図 3.23. ハンドラを定義する DivByZeroMultiShot

第 4 章

実験

本章では，提案手法を JavaScript 上で実装した処理系 `geneffjs` に対して，実装の正当性および実行性能の観点から評価を行う．正当性に関しては，マルチショット・エフェクトハンドラを言語機構として備えるプログラミング言語 Koka と出力を比較し，提案手法が意図した挙動を満たすかを確認する．実行性能に関しては，CPS 変換に基づく既存の実装と比較する．以降，まず `geneffjs` の実装について述べ，その後，Koka との比較による正当性検証，および実行速度評価について順に示す．

4.1 `geneffjs` の実装

著者は，提案手法を JavaScript に実装した処理系 `geneffjs` を作成した．`geneffjs` は，エフェクトハンドラを含む拡張 JavaScript プログラムを入力として受け取り，提案手法に基づく同値なネイティブ JavaScript プログラムへと変換するトランスパイラと，その実行を支えるスケジューラから構成される．トランスパイラの実装には Babel を用いた．

トランスパイラにおける構文解析については，構文が近いことから既存ライブラリ `effectsjs`[25] の実装を参考にした．`effectsjs` は，ジェネレータ関数を用いてシングルショット・エフェクトハンドラを実装する JavaScript ライブラリである．`geneffjs` のスケジューラは，JavaScript におけるジェネレータ関数を用いたシングルショット・エフェクトハンドラ実装を解説した既存の実装例 [26] を参考にしつつ，本研究の変換規則に合わせて新たに設計・実装した．

4.2 Koka の動作との比較

本節では，本研究で提案した実装 `geneffjs` の正当性を検証するため，マルチショット・エフェクトハンドラを言語機構として備えるプログラミング言語 Koka と動作を比較する．ただし，2025 年 7 月 22 日にリリースされた Koka v3.2.2 を用いた．比較では，エフェクトハンドラを用いた同値なサンプルプログラムを Koka および `geneffjs` の両方で実行し，標準出力が文字列として完全に一致するかを確認する．

4.2.1 サンプルプログラム

比較に用いたサンプルプログラムは，以下の四つのカテゴリから構成されている．

- Koka の公式ドキュメント [27] に掲載されているエフェクトハンドラを用いたプログラム例
- Pretnar らによるエフェクトハンドラの解説論文 [28] に含まれる例
- Koka 処理系のテストに使われるプログラム [29] のうち，エフェクトハンドラに関連するもの
- 著者が作成した独自のサンプルプログラム

それぞれの内訳は，Koka チュートリアル由来のプログラムが 10 個，Pretnar 由来のプログラムが 7 個，Koka 処理系のテスト由来のプログラムが 22 個，独自作成のプログラムが 1 個であり，合計 40 個のサンプルプログラムを用いた．ただし，Koka の公式ドキュメントあるいは処理系のテスト用プログラムに含まれるサンプルプログラムのうち，本研究の実装で対応していない shallow handlers や mask などの機能を含むプログラムは，比較対象から除外した．また，`geneffjs` に移植した結果，実質的に同値のプログラムとなるものについては，代表的な一例のみを用いた．各サンプルプログラムの名称と，確認したい性質・挙動の概要を表 4.1 に示す．

表 4.1: 実験に使用したサンプルプログラム一覧

ID	プログラム名	性質・確認事項
Koka Tutorial		
S01	ask_const	Reader エフェクトの基本動作．
S02	ask_once	Reader エフェクト．限定継続を破棄し，計算を途中で中断する動作の確認．
S03	choice_all	非決定性計算の基本動作．
S04	state_choice	ハンドラの合成．state エフェクトを外側，非決定性を内側に配置した際の相互作用．
S05	choice_state	ハンドラの合成．非決定性を外側，state エフェクトを内側に配置した際の相互作用．
S06	emit	Writer エフェクト．副作用をリストに蓄積し，最終的に集約する処理の検証．
S07	raise_const	限定継続を破棄し，計算を途中で中断する動作の確認．
S08	state	標準的な state エフェクト．再帰呼び出しを伴う計算における状態遷移の正確性．
S09	raise_state	ハンドラの合成（例外 + state エフェクト）．例外発生時に外側のハンドラへ正しく制御が移るかの検証（multi-prompt の検証）．
S10	state_raise	ハンドラの合成（state エフェクト + 例外）．例外発生時，内側の計算のみが中断され，状態が維持されるかの確認．

次ページに続く

表 4.1（続き）

ID	プログラム名	性質・確認事項
Pretnar		
P01	backtrack	バックトラッキングを伴う探索．捕捉した継続を，新たなハンドラで囲って再開した際の挙動の確認．
P02	collect_reverse	副作用の実行順序の制御．reverse ハンドラにより，限定継続を再開したあとに自身の処理を行うことで，出力の順序を逆転させる．
P03	collect2_reverse	関数を返すハンドラ．CPS 的な状態伝搬をエフェクトハンドラで実現できるかの確認．
P04	collect	副作用の集約．計算の途中で発生した print エフェクトを捕捉し，最終的な戻り値の一部とする．
P05	alwaysread	Reader エフェクトの基本動作．
P06	picktrue	decide エフェクトに対し，常に true のみを選択する単純な探索．
P07	pickmax	全探索と集計．decide エフェクトに対し両方の分岐を探索し，それぞれの継続から得られた結果を比較して最大値を選択する．
Koka Testcases		
T01	effs1a_1	純粋な非決定性計算．複数回の選択と，発生し得る全結果のリスト集約．
T02	effs1a_2	ハンドラの合成．state エフェクトを扱うハンドラが外側にあり，全ブランチが一つの状態を共有する動作の検証．
T03	effs1a_3	ハンドラの合成．ブランチごとに状態が独立する動作の検証．
T04	effs1b	シンプルな非決定性計算．
T05	effs3	基本的な state エフェクト．
T06	effs5	state エフェクトと非決定性計算の合成．
T07	linear1	基本的な state エフェクト．
T08	linear2	state エフェクトとマルチショットの組み合わせ．一つのハンドラ内で resume を 2 回呼び出し，かつ再開の間に状態を再設定する．
T09	linear3	state エフェクトと継続の破棄の組み合わせ．
T10	nim_perfect1, 2	石取りゲーム（Nim）を 2 プレイヤーの相互再帰で実装したもの．ただし，お互い最適な戦略を用いた場合．
T11	nim_gt	継続を用いた木構造の構築．ゲームを全探索した結果を木構造としてキャプチャするもの．
T12	nim_check	エフェクトの委譲．ハンドラ内でさらに別のエフェクトを発生させる動作の確認．
T13	nim_pc1, 2	プレイヤーごとに異なる戦略（ハンドラ）を適用し，特定のルール下でのゲーム実行を検証．
T14	nim_choose	choose エフェクトにより，異なるハンドラ構成を非決定的に切り替えて結果を収集．
T15	noyield1, 2	あるハンドラから別のエフェクトを発生させる動作の検証．
T16	scoped1	ナップサック問題．非決定性計算を用いた典型的な探索アルゴリズムの正当性確認．
T17	scoped2	state エフェクトを扱うハンドラが外側にあり，全探索ブランチで単一のカウンタを共有する挙動の確認．
T18	scoped3	探索ブランチごとに独立した状態を持つ構成．

次ページに続く

表 4.2. 出力の不一致が認められたケースと原因の分類

ID	プログラム名	該当する原因カテゴリ
S05	choice_state	ケース 1：限定継続再開時の状態復元不可
P01	backtrack	ケース 2：限定継続内のハンドラ捕捉失敗
T03	effs1a_3	ケース 1
T18	scoped3	ケース 1
U01	generator	ケース 3：ジェネレータの中断地点より後の地点からの再開不可

表 4.1（続き）

ID	プログラム名	性質・確認事項
T19	scoped4	探索に枝切り（cut）を組み合わせた場合の挙動の確認.
独自作成		
U01	tree_generator	二分木の走査において，複数地点で限定継続を外部変数に保存し，その後，保存した順序とは異なる順で複数の継続を再開するプログラム.

4.2.2　結果と不一致が起きた原因

40 個のサンプルプログラムのうち，35 個については Koka と `geneffjs` の出力が一致した．一方で，5 個のプログラムにおいて不一致が確認された．不一致が生じたプログラムについて原因を分析した結果，大きく三つのカテゴリに分類できることが分かった．表 4.2 に，不一致が確認されたプログラムと，対応する原因カテゴリを示す．以下では，不一致となった原因三つを詳細に説明する．

限定継続再開時の状態復元不可

一つ目のケースは，state エフェクトを含むプログラムにおいて，マルチショットで限定継続を再開した際に，状態が限定継続生成時の値に復元されない場合である．ただし，state エフェクトとは通常 `get` エフェクトと `set` エフェクトの二つのエフェクトからなり図 4.1 に示すような state ハンドラによって捕捉される．state ハンドラは状態変数 st を管理し，`get` を捕捉した場合，現在の st を限定継続に渡して再開する．そして `set` を捕捉した場合，与えられた値 i によって st を更新し，限定継続を再開する．

ここで，state ハンドラの外側に，別のエフェクト E を扱うマルチショット・ハンドラが存在する場合，意図しない挙動が生じることがある．例えば図 4.2 の `choice_state` 関数はそのような状況を示しており，ケース 1 で不一致が起きたサンプルプログラム `choice_state` を簡略化したものである．1 ～ 8 行目の `choice_all` 関数では，`choice` エフェクトを捕捉した際に，渡された限定継続を二回実行する `choice` ハンドラを定義している．27 ～ 33 行の `choice_state` は，この `choice` ハンドラで `state` ハンドラを外側から囲み，`state` ハンド

```
1  function state(init, action) {
2    let st = init;
3    try {
4      return action();
5    } handle "get" with {
6      return resume(st);
7    } handle "set" with ({i}) {
8      st = i;
9      return resume();
10   }
11 };
```

図 4.1. state エフェクトを扱うハンドラ

ラはさらに surprising 関数を囲って実行する．ただし，state ハンドラにおける st の初期値は 0 となる．

surprising 関数が実行されると，まず 10 行目で set エフェクトにより st を 1 に更新した後，11 行目で choice エフェクトを発生させる．このとき，発生地点から choice_all 関数における try 節終端までに対応する限定継続が，choice ハンドラに渡される．この限定継続は，choice ハンドラが外側から囲っている state ハンドラの状態 st を含む．

choice ハンドラが渡された限定継続を一回目に実行すると，surprising の 11 行目以降が再開され，get エフェクトが発生する．get ハンドラは現在の st の値を限定継続へ渡すため，ここでは $st = 1$ が渡される．その後，13 行目で set エフェクトが発生し，state ハンドラ内の st は 2 に更新される．surprising が終了し，それを囲っていた state ハンドラが終了すると，制御は外側の choice ハンドラにおける最後に resume を呼んだ 5 行に戻る．

次に，同じ 6 行目で二回目の限定継続が呼ばれる．このとき，st は限定継続生成時点の状態に復元された上で再開されるべきである．すなわち二回目の実行においても，12 行目の get エフェクトにより観測される st は 1 であることが期待される．しかし，geneffjs では，限定継続の再開に際して変数環境を過去のある時点へと復元する機構を持たないため，二回目以降の再開では $st = 2$ がそのまま観測されてしまい，Koka と出力が不一致となる．この問題は，限定継続を複数回再開する際に，限定継続生成時点の変数環境を保存・復元する必要があることに起因する．限定継続を生成するたびに変数環境全体を保存し，再開時に復元することで解決できる可能性はあるが，現状の geneffjs では対応できていない．

```
1  function choice_all(action) {
2    try {
3      return action();
4    } handle "choice" with {
5      resume();
6      resume();
7    }
8  }
9  function surprising() {
10   perform({type: "set", i: 1});
11   perform({type: "choice"});
12   console.log(perform({type: "get"}))
13   perform({type: "set", i: 2});
14   return;
15 }
16 function state(init, action) {
17   let st = init;
18   try {
19     return action();
20   } handle "get" with {
21     return resume(st);
22   } handle "set" with ({i}) {
23     st = i;
24     return resume();
25   }
26 };
27 function choice_state() {
28   return choice_all(
29     function(){
30       return state(0, surprising)
31     }
32   );
33 };
34 choice_state();
```

図 4.2. state ハンドラをマルチショット・ハンドラで囲んだ場合に，限定継続の再開時に状態 *st* が生成時点へ復元されず，Koka と異なる結果を生じるプログラム

```
1  function backtrack(action) {
2    try {
3      return action();
4    } handle "decide" with {
5      try {
6        return resume(true);
7      } handle "fail" with {
8       //
9      }
10   }
11 }
```

図 4.3. 受け取った限定継続を別のハンドラで囲って実行するプログラム

限定継続を囲ったハンドラによるエフェクトの捕捉失敗

二つ目のケースは，受け取った限定継続を別のハンドラで囲って再開する場合である．限定継続の呼び出し中に新たに発生したようなエフェクトは，本来そのような新たに囲ったハンドラによって捕捉可能である．しかし geneffjs では，新たに囲ったハンドラの存在は探索対象として扱われず，本来捕捉されるべきエフェクトが捕捉されない．その結果，実行時エラーとなるか，Koka と異なる制御の流れが発生する．

例えば図 4.3 は，サンプルプログラム backtrack における エフェクトハンドラを示す関数 backtrack を簡略化したものである．ハンドラは受け取った限定継続を，新たに定義した fail ハンドラで囲んだ状態で resume により再開する．再開後に fail エフェクトが発生した場合，新たに囲まれた fail ハンドラが捕捉するのが望ましい挙動である．しかし geneffjs では捕捉されず，エラーとなってしまう．ただし，この問題はハンドラ探索アルゴリズムを拡張することで解決可能であると考えられる．本研究ではこの点を将来的な改良点として位置づける．

ジェネレータの中断地点より後の地点からの再開不可

三つ目のケースは，複数箇所で限定継続を生成して外部に保存し，それらを生成順と異なる順序で再開するような場合において生じる．geneffjs では，複数箇所でエフェクトが発生する場合，各エフェクトを発生させるような perform および関数呼び出しを yield に置換し，それ以後の処理を while 文で囲む．二回目以降に限定継続を再開する際，通常，再開したい地点はジェネレータの中断地点より前方に位置しており，そのため while により巻き戻すことで目的の地点へ到達できる．しかし，限定継続の再開順序によっては，ジェネレータの中断地点よりも後方の地点から再開したい状況が生じ得る．このとき，中断地点から後方の任意の地点へ直接ジャンプすることはできないため，Koka と同一の挙動を再現できない．

以下では，この原因により不一致が生じたサンプルプログラム tree_generator を簡略化した例を用いて説明する．図 4.4 は変換前のプログラム，図 4.5 は変換後のプログラムである．

図 4.4 ではまず，contA という空リストを定義し，try 節に入る．なお，contA[$i-1$] には，try 節における i 番目の perform により生成された限定継続が格納される．12 ~ 14 行に示すように，contA に限定継続が保存されるたびに，直ちにその継続を実行することで，順に try 節内の処理を進め，最終的に try 節が終了する．その後，15 行目 の（contA[0]）() では 1 番目の perform により渡された限定継続が実行され，処理 A1 が再実行される．続いて，16 行目の（contA[2]）() では 3 番目の perform により渡された限定継続が実行され，処理 A3 が再実行される．

図 4.5 では, contA に保存される限定継続は, 中断されたジェネレータ _tryGen と, その再開地点を表すラベルの組として表現される．28 行目 における runScheduler(_tryGen, {"A": handler_A}) の呼び出しにより，_tryGen が実行され, 3 行目の yield PERFORM まで進んだ時点で，ラベル l1 と _tryGen が contA[0] に保存される．次に，29 行目 の runScheduler(contA[0]）により _tryGen が再開され，処理 A1 を実行した後，6 行目の yield PERFORM まで進み，ラベル l2 と _tryGen が contA[1] に保存される．同様に, 30 行目の runScheduler(contA[1]）によって_tryGen は 3 番目の yield PERFORM まで進み, ラベル l3 と _tryGen が contA[2] に保存される．さらに，31 行目の runScheduler(contA[2]）により，処理 A3 が実行された後，12 行目 の yield TRY_DONE において_tryGen が中断される．

その後，32 行目の runScheduler(contA[0]）により，12 行目の yield で中断している _tryGen にラベル l1 が渡され，continue l1 によって 5 行目まで巻き戻される．この巻き戻しにより，処理 A1 が再実行された後，ラベル l2 に対応する 6 行目の yield PERFORM において_tryGen が再び中断される．

ここで，33 行目の runScheduler(contA[2]）が呼び出されると，_tryGen にはラベル l3 が渡される．すなわち，_tryGen における 11 行以降を実行するべきである．しかし，この時点での _tryGen の中断地点は，それよりも前方にある 6 行目に位置しているため，対応する地点へと制御を移すことができない．このように，変換後のプログラムにおいて，ジェネレータが前方の地点で中断している状態で，後方の地点から再開したい状況が生じる場合，geneffjs では Koka と同一の挙動を実現できない．

```
1  let contA = []
2  try {
3    perform(A);
4    // 処理A1
5    perform(A);
6    // 処理A2
7    perform(A);
8    // 処理A3
9  } handle with A {
10   contA.push(resume)
11 }
12 (contA[0])()
13 (contA[1])()
14 (contA[2])()
15 (contA[0])()
16 (contA[2])()
```

図 4.4. 複数の限定継続を保存し，特定順序で再開するプログラム例

```
1  let contA = []
2  function* _tryGen() {
3      yield PERFORM({ type: "A", label: "l1"});
4      l1: while (true) {
5          // 処理A1
6          yield PERFORM({ type: "A", label: "l2"});
7          l2: while (true) {
8              // 処理A2
9              yield PERFORM({ type: "A", label: "l3"});
10             l3: while (true) {
11                 // 処理A3
12                 const { LABEL, RESUME_ARG } = yield
                       TRY_DONE();
13                 switch (LABEL) {
14                     case "l1":
15                         continue l1;
16                     case "l2":
17                         continue l2;
18                     case "l3":
19                         continue l3;
20                 }
21             }
22         }
23     }
24 }
25 function handler_A(resume) {
26     contA.push(resume);
27 }
28 runScheduler(_tryGen, {"A": handler_A})
29 runScheduler(contA[0]);
30 runScheduler(contA[1]);
31 runScheduler(contA[2]);
32 runScheduler(contA[0]);
33 runScheduler(contA[2]);
```

図 4.5. 図 4.4 を提案手法に基づいて変換したプログラム

4.3 CPS 変換による実装との実行速度の比較

本節では，提案手法に基づく実装 `geneffjs` の実行性能を評価する．具体的には，CPS 変換に基づくエフェクトハンドラ実装と実行時間を比較する実験を行った．

4.3.1 ベンチマークプログラム

実行速度の評価には，表 4.3 に示した合計 16 個のベンチマークプログラムを用いた．これらは，既存研究で用いられている代表的なベンチマーク [30] に加え，本研究の実装特性を評価するために著者が独自に作成したものから構成されている．また，各ベンチマークについては，実行時間がおおよそ 10 ms から 100 ms の範囲に収まるよう，入力サイズを調整した．

既存ベンチマークは，非決定性計算，state エフェクト，早期終了，非末尾再開など，エフェクトハンドラの代表的な利用形態を網羅している．一方，独自ベンチマークでは，ループ文内でのエフェクト発生，極端なマルチショット，深いハンドラ探索などを対象とする．

表 4.3: ベンチマークプログラムの概要と評価の主眼

プログラム名	出典	性質・評価の主眼
b.countdown	既存	頻繁なエフェクトの発生と限定継続の再開．
b.fibonacci	既存	エフェクトが発生しない，純粋な再帰．
b.generator	既存	木構造の走査と計算のサスペンド．
b.iterator	既存	再帰のたびにシングルショット・エフェクトが発生する関数呼び出し．
b.nqueens	既存	マルチショット継続による探索．
b.product_early	既存	関数の再帰呼び出しと限定継続の破棄の組み合わせ
b.resume_nontail	既存	再帰のたびにハンドラを生成．
b.tree_explore	既存	複雑な非決定性計算．
b.triples	既存	深い再帰を伴う数値探索．
o.for.sshot	独自	ループ内でのシングルショット・エフェクト発生．
o.for.try.fib	独自	ループ毎に，エフェクトの発生と重い再帰関数の呼び出しがある計算をハンドラで囲う．
o.for.try.multi	独自	ループ内で毎回 `try-handle` 節に入り，マルチショット・エフェクトを発生させる．
o.func.multi	独自	再帰毎にマルチショット・エフェクトを発生するような関数の呼び出し．
o.nest	独自	深い再帰の末端で `try-handle` 節に入る．
o.nest.resume	独自	フィボナッチ計算とマルチショット・エフェクトの組み合わせ．
o.try.multi	独自	マルチショット・エフェクトの複数回の発生．

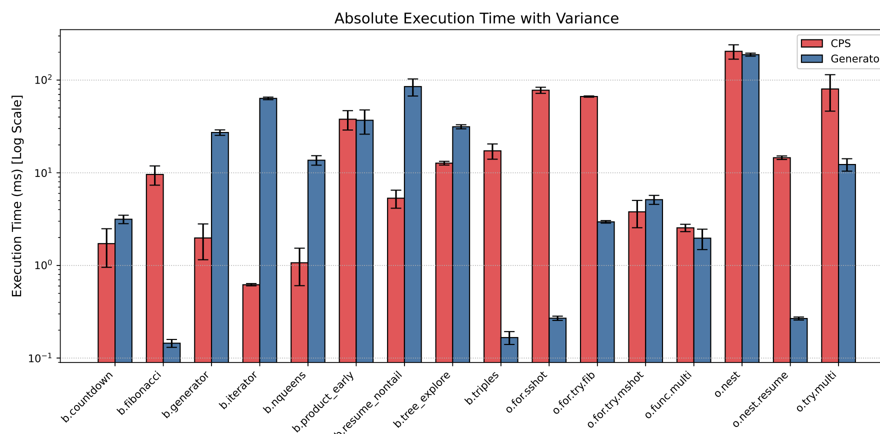

図 4.6. 実験結果のグラフ

4.3.2　実験環境と測定方法

実験は MacBook Air 上で実行した．使用した機材は Apple M1 チップ，16 GB メモリ，512 GB ストレージを搭載し，OS は macOS 15.3.1 である．実験プログラムはすべて Node.js v21.7.3 上で実行した．また，本実験における CPS 変換に基づくエフェクトハンドラ実装は，先行研究 [22] に記載されている変換規則に基づき，ベンチマークプログラムを著者が JavaScript に移植したものである．

測定は以下の手順で行った．まず，JIT コンパイラによる最適化を安定させるため，各ベンチマークプログラムを計測を行わずに 20 回実行した．その後，同一条件で 100 回実行し，得られた実行時間のうち上位 3 件および下位 3 件を除外した残りのデータから，平均値および分散を算出した．

4.3.3　結果

図 4.6 と表 4.4 は，それぞれ結果を示したグラフと表である．全 16 個中 7 個のベンチマークで `geneffjs` の方が高速であり，4 個で同等であり，5 個で低速であった．また，全体として，CPS 変換による実装の実行時間と `geneffjs` による実行時間の比率は 0.01 倍 ～ 289.32 倍となった．

特に，`o.for.sshot` は 289.32 倍，`b.triples` は 103.43 倍，`b.fibonacci` は 66.36 倍，`o.nest.resume` は 54.42 倍，`o.for.try.fib` は 22.50 倍と，CPS 変換による実装の方が大幅に遅くなるベンチマークが存在した．一方で，`b.iterator` は 0.01 倍，`b.generator` は 0.07 倍，`b.resume_nontail` は 0.06 倍など，逆に `geneffjs` の方が大幅に遅くなるベンチマークも存在した．

表 4.4. CPS 変換による実装と geneffjs の実行時間比較

Benchmark	CPS [ms]	Geneffjs [ms]	CPS / Geneffjs
b.countdown	1.72	3.15	0.55
b.fibonacci	9.60	0.14	66.36
b.generator	1.98	27.19	0.07
b.iterator	0.62	63.50	0.01
b.nqueens	1.07	13.68	0.08
b.product_early	37.88	36.88	1.03
b.resume_nontail	5.32	85.17	0.06
b.tree_explore	12.70	31.41	0.40
b.triples	17.24	0.17	103.43
o.for.sshot	77.88	0.27	289.32
o.for.try.fib	66.48	2.95	22.50
o.for.try.mshot	3.79	5.13	0.74
o.func.multi	2.55	1.97	1.29
o.nest	204.50	188.42	1.09
o.nest.resume	14.55	0.27	54.42
o.try.multi	80.36	12.30	6.54

4.3.4　cps 変換による実装との差の考察

CPS 変換による実装と `geneffjs` の実行速度差は，個々のベンチマークが継続を分割して実行する局面をどれだけ含むか，およびその分割された継続がどれだけ繰り返し実行されるかによって説明できると考えられる．以下では，`geneffjs` が CPS 的な実行形態を取らざるを得ない場合と，最適化によりネイティブな制御構造として実行できる場合を対比しつつ考察する．

本提案手法では，エフェクトを発生させ得る関数呼び出しや `if` 文，およびマルチショット・エフェクトを発生するループ文など，制御構造の内部でエフェクトが起こり得る箇所に対しては，CPS 変換と同様に計算を複数の継続へ分割し，それらを順に実行する形で意味論を実現している．このとき `geneffjs` では，分割された継続の境界で `yield` による中断が発生し，さらにスケジューラが次に実行すべき関数・ジェネレータを選択して呼び出す必要がある．

したがって，この種の局面が支配的なプログラムでは，CPS 変換における継続の生成・呼び出しによるオーバーヘッドに加えて，`yield` とスケジューラ呼び出しの追加オーバーヘッドが上乗せされる．その結果，`b.nqueens` や `b.iterator` のように，エフェクトが発生し得る関数呼び出しや `if` 文による分岐が頻繁に評価されるベンチマークでは，`geneffjs` が CPS 変換による実装よりも遅くなる傾向が現れる．

一方で，エフェクトを発生しない関数呼び出しが主となる場合，geneffjs では最適化により，それらをジェネレータ化せずネイティブ関数・ネイティブ分岐として実行できる．すなわち，継続の分割やスケジューラ介入が不要であり，JavaScript エンジンの最適化をそのまま享受できる．これに対して CPS 変換では，エフェクトを発生しない関数であっても，計算をクロージャの連鎖として記述する形になるため，元のプログラムに比べて呼び出し回数や割り当てが増え，オーバーヘッドが避けられない．例えば b.fibonacci では，呼び出される関数がいずれもエフェクトを発生しないにもかかわらず，CPS 変換側は継続の生成・呼び出しを繰り返すため，geneffjs が大きく有利となる

同様に，ループ文についても，発生し得るエフェクトが常にシングルショット・ハンドラにより捕捉される場合には，geneffjs ではループを変換せずネイティブのまま実行できる．この場合，CPS 変換ではループを再帰として表現する必要があるため，geneffjs が有利になりやすい．実際に o.for.sshot では，CPS 変換による実装の方が大幅に遅くなっている．

もう一つの重要な要因は，同一の限定継続が何回実行されるかである．一般に，再開回数が増えるほど geneffjs が有利になりやすい．これは，CPS 変換による実装では resume のたびに限定継続に対応するクロージャを生成して実行する必要があるのに対し，geneffjs では一度生成したジェネレータを再開し，内部で while 文による巻き戻しを用いて同一箇所からの再実行を実現できるためである．例えば図 4.7 に示す o.try.multi では，一つの try 節内で複数回 perform し，それぞれをマルチショットで繰り返し再開する．このプログラムは perform 自体も頻繁に評価されるため，前述の議論だけを見ると geneffjs に不利にも見えるが，各限定継続が繰り返し実行されることにより，CPS 変換側の，再開のたびのクロージャ生成コストが支配的になり，結果として geneffjs が有利となる．

以上より，geneffjs は継続分割が必要な局面では CPS 変換のオーバーヘッドに加えて yield・スケジューラのオーバーヘッドが生じるため不利になり得る一方，分割せずネイティブに実行できる局面や，同一限定継続を多数回再開する局面では有利になりやすい．したがって，図 4.6・表 4.4 に現れた結果のばらつきは，各ベンチマークがこれらの局面をどの割合で含むか，および再開回数がどの程度かの違いとして理解できる．

```
1  function f() {
2    try {
3      perform("a");
4      perform("a");
5      perform("a");
6      perform("a");
7      perform("a");
8    } handle "a" with {
9      resume();
10     resume();
11     resume();
12     resume();
13     resume();
14     resume();
15     resume();
16     resume();
17     resume();
18     resume();
19   }
20 }
21 f();
```

図 4.7. o.try.multi のベンチマークプログラム

第 5 章

まとめと今後の課題

本研究では，JavaScript においてマルチショット・エフェクトハンドラを可能な限り実用的な形で実現することを目的として，ジェネレータ関数を基盤とした実装手法を提案した．本章では，背景や提案手法， geneffjs の実装について整理し，実験結果を踏まえて得られた知見をまとめる．最後に，今後の課題について述べる．

エフェクトハンドラは，エフェクト発生地点からの限定継続を捕捉し，任意のタイミングで再開できる強力な制御機構である．とりわけマルチショット・エフェクトハンドラは，非決定性計算や探索，確率的プログラミングなどを簡潔に記述できる一方，限定継続の複製や管理を必要とし，実行効率の点で課題が生じやすい．JavaScript においては，CPS 変換に基づく実装がマルチショットに対応できる一方で，クロージャの生成・呼び出しの増加によるオーバーヘッドが大きい．一方，ジェネレータ関数を利用する実装は，言語組み込みの再開機構を活用できるため性能面の利点が期待されるが，一般にはシングルショットに制限される．本研究の動機は，これら既存手法のトレードオフを踏まえ，ジェネレータ関数を基盤としつつ，可能な限りマルチショット再開を実現することで，実行効率と表現力の両立を目指す点にあった．

提案手法では，`perform` によるエフェクト発生を `yield` に対応付け，限定継続を中断状態のジェネレータとして表現する．さらに，`yield` 直後の処理をラベル付き `while` 文で囲み，ジェネレータ末尾に制御用 `yield` を挿入することで，同一の限定継続を再度実行する際に，`while` による擬似的な巻き戻しを通じて実行位置を復元し，ジェネレータが本来備えていないマルチショット再開を可能な限り実現した．

また，提案手法の妥当性と性能特性を評価するため，Babel を基盤とした処理系 `geneffjs` を実装した．`geneffjs` は，`try-handle` や `perform` といった独自構文を含む拡張 JavaScript を入力とし，提案手法に基づくネイティブ JavaScript へ変換するトランスパイラと，変換後プログラムの実行を担うスケジューラから構成される．

本研究では，Koka との動作比較による正当性検証，CPS 変換実装との実行時間比較による性能評価，の二種類の実験を行った．以下では，それぞれの結果と得られた知見をまとめる．

正当性検証では，マルチショット・エフェクトハンドラを言語機構として備える Koka を参照実装とし，表 4.1 に示す 40 個のサンプルプログラムについて，Koka と `geneffjs` の標準出力が完全一致するかを確認した．その結果，35 個のプログラムで出力が一致し，提案手法

が意図した意味論を広範なケースで再現できることを確認した．

一方で，5 個のプログラムにおいて不一致が観測され，原因分析の結果，三つのカテゴリに分類できることが分かった．第一は，state エフェクトを含むプログラムにおいて，マルチショットで限定継続を再開した際に限定継続生成時点の変数環境が復元されない問題である．第二は，受け取った限定継続を別のハンドラで囲って再開する場合に，再開後に発生したエフェクトが本来捕捉されるべきハンドラで捕捉されない問題である．第三は，複数地点で生成した限定継続を生成順と異なる順序で再開する状況において，ジェネレータの中断地点より後方の地点から再開する必要が生じ，`while` による巻き戻しだけでは同一の制御を再現できない問題である．

これらは，提案手法がジェネレータの再開と巻き戻しによりマルチショット再開を擬似的に実現していることに由来する限界であり，現状の `geneffjs` が対応していないケースを明確にしたという点で，今後の改良に向けた重要な整理結果である．

実行性能評価では，CPS 変換に基づく既存実装と `geneffjs` の実行時間を比較し，図 4.6 および表 4.4 に示すように，ベンチマークによって高速化・低速化の双方が生じることを確認した．具体的には，全 16 個のベンチマークのうち 7 個で `geneffjs` の方が高速であり，4 個で同等，5 個で低速であった．すなわち，提案手法は常に CPS 変換より高速とはならず，プログラム構造に依存した性能特性を持つ．この性能特性は，`geneffjs` がどの程度継続分割を伴う実行を要求されるか，および同一限定継続がどの程度繰り返し実行されるかにより説明できる．エフェクトを発生させ得る関数呼び出しや `if` 文，マルチショット・エフェクトを発生するループ文など，制御構造の内部でエフェクトが起こり得る箇所では，提案手法も意味論を実現するために CPS 変換と同様に計算を分割して実行する．このとき `geneffjs` では，CPS 変換における継続生成・呼び出しのオーバーヘッドに加えて，`yield` による中断とスケジューラ呼び出しに伴う追加オーバーヘッドが発生する．したがって，その種の局面が支配的なプログラムでは，CPS 変換よりも遅くなり得る．

一方で，エフェクトを発生しない関数呼び出しや `if` 文が主となる場合，`geneffjs` では最適化によりそれらをジェネレータ化せず，ネイティブな関数呼び出しや分岐として実行できる．この場合，継続分割やスケジューラ介入を回避できるため，JavaScript エンジンの最適化の恩恵を受けやすく，CPS 変換に比べて有利になりやすい．同様に，シングルショットに限定してエフェクトを発生するループ文についても，提案手法では変換を回避しネイティブに実行できる場合があり，CPS 変換が再帰や継続の連鎖としてループを表現せざるを得ないのに対して有利となる．さらに，同一の限定継続を多数回再開するようなプログラムでは，`geneffjs` が有利になる傾向が確認された．CPS 変換では `resume` のたびに限定継続に対応するクロージャを新たに生成して実行する必要がある一方，`geneffjs` では一度生成したジェネレータを再開し，内部で `while` による巻き戻しを用いて実行すればよい．したがって，再開回数が増えるほど CPS 変換側の，再開のたびの生成コストが支配的になり，相対的に `geneffjs` が有利となる．

以上より，提案手法は，ネイティブに実行できる部分が支配的なプログラムや同一限定継続を多数回再開するプログラムにおいて有利になりやすい．一方で，継続分割を要する制御構造

が支配的なプログラムでは不利になり得る，という性能特性を持つことが明らかになった

今後の課題としては，以下の三点が挙げられる．

第一に，限定継続再開時の変数環境の保存・復元である．一部の状況におけるマルチショット再開においては，限定継続生成時点の環境を再現する必要があり，これを実現するためには，変数環境のスナップショットを保存するなどの方法が考えられる．ただし環境保存は実行コストやメモリ消費とトレードオフになるため検討が必要である．

第二に，限定継続を新たなハンドラで囲って再開するケースへの対応である．これは主としてハンドラ探索アルゴリズムの設計問題であり，スケジューラが「再開時点で有効なハンドラ」を適切に認識できるよう拡張することで改善できる．

第三に，実行性能の改善である．本研究の結果から，`yield` とスケジューラ呼び出しが継続分割局面における主要な追加オーバーヘッドとなることが示唆された．したがって，スケジューラの呼び出し回数の削減や `yield` の発生頻度を抑える変換などの実装上の最適化が有効であると考えられる．また，どのような構造のプログラムに対して提案手法が有利かを静的に判定し，不利な箇所は CPS 変換に委ねるといったハイブリッド方式の検討も今後の方向性として有望である．

発表文献と研究活動

(1) 武樋一樹，山崎徹郎，千葉滋．JavaScript における multi-shot Effect Handler のジェネレータベース実装の試み．日本ソフトウェア科学会 第 42 回大会, 2025.09.3-5

参考文献

[1] Gordon Plotkin and John Power. Algebraic operations and generic effects. *Applied Categorical Structures*, Vol. 11, pp. 69–94, 2003.

[2] Gordon Plotkin and Matija Pretnar. Handling algebraic effects. *Logical Methods in Computer Science*, Vol. 9, No. 4, pp. 1–36, 2013.

[3] Andrej Bauer and Matija Pretnar. Programming with algebraic effects and handlers. *Journal of Logical and Algebraic Methods in Programming*, Vol. 84, No. 1, pp. 108–123, 2015.

[4] OCaml Multicore Team. Multicore OCaml. `https://github.com/ocaml-multicore/ocaml-multicore`, 202 4. Accessed: 2026-01-08, archived repository.

[5] KC Sivaramakrishnan, Stephen Dolan, Leo White, Tom Kelly, Sadiq Jaffer, and Anil Madhavapeddy. Retrofitting effect handlers onto OCaml. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, pp. 206–221. ACM, 2021.

[6] Oleg Kiselyov and KC Sivaramakrishnan. Eff directly in OCaml. *Electronic Proceedings in Theoretical Computer Science (EPTCS)*, Vol. 285, pp. 23–58, 2018.

[7] Jonathan Immanuel Brachthäuser and Philipp Schuster. Effekt: extensible algebraic effects in Scala (short paper). In *Proceedings of the 8th ACM SIGPLAN Symposium on Scala*, pp. 67–72. ACM, 2017.

[8] Daan Leijen. Algebraic effects for functional programming. Technical Report MSR-TR-2016-29, August 2016.

[9] MDN Web Docs contributors. function* - JavaScript — MDN. `https://developer.mozilla.org/ja/docs/Web/JavaScript/Reference/Statements/function*`, 2025. Accessed: 2026-01-08.

[10] Babel Team. Babel web page. `https://babeljs.io/`, 2026.

[11] Olivier Danvy and Andrzej Filinski. Abstracting control. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, LFP '90, pp. 151–160. ACM, 1990.

[12] Carl Bruggeman, Oscar Waddell, and R. Kent Dybvig. Representing control in the

presence of one-shot continuations. In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation (PLDI)*, pp. 99–107, 1996.

[13] Eric W. Weisstein. Monty hall problem. `https://mathworld.wolfram.com/MontyHallProblem.html`.

[14] Daan Leijen. Implementing algebraic effects in C: "monads for free in C". In *Programming Languages and Systems*, Vol. 10695 of *Lecture Notes in Computer Science*, pp. 339–363. Springer, 2017.

[15] Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. Effect handlers for the masses. *Proceedings of the ACM on Programming Languages*, Vol. 2, No. OOPSLA, pp. 1–27, 2018.

[16] Oleg Kiselyov and Hiromi Ishii. Freer monads, more extensible effects. In *Proceedings of the 2015 ACM SIGPLAN International Symposium on Haskell*, pp. 94–105, 2015.

[17] Ohad Kammar, Sam Lindley, and Nicolas Oury. Handlers in action. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pp. 145–158, 2013.

[18] Ana Lúcia De Moura and Roberto Ierusalimschy. Revisiting coroutines. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, Vol. 31, No. 2, pp. 1–31, 2009.

[19] Roshan James and Amr Sabry. Yield: mainstream delimited continuations. In *Proceedings of Workshop on the Theory and Practice of Delimited Continuations*, pp. 1–12, 2011.

[20] Satoru Kawahara and Yukiyoshi Kameyama. One-shot algebraic effects as coroutines. In *Trends in Functional Programming*, Lecture Notes in Computer Science, pp. 159–179. Springer International Publishing, 2020.

[21] Daniel Hillerström, Sam Lindley, and Robert Atkey. Effect handlers via generalised continuations. *Journal of Functional Programming*, Vol. 30, No. e5, pp. 1–69, 2020.

[22] Daniel Hillerström, Sam Lindley, Robert Atkey, and KC Sivaramakrishnan. Continuation passing style for effect handlers. In *2nd International Conference on Formal Structures for Computation and Deduction (FSCD 2017)*, Vol. 84 of *Leibniz International Proceedings in Informatics (LIPIcs)*, 2017.

[23] Ocsigen Team. js_of_ocaml: Compiler from OCaml to JavaScript. `https://github.com/ocsigen/js_of_ocaml`, 2025. Accessed: 2026-01-08, commit b329014.

[24] Links Team. Links: Linking theory to practice for the web. `https://github.com/links-lang/links`, 2025. Accessed: 2026-01-08, commit 7100e28.

[25] Christopher Dieringer and Dave Campion. effectsjs: Algebraic effects in JavaScript. `https://github.com/effectsjs/effectsjs`, 2020. Accessed: 2026-01-08, commit ebd01e5.

[26] Yassine Elouafi. Algebraic effects in JavaScript (part 4) - implementing algebraic effects and handlers. https://dev.to/yelouafi/algebraic-effects-in-javascript-part-4---implementing-algebraic-effects-and-handlers 2018. Accessed: 2026-01-08.

[27] Daan Leijen. The koka programming language. https://koka-lang.github.io/koka/doc, 2025. Accessed: 2026-01-08.

[28] Matija Pretnar. An introduction to algebraic effects and handlers. Invited tutorial paper. *Electronic Notes in Theoretical Computer Science*, Vol. 319, pp. 19–35, 12 2015.

[29] Koka Team. Koka language test cases for algebraic effects. https://github.com/koka-lang/koka/tree/dev/test/algeff, 2026. Accessed: 2026-01-08.

[30] Effect Handlers Team. Koka benchmarks in the effect handlers benchmark suite. https://github.com/effect-handlers/effect-handlers-bench/tree/main/benchmarks/koka, 2023. Accessed: 2026-01-08, commit 1cf3237.

謝辞

本論文は，東京大学大学院情報理工学系研究科創造情報学専攻千葉研究室において行った研究をもとに，その成果をまとめたものである．

本研究を進めるにあたり，ご多忙の中にもかかわらず，毎週のミーティングを通して継続的かつ丁寧なご指導を賜った千葉教授に，深く感謝申し上げます．研究の方向性の定め方から，発表構成，論文執筆に至るまで，多岐にわたる貴重な助言をいただきました．

また，山崎助教にも心より御礼申し上げます．プログラミング言語やエフェクトハンドラに関する専門的知見に加え，研究内容を分かりやすく伝えるための説明の仕方など，研究活動全般において重要となる能力を養っていただきました．

最後に，日頃より有益な議論や助言をいただいた千葉研究室の皆様，ならびに鵜川准教授および鵜川研究室の皆様に深く感謝いたします．皆様の支えがあってこそ，本研究を成し遂げることができました．ここに記して，厚く御礼申し上げます．