Department of Creative Informatics
Graduate School of Information Science and Technology
THE UNIVERSITY OF TOKYO

Master's Thesis

A study of efficient effect handling in C++: Encoding tail-resuming as function returning

C++ における効率的なエフェクトハンドリングの研究：
末尾リジュームの関数リターンによる実装

Yuze Fu
傅 禹沢

Supervisor: Professor Shigeru Chiba

January 2025

Abstract

Algebraic effect handling is a superior abstraction for non-local control flows, unifying over the existing non-local control flow constructs such as try/catch, destructors, shared state, async/await and generators. To encourage the adoption of effect handlers, improving their performance is essential. Despite having a number of implementations, they lack a focus on tail resumptive handlers which leaves space for an improvement. We believe that tail resumptive handlers are invoked more frequently and contributes more to the overall performance of programs. The characteristic of them implies the possibility of an implementation with little overhead over function invocation. We propose eff-unwind, an implementation of effect handling as a C++ library which is optimized for tail resumptive handlers at the cost of others. Our implementation uses function calling and returning for tail resuming for improved efficiency while using stack copying and setjmp for general resuming and stack unwinding for yielding. It eliminates the need of recomposing the stack or preserving memory in the lifecycle of a tail resumptive handler at the cost of a less efficient yielding and non-tail resuming. Additionally, our library exposes a functional interface and preserves C++ destructor semantics. We evaluate our approach based on a total of 12 cases containing both tail resumptive handlers and non-tail-resumptive ones. The result shows about 1.5 - 3.2 times improvement for tail resuming and 1.9 - 73.9 times slowdown for non-tail-resuming. We also discover that multishot handlers presents challenges with C++ destructors.

概要

代数的エフェクトハンドリングは、非局所的な制御フローにおける優れた抽象化を提供し、try/catch、デストラクタ、共有状態、async/await、ジェネレータといった既存の非局所的制御フロー構造を統一する。エフェクトハンドリングの採用を促進するには、その性能向上が不可欠である。既存の実装は数多く存在するものの、末尾再開型ハンドラに焦点を当てたものは少なく、性能改善の余地がある。我々は、末尾再開型ハンドラがより頻繁に呼び出され、プログラム全体の性能に大きく寄与すると考えている。その特性は、関数呼び出しと同等の低オーバーヘッドで実現可能な実装の可能性を示唆している。そこで、我々は *eff-unwind* を提案する。これは、他のハンドラの性能を犠牲にして、末尾再開型ハンドラに最適化されたエフェクトハンドリングの C++ ライブラリ実装である。我々の実装では、効率向上のために末尾再開には関数呼び出しと戻りを利用し、一般的な再開にはスタックコピーと setjmp を、生成にはスタックアンワインディングを使用する。この実装により、末尾再開型ハンドラのライフサイクルにおけるスタックの再構成やメモリの保持が不要になり、その代わりに生成や非末尾再開の効率が低下する。さらに、我々のライブラリは関数型インターフェースを提供し、C++ のデストラクタのセマンティクスを保持する。我々は、末尾再開型ハンドラと非末尾再開型ハンドラの両方を含む合計 12 のケースでこのアプローチを評価した。その結果、末尾再開では約 1.5 倍から 3.2 倍の性能向上が見られた。一方、非末尾再開では 1.9 倍から 73.9 倍の性能低下が確認された。また、複数回実行型ハンドラが C++ のデストラクタにおいて課題を呈することも発見した。

Contents

Chapter 1

Introduction

Algebraic effect handling [1, 2] is a superior abstraction for non-local control flows. It is believed to be a unified abstraction over the existing non-local control flow constructs such as try/catch, destructors, shared state, async/await and generators, which additionally empowers new programming paradigms for probabilistic programming and more.

In order to promote adoption of effect handlers, it is important to optimize the performance of effect handling. Especially, we believe that it is important to optimize for tail resumptive cases of effect handling to achieve improved performance, since we hypothesis that this kind of handlers are more frequently invoked than the other kinds of handlers, which is based on observation of the discovered usecases of effect handling, and contributes more to the overall performance of programs. However, the existing implementation is not focused on the efficiency of tail resumptive handlers but prioritizes more complex usecases, which leaves space for improvement.

We introduce *eff-unwind*, a C++ library for effect handling that prioritizes the performance of **`raise`** and tail **`resume`** at the expense of **`yield`**. This trade-off enhances the overall performance of effect handling when most invocations involve tail-resumptive handlers. Our approach draws inspiration from exception handling, which optimizes for the normal execution path at the cost of the exceptional path. Similarly, we design raising and tail resumption to function like standard invocation and return, avoiding stack recomposition or closure creation to minimize overhead. This design incurs the cost of using stack unwinding for yielding, and supporting non-tail resumption to handle additional control flow operations in effect handling, achieved through stack preservation and the use of setjmp/longjmp.

We evaluate eff-unwind based on 11 cases from [3] and 1 case from [4], comparing against 3 existing implementations, *cpp-effects* [4], *Koka* [5] and *OCaml* [6], which we believe covers most of the explored usecases of effect handling and the state-of-the-art works. It shows that our implementation has about 1.5 - 3.2 times faster than SOTA works where invocations to tail resumptive handlers are taking the majority, while being 1.9 - 73.9 times slower than SOTA works for non-tail-resumptive handlers.

The main contributions of this thesis are:

- An implementation of effect handling containing different techniques for the different control-flow operations, which has a performance advantage based on the hypothesis that most effect raising invokes a tail-resuming handler.
- A C++ library supporting effect handling with a functional interface and incorporates well with C++.
- Identifying a problem with C++ destructors while multiple resumption is used which has not been reported by existing works.

The structure of this thesis is as follows: In Chapter 2, we first give a brief introduction to effect handling, the categorization of effect handlers and the existing implementation techniques, and then give an illustration of why we believe that tail resumptive handlers are important. In Chapter 3, we give a detailed explanation of our approach for the implementation of the control-flow operators, **`raise`**, **`resume`**, **`break`** and **`yield`**. In Chapter 4, we give the results of our evaluation and an analysis of the performance of our implementation. Finally, Chapter 5 concludes the thesis and gives discussion of future works.

Chapter 2

Improving the performance of effect handling

In this chapter, we first give a brief explanation of effect handling since it is a fairly new programming construct. We demonstrate the control flow of effect handling along with a variety of effect handing examples and the categorization of these examples. After that, we give a typical formal definition of effect handing as an extra background. Then we talk about the related works to this paper, including programming interfaces and implementation techniques.

After presenting the preliminary, we give an explanation for the motivation of the work in this thesis. It is observed that a specific kind of effect handlers, *tail resumptive* handlers, are supposed to contribute a larger proportion to performance. However, the existing works are mostly not focused on tail resumptive handlers and leaves room for improvement. We identifies that tail resumptive handlers can be implemented close to a function call and return which eventually improves the performance of effect handling if our previous statement holds.

2.1 Effect handling

Algebraic effect handling [1, 2] is a superior abstraction for non-local control flows. It provides a new programming language feature which could be used to express existing use-cases like exceptions, destructors, shared state and much more, while providing new methods for expressing probabilistic programs and others.

Elements of effect handling

In a typical program with effect handling, there are three constructs, *effect declaration* which declares the *name* of an effect with its *effect parameter* and *result type*, *do block* where the code to be executed is defined and effects could be raised and *effect handler*s inside the *handle block* which define the code to be executed when a specific effect is raised. The do block and handle block must come as a pair, forming a *do-handle block*. Additionally, there are 4 operators which are **`raise`**, **`resume`**, **`break`** and **`yield`** for manipulating control-flow.

For example, in Figure 2.1, the code uses effect handling to provide a random generator available within a scope, and the nested function uses the random generator to split the inbound web traffic to two different implementations for testing whether the new method `getUserIdNew` contains any bug. It contains the effect declaration on line 1, which declares an effect named as `choice` with no parameter but a result of type **`bool`**. On line 4 - 11, there is the do-handle block, where the do block spans from the **`do`** keyword

```
effect choice() -> bool

func handleRequest()
  res := do {
    userId := getUserId()
    response := getUserEmails(userId)
    break(response)
  } handle { choice() {
    r := resume(rand_bool())
    yield(r)
  } }
  write(res)

func getUserId()
  flag := raise choice()
  if flag { return getUserIdOld() }
  else { return getUserIdNew() }
```

Fig. 2.1: Random generator with effect handling

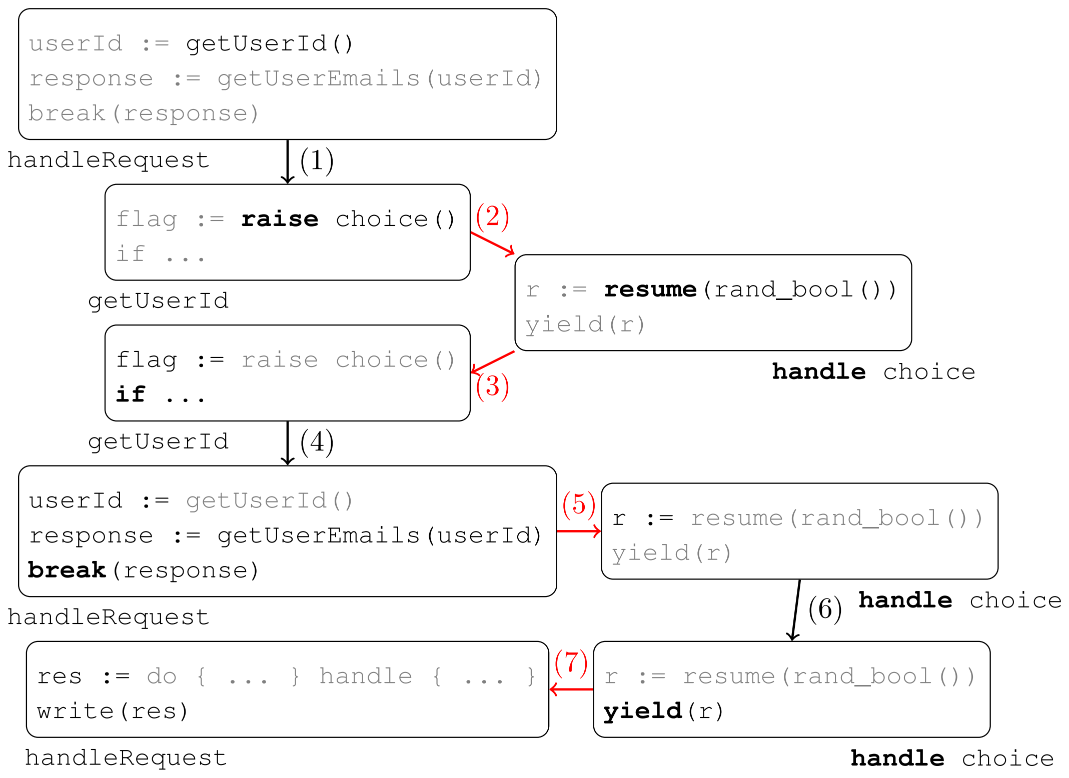

```
userId := getUserId()
response := getUserEmails(userId)
break(response)
```
```
flag := raise choice()
if ...
```
```
r := resume(rand_bool())
yield(r)
```
```
flag := raise choice()
if ...
```
```
userId := getUserId()
response := getUserEmails(userId)
break(response)
```
```
r := resume(rand_bool())
yield(r)
```
```
res := do { ... } handle { ... }
write(res)
```
```
r := resume(rand_bool())
yield(r)
```

Fig. 2.2: Control flow of random generator in Figure 2.1

getUserId
handleRequest*

Fig. 2.3: Call stack of random generator in Figure 2.1

on line 4 to the first closing brace on line 8, and there is an effect handler defined on line 8 - 11 spanning from the effect name `count` to the first closing brace on line 11.

The control-flow graph of the example is shown in Figure 2.2. When the function `handleRequest` is executed, it begins with the code defined in the do block on line 5 and calls the function `getUserId` which is marked with (1) in the control-flow graph, until it reaches a **`raise`** which raises an effect. If an effect is raised, the control flow is passed from the do block to the corresponding handler with the control-flow operators.

`raise` finds the nearest handler of the designated effect in the stack and executes it, which is similar to **`throw`** in exception handling. It passes a value called *argument* of the effect which is similar to the exception value passed to exception handlers.

For example, when code on line 15 of Figure 2.1 raises the effect `choice`, it finds downwards the callstack shown in Figure 2.3, and discovers `handleRequest` which defines the handler for the effect `count` (on line 8). It executes the code in the handler, which is the edge marked with (2) in the control-flow graph Figure 2.2.

Being a dual, **`resume`** passes the control flow back to the do block as if **`raise`** is a function and has just returned. It passes a value called *result* of the effect to the do block. For example, when code on line 9 of Figure 2.1 resumes back to the do block, it passes the control flow back to the point as if **`raise`** has returned, like shown with the edge (3) in the control-flow graph, and continues with the code in do block like edge (4).

Different from **`try`** block in exception handling, the do block explicitly ends with a **`break`** which provides the value of the do block. This operation passes the control flow back to the handler as if **`resume`** is a function and has just returned which continues the *remaining code* in handler after **`resume`**. The value of the do block is passed to the handler.

For example, when code on line 7 of Figure 2.1 marks the end of the do block, it passes the control flow back to the point as if **`resume`** has returned, and provides the value `response` to the assignment statement `r := resume(rand_bool())` like shown with the edge (5) in the control-flow graph.

Finally, the handler explicitly ends with a **`yield`** which provides the resulting value of the entire do-handle block. This is also different from exception handling since **`try-catch`** is not an expression and do not have a value. For example, when code on line 10 of Figure 2.1 yields, it marks the end of the do-handle block and provides the value `r` as the value of the block which is later assigned to the variables `res` of `handleRequest`, as shown with the edge (7) in the control-flow graph.

To sum it up, for the code shown in Figure 2.1, the handler is used to generate a random number, and the do-handle construct is used to provide this handler within the scope of the `handleRequest` function. This usage of effect handling enables decoupling an interface of random generator (the effect declaration of `choice`) with the implementation of it (the effect handler defined on line 8). The implementation of the handler could be swapped easily in `handleRequest` without changing any code in `getUserId` to switch to different random generating policies. Additionally, effect handling permits providing multiple different implementations for the same interface in different scopes of the program, which could not be replaced by a global function.

From the explanation, it shows the power of effect handling in two aspects. First, the handler can additionally continue with the do block and alternate the resulting value of it (despite in the example of Figure 2.1 it does not alternate). Second, the control-flow transitions are associated with a value which enables the exchange of values between the do block and the effect handler. Therefore, it is a generalized abstraction to replace many of the existing programming paradigms and provide new ways to express side effects or other cases, whose examples are later shown in subsection 2.1.1.

The terminology used in this paper is one of the typical terminology for effect handling.

```
1  effect count()
2
3  func runCounter() {
4    x, counter := do {
5      break (tarai(1, 2, 3), 0)
6    } handle { count() {
7      (x, counter) := resume()
8      yield (x, counter + 1)
9    } }
10   print(counter)
11   return x
12 }
13
14 func tarai(x, y, z) {
15   raise count()
16   if (x >= y) { return y }
17   else { return tarai(
18     tarai(x - 1, y, z),
19     tarai(y - 1, z, x),
20     tarai(z - 1, x, y)
21   ) }
22 }
```

Fig. 2.4: Counting function execution with effect handling (counter)

Different works or papers use slightly different terminologies, such as handle-with [7] instead of do-handle, but the core idea of effect handling and the basic elements are the same.

This paper introduces *deep handlers*, while there is a variant called *shallow handlers* [8]. For deep effect handlers, it is automatically re-registered after it handles the effect, which means it could handle subsequent effects. However, shallow handlers only handles the first effect performed, and thus cannot handle subsequent effects.

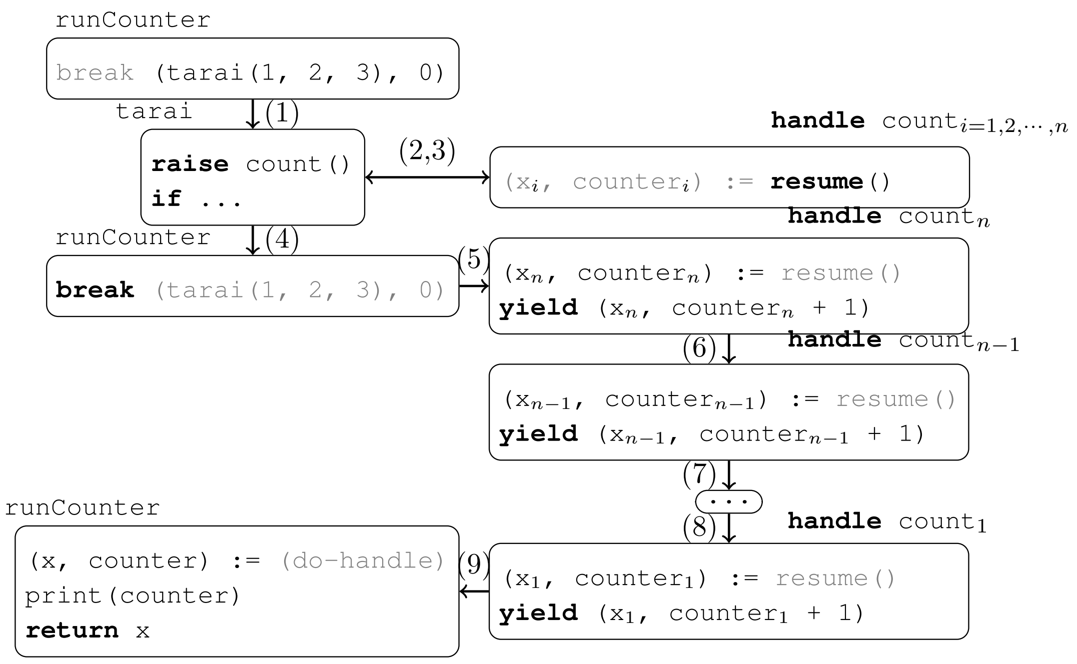
This paper discusses effect handling where the handlers are found dynamically at runtime. There are implementations where the handlers could be determined statically [9, 4].

Execution order of remaining code

It is a natural ask that what are the orders of executing the remaining code in the handlers if multiple handler are invoked. The answer is that it is executed in the reverse order of executing the handler. Additionally, the value of **`break`** is passed to the first remaining code, and the value of **`yield`** is passed from the previous remaining code to the next remaining code.

To illustrate this property, we demonstrate another example in *counter* Figure 2.4 [3] where the code uses effect handling to trace the execution count of the function `tarai`. It defines the effect `count` on line 1, and in the first line of `tarai` it raises the effect, so it expects an handler defined in the outer scope to provide a counting implementation.

The control-flow graph is given in Figure 2.5. The invocations to the handler and the variables `x`, `counter` inside the handler is annotated with subscript $1, 2, \cdots, n$ in the order of **`raise`**. When `runCounter` breaks with the value `(tarai(1, 2, 3), 0)`, the value is passed to the handler last invoked ($i = n$) as the edge (5) in the graph, and the handler



runCounter
break (tarai(1, 2, 3), 0)
tarai
(1)
raise count()
if ...
(2,3)
handle count$_{i=1,2,\cdots,n}$
(x$_i$, counter$_i$) := resume()
handle count$_n$
runCounter
(4)
break (tarai(1, 2, 3), 0)
(5)
(x$_n$, counter$_n$) := resume()
yield (x$_n$, counter$_n$ + 1)
(6)
handle count$_{n-1}$
(x$_{n-1}$, counter$_{n-1}$) := resume()
yield (x$_{n-1}$, counter$_{n-1}$ + 1)
(7)
...
(8)
handle count$_1$
runCounter
(x, counter) := (do-handle)
print(counter)
return x
(9)
(x$_1$, counter$_1$) := resume()
yield (x$_1$, counter$_1$ + 1)

Fig. 2.5: Control flow of counter

Term in Exception Handling	Term in Effect Handling
Exception Class	Effect Declaration
try	**do**
catch	**handle**
throw	**raise**
End of **try**	**break**
End of **catch**	**yield**

Table 2.1: Term correspondence between exception handling and effect handling

increases the second component of the pair by 1 to get $(tarai(1, 2, 3), 1)$, and yields the value to the next last invoked handler ($i = n - 1$) as the edge (6) in the graph. The yielded value is passed along the long chain of remaining code of the handlers, and each handler increases the second component (`counter`$_i$) by 1. Finally, the firstly invoked handler yields the result `x`$_1$ and `counter`$_1$, and provides the value as the result of the do-handle block back to `runCounter` as shown with the edge (9). As a result, the return value of `tarai(1, 2, 3)` is kept as-is, which means `x`$_i$ is all the same, while `counter`$_i$ is increasing stepping by 1 for $i = n, n - 1, \cdots, 1$, counting the invocation count of `tarai`.

Another point of view: resume as delimited continuation

Another point of view is that effect handling is an abstract over delimited continuation, where effect handling is mostly the same as exception handling while **`resume`** is added as a delimited continuation which stands for some suspended computation to be possibly continued later on. The terms in effect handling except resume have corresponding terms in exception handling as in Table 2.1. **`resume`** is provided additionally to the handler, which represents the computation to be executed from the returning point of raise until the end of the current do-handle block.

In order to illustrate this viewpoint, the previous example of counting function execution

```
effect count()

func runCounter() {
  x, counter := do {
    break (counter(3), 0)
  } handle { count() {
     (x, counter) := resume()
    yield (x, counter + 1)
  } }
  print(counter)
  return x
}

func counter(x) {
  raise count()
  if (x == 0) { return x }
  else { return counter(x - 1) }
}
```

Fig. 2.6: Simplified counter

in Figure 2.4 is simplified into *simplified counter*Figure 2.6. This time `tarai` is replaced by a simple `counter` function which uses recursion to count from `x` down to 0.

Every time the effect `count` is raised, the current execution inside the do block is suspended, and it creates an delimited continuation **`resume`** representing the computation to be executed from the returning point of raise until the end of the do-handle block. The corresponding effect handler is executed similarly to exception handling which fully gains the control, but it gets access to the **`resume`**, and may execute it at wish.

For instance, when the first time `count` is raised, the corresponding handler gets control and the code to be executed for evaluating do-handle block becomes as in Figure 2.7a. The remaining code in the do block are the rest recursions in `counter` on line 2 - 7 and the break statement in do block (which is transformed into a return statement since it is now in a function) on line 8, which forms the resume function on line 1 - 9. The variables are subscripted with the index of the corresponding handler invocation just as previously in Figure 2.5. The code transformation here only serves for illustration and may differ from the actual implementation of effect handling.

When the second time `count` is raised, things are a little bit different as the continuation should also include the remaining code of the first handler invocation since the end of do-handle evaluation has been changed to include the first handler, where the code to be executed are shown in Figure 2.7b. This time the continuation contains 4 parts, (1) the remaining code in `counter(2)` on line 7 - 12, (2) the remaining code in `counter(3)` on line 4 - 14, (3) the remaining code in do block on line 2 - 16 and (4) the remaining code in handler$_1$ of count on line 1 - 20.

2.1.1 Categorization of effect handlers

Effect handlers can be categorized into *tail-resumptive*, *non-tail-resumptive* and *yielding* based on their occurrence and location of **`resume`**.

```
// counter(3)'s remaining code
resume_1 := func() {
  func counter_1() {
    x_1 := 3 // captured
    if (x1 == 0) { return x_1 }
    else
    { return counter(x_1 - 1) }
  }
  return (counter_1(), 0)
}
// handler of count
(x_1, counter_1) := resume_1()
yield (x_1, counter_1 + 1)
```

(a) First raise

```
resume_2 := func() {
  func resume_1*() {
    // counter(3)'s remaining code
    func counter_1*() {
      x_1 := 3 // captured
      // counter(2)'s remaining code
      func counter_2() {
        x_2 := 2 // captured
        if (x1 == 0) { return x_2 }
        else
        { return counter(x_2 - 1) }
      }
      return counter_2()
    }
    return (counter_1*(), 0)
  }
  // handler of count
  (x_1, counter_1) := resume_1*()
  return (x_1, counter_1 + 1)
}
// handler of count
(x_2, counter_2) := resume_2()
yield (x_2, counter_2 + 1)
```

(b) Second raise

Fig. 2.7: Code of resume

Tail-resumptive handler

Tail-resumptive cases contains a special combination of `yield(resume(...))`, which is used in shared state, iterators and other cases. Recalling the introduction in section 2.1, **yield** marks the end of a handler, so chaining **resume** with **yield** means there is no remaining code in the handler, and therefore no code should be executed after **break** in do. This kind of handlers are used for creating a scope where a "function" is available. As an example of tail-resumptive handlers, the case *shared state* [7, 5] registers two handlers for accessing a shared value, whose pseudocode is given in Figure 2.8.

In Figure 2.8, there are two effects `get` and `set`, which is used for getting and setting the shared value correspondingly. The `countdown` function first declares a variable `state` which is intended for shared access. The do block accesses the shared value `state` by calling the two effects, decreases the value by 1 each iteration until reaching 0 and evaluates to the final value of the shared value. In the handle block, it registers two handlers for the two effects respectively, which resumes with the shared value and does not have additional code after the resumption. Differentiating from capturing the shared value by closure or by global variable, the usage of effect handlers provides well-defined and fine-grained control over the variable as the semantic scope of the handlers defined by do-handle block and it is possible for the handlers to add additional code to execute on accessing the variable.

```
effect get() -> int
effect set(int) -> int

func countdown(int n)
  state := n
  do { while true {
      i := raise get()
      if i == 0 { break i }
      else { raise set(i - 1) }
  } } handle { get() {
    yield(resume(state))
  } set(int i) {
    state = i
    yield(resume(state))
  } }
```

Fig. 2.8: Shared state with effect handling

```
1  effect divZero(int q)
2
3  func div(int d, int q)
4    do {
5      if q == 0
6        raise divZero(0)
7      break d / q
8    } handle { divZero() {
9      print("division␣by␣0")
10     yield(0)
11   } }
```

Fig. 2.9: Exception handling

Yielding handler

Yielding is another simpler case of effect handlers, where the handler does not contain any **`resume`** but only a single **`yield`** like in Figure 2.9 [5, 4]. Recalling the introduction in section 2.1, the extra power of effect handling over exception handling is **`resume`**, so this kind of handler is nothing more than a exception handler, and its discovered usage so far is only exception handling.

In Figure 2.9, there is an effect `divZero` which stands for a division-by-zero error. To add some context to the error, the dividend is passed as the effect argument. On line 5 - 6, if the divisor is 0, it raises the effect similarly to throwing an exception, and the effect handler on line 8 - 10 is executed similarly to catching an exception. The handler prints an error log and provides a fallback value 0 as the result in case the error happens, without resuming line 7 since that division is invalid.

```
effect ray() -> vector3

func rayCast(pos: vector2)
  do { while true {
      ray := raise ray()
      break render(ray, pos)
  } } handle { ray() {
    pixel-color := 0
    for i := range(0, 1000) {
      pixel-color += resume(random)
    }
    pixel-color /= 1000
    yield(pixel-color)
  } }
```

Fig. 2.10: Ray tracing with multi-shot resumption

Non-tail-resumptive handlers

Being the most generic and powerful case, non-tail-resumptive handlers does not cast assumptions over the position and count of **`resume`** and **`yield`**. There could be remaining code after **`resume`** and **`resume`** could be used multiple times. If there is only a single **`resume`**, it is called *single-shot* handler while the others are called *multi-short* handlers. Single-shot handlers are already demonstrated in Figure 2.4.

Multi-short handlers are much more complex and could create new programming paradigms. For example, in Figure 2.10, the code uses effect handling to provide a random ray generator to perform ray casting rendering, but unlike tail-resumptive random generator before, it does not immediately `yield(resume(...))` with a random ray and resumes for a multiple of times to sample the pixel based on 1000 random rays. This implementation provides the random generator while collecting the result and performing sampling, which could provide new point of view of writing probabilistic programs.

2.1.2 Formal definition of effect handling

To give a formal definition of effect handling, we give an introduction following the formal definition of the language L_S in [7] which is based on $\lambda^{\rho}_{\text{eff}}$ in [10]. There are a number of various formal definitions [10, 5, 8, 11] and the definition of the first proposing paper [1, 2], but we arbitrarily selected one out of them which is believed to cover the most important commonalities of these definitions and easy to be understood.

Syntax

The syntax of the language is given in Figure 2.11. There are syntax items representing types, effects and terms. The types and terms are spitted into 3 kinds, value, computation and handlers.

Value types include the *Unit* type and function types $A \to C$. It does not include integers, strings nor others since these types are irrelevant to effect handling. Computation types are composed from a value type A and a effect set E, which stands for the result of the computation and the set of effects possibly to be raised during the computation.

$$
\begin{array}{rrl}
\text{Value types} & A, B ::= & \mathit{Unit} \mid A \to C \\
\text{Computation types} & C, D ::= & A \,!\, E \\
\text{Handler types} & F ::= & C \Rightarrow D \\
\text{Effect} & S ::= & l : A \to B \\
\text{Effect sets} & E ::= & \phi \mid \{S\} \uplus E \\
\text{Values} & V, W ::= & \mathit{unit} \mid x \mid \lambda x.\ M \\
 & M, N ::= & V\ W \mid \mathit{return}\, V \mid \\
\text{Computations} & & \mathit{raise}\, l\ V \mid \\
 & & \mathit{let}\, x = M\ \mathit{in}\, N \mid \\
 & & \mathit{do}\, M\ \mathit{handle}\, H \\
\text{Handlers} & H ::= & \{l\ p\ k \to M\}
\end{array}
$$

Fig. 2.11: Syntax of L_S

Handler types are composed from pairs of two computation types C and D, which stands for the computation before and after installing the handlers as the result type and the set of effects could be changed by effect handlers.

Effects are defined with the label l, the effect parameter type A and the effect result type B. An effect set is composed from a set of effects S.

The values consist of the unit value, variables x and function abstractions (definitions) $\lambda x.\ M$. The computations consist of function applications (invocations) $V\ W$, return expressions $\mathit{return}\, V$, raising expressions $\mathit{raise}\, l\ V$ which raises effect l with argument V, let bindings $\mathit{let}\, x = M\ \mathit{in}\, N$ which binds x to the evaluated result of M and evaluates N with x in scope, and do-handle expressions $\mathit{do}\, M\ \mathit{handle}\, H$ which evaluates M with the handler H. Note that the language does not distinguish **`break`**, **`yield`** and **`break`**, but it uses a unified *return* expression.

A handler is defined as $\{l\ p\ k \to M\}$, where l stands for the label (name) of the effect to be handled, p stands for the effect argument and r stands for **`resume`** which is represented as a function of delimited continuation.

The definition here only permits a single do-handle expression to handle one effect for simplicity, and the keywords for the expressions are aligned with the previous introduction, which differs from the original definition in [7].

Typing rules

The typing rules of the language is given in Figure 2.12.

For the values, the typing rules are trivial. The *unit* value is of type *Unit*, variables in the typing context Γ have the defined types, and the function abstractions have resulting computation inferred by their body (M) when assigning a valid type to the parameter ($x : A$) in the typing context.

For the computations, function applications ($V\ W$) have resulting type C inferred by the type of the function ($V : A \to C$) and the argument ($W : A$). Raise expressions ($\mathit{raise}\, l\ V$) have type $B \,!\, E$ inferred by the effect ($(l : A \to B) \in E$) and the raise argument ($V : A$), where the effect must be present in the set of effects of the type of computation of the raise expression. Return expressions ($\mathit{return}\, V$) have type $A \,!\, E$ inferred by the returning value ($V : A$), where the set of effects are arbitrary. Let bindings ($\mathit{let}\, x = M\ \mathit{in}\, N$) have type inferred from the body ($N : B \,!\, E$) with the type of variable ($M : A \,!\, E$) provided, which deducts from replacing the variable name x with the value of the variable in the body N. Do-handle expressions have computation types D inferred from the *do* part ($M : C$) and the handler part $H : C \Rightarrow D$, which changes the original computation of type C into a

$\boxed{\Gamma \vdash V : A}$

$$\Gamma \vdash \mathit{unit} : \mathit{Unit} \quad (\textsc{T-Unit}) \qquad \frac{x : A \in \Gamma}{\Gamma \vdash x : A} \quad (\textsc{T-Var}) \qquad \frac{\Gamma, x : A \vdash M : C}{\Gamma \vdash \lambda x.\ M : A \to C} \quad (\textsc{T-Abs})$$

$\boxed{\Gamma \vdash M : C}$

$$\frac{\Gamma \vdash V : A \to C \quad \Gamma \vdash W : A}{\Gamma \vdash V\ W : C} \quad (\textsc{T-App}) \qquad \frac{(l : A \to B) \in E \quad \Gamma \vdash V : A}{\Gamma \vdash \mathit{raise}\, l\ V : B\,!\,E} \quad (\textsc{T-Raise})$$

$$\frac{\Gamma \vdash V : A}{\Gamma \vdash \mathit{return}\, V : A\,!\,E} \quad (\textsc{T-Return}) \qquad \frac{\Gamma \vdash M : A\,!\,E \quad \Gamma, x : A \vdash N : B\,!\,E}{\Gamma \vdash \mathit{let}\, x = M\ \mathit{in}\, N : B\,!\,E} \quad (\textsc{T-Let})$$

$$\frac{\Gamma \vdash M : C \quad \Gamma \vdash H : C \Rightarrow D}{\Gamma \vdash \mathit{do}\, M\ \mathit{handle}\, H : D} \quad (\textsc{T-Handle})$$

$\boxed{\Gamma \vdash H : F}$

$$\frac{C = A\,!\{l : A_l \to B_l; E'\} \quad \Gamma, (p : A_l), (k : B_l \to A\,!\,E') \vdash N_l : D}{\Gamma \vdash \{l\ p\ k \to N_l\} : C \Rightarrow D} \quad (\textsc{T-Handler})$$

Fig. 2.12: Typing rules of L_S

$$\begin{array}{rrl}
\textsc{E-App} & (\lambda x.\ M)\ V & \rightsquigarrow M[V/x] \\
\textsc{E-Let} & \mathit{let}\, x = \mathit{return}\, V\ \mathit{in}\, N & \rightsquigarrow N[V/x] \\
\textsc{E-Handle} & \mathit{do}\, E[\mathit{raise}\, l\ V]\ \mathit{handle}\, H & \rightsquigarrow M[V/p, (\lambda y.\ \mathit{do}\, E[\mathit{return}\, y]\ \mathit{handle}\, H)/k] \\
 & & \quad \text{where } H^l = \{l\ p\ k \to M\} \\
\textsc{E-Lift} & E[M] & \rightsquigarrow E[N] \text{ if } M \rightsquigarrow N \\
\text{Evaluation Contexts} & E & ::= [\,] \mid \mathit{let}\, x = E\ \mathit{in}\, N
\end{array}$$

Fig. 2.13: Operational semantics of L_S

computation of type D.

For the handlers, given a computation C which includes the effect $\{l : A_l \to B_l\}$, the handler have the raise argument p bound to type A_l according to the effect and the resumption r bound to type $B_l \to A\,!\,E'$ based on the effect resulting type B_l and the type of the body of the *do* part $A\,!\,E'$ which removes the handled effect $\{l : A_l \to B_l\}$ from computation C. The type of the handler is inferred from the *do* part and the handler body $(N_l : D)$ which forms the handler type transforming a computation of type C to a computation of type D.

Operational semantics

The operational semantics of L_S is given in Figure 2.13. The rules E-App and E-Let are replacing the corresponding terms x with their values M or N. The rule E-Handle invokes the corresponding handler for a *raise* $l\ V$ expression, where the control flow is handed to the handler with the continuation k representing the remaining code in the do-handle expression.

```
1 effect ask<a>
2   ctl ask() : a
3
4 fun add-twice() : ask<int> int
5   ask() + ask()
6
7 fun ask-random() : random int
8   with ctl ask() resume(random-int())
9   add-twice()
```

Fig. 2.14: Effect handling in Koka

2.2 Related works

For the related works of effect handling, there are explorations over the programming interface and the implementation techniques.

2.2.1 Programming interfaces

There are a number of both industrial and research programming languages and libraries which provides support of effect handling, including OCaml [12], Koka [13], cpp-effects [4], Flix [14] and many more. Additionally, there are low-level programming interfaces related to effect handling such as WasmFx [15] and libhandler [16].

For the programming languages and the libraries, most of them adopts a functional interface. For example, in Koka [13], the effect handling are written like in Figure 2.14. The effect is defined with the keyword `effect` continued by its name on line 1. It is raised with a syntax similar to function invocation with `ask()` on line 5. The handler is registered with a `with` expression on line 8 - 9 where the `ctl` defines the handler on line 8, and the computation is defined with the expression on line 9. The resumption is represented as a intrinsic function like `resume(random-int())` on line 8.

Most languages also adopts explicit type annotation of the possible effects. For example, in the Koka code in Figure 2.14, the function signature on line 4 includes the possible effect `raise`. On line 7, since the effect `ask` is handled while the handler raises new effect `random`, the type signature of `ask-random` only includes the effect `random`.

By contrast, cpp-effects [4], being the only existing work in C++, exhibits an Object-Orientated Programming (OOP) style interface. The effects are defined as subclass of `eff::command` and raised by `eff::invoke_command`. The handlers are defined as subclass of `eff::handler` and are registered with `eff::handle` which is equivalent to a do-handle block.

For example, in Figure 2.15, it resembles the previous example of shared state in Figure 2.8. There are two effects `Put` and `Set` for the getter and setter respectively on line 2 and 6. The raise parameter are defined as member fields of the class on line 3. The result is defined as the template parameter of `eff::command` on line 7. The effects are raised by `eff::invoke_command` with the effect object constructed with the raise argument.

The handler is defined on line 19 - 34, where `eff::flat_handler` is a sugar for tail-resumptive handler, and the template arguments are telling the list of the effects handled. It keeps an internal state with the field `state` on line 25. It overrides `handle_command` for the commands (effects) `Put` and `Get` to define the code to be executed when the

```cpp
template <typename S>
struct Put : eff::command<> {
  S newState;
};

template <typename S>
struct Get : eff::command<S> { };

template <typename S>
void put(S s) {
  eff::invoke_command(Put<S>{{}, s});
}

template <typename S>
S get() {
  return eff::invoke_command(Get<S>{});
}

template <typename Answer, typename S>
class HStateful : public eff::flat_handler<
  Answer, eff::plain<Put<S>>, eff::plain<Get<S>>> {
public:
  HStateful(S initialState) : state(initialState) { }
private:
  S state;
  void handle_command(Put<S> p) final override
  {
    state = p.newState;
  }
  S handle_command(Get<S>) final override
  {
    return state;
  }
};

void test()
{
    std::cout << get<int>() << "␣";
    put(get<int>() + 1);
    std::cout << get<int>() << "␣";
    put(get<int>() * get<int>());
    std::cout << get<int>() << std::endl;
}

void testStateful()
{
  eff::handle<HStateful<void, int>>(test, 100);
}
```

Fig. 2.15: Effect handling in cpp-effects

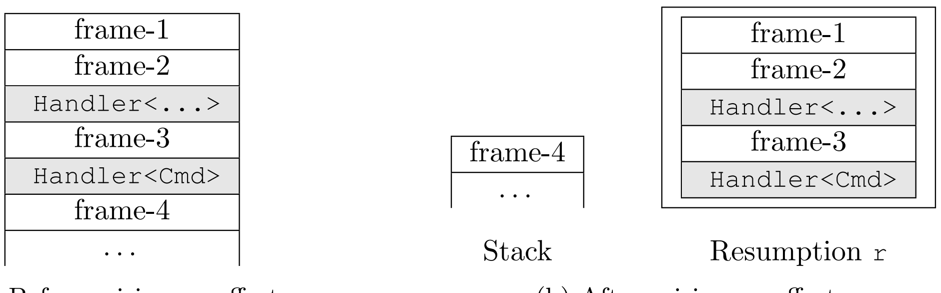


Fig. 2.16: Stack layout of cpp-effects

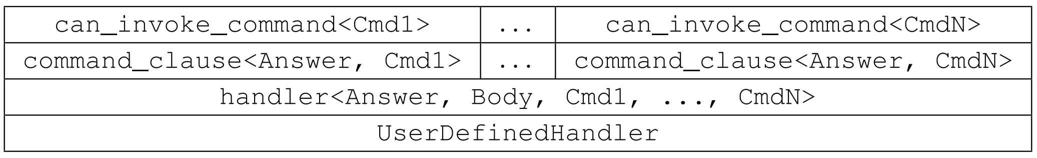

can_invoke_command<Cmd1>
...
can_invoke_command<CmdN>
command_clause<Answer, Cmd1>
...
command_clause<Answer, CmdN>
handler<Answer, Body, Cmd1, ..., CmdN>
UserDefinedHandler

Fig. 2.17: Class hierarchy of cpp-effects

corresponding effect is raised.

The function `testStateful` executes the function `test` with the handler `HStateful` in scope by calling `eff::handle` on line 47, and the constructor arguments to `HStateful` are passed as the arguments of `eff::handle`.

2.2.2 Implementation techniques

Implementation of effect handling could be sorted into stack-manipulation-based and code-transformation-based.

Implementation based on stack manipulation

MultiCore OCaml [6], Eff [17], cpp-effects [4], Helium [18], Frank [19], Links server backend [20], WasmFx [15] and libseff [21] uses stack manipulation approach to implement effect handlers. This kind of implementation usually splits the stack into smaller chunks (named *segmented stack*) which contains several frames delimited by effect raising or handlers, and the stack segments are recomposed to resemble the control-flow operations, but there are also implementations based on copying the stack [22]. The stack splitting is usually based on an existing coroutine (fiber) library, but there are also implementations with their own implementation of segmenting [6].

Taking cpp-effects [4] for an example, the function `eff::`**`handle`**`<H>(f)` registers the handler `H` by pushing a frame of object of type `H` (a *handler frame*) and executes `f()`, like in Figure 2.16a. When an effect is raised with `eff::invoke_command<Cmd>(c)`, the stack is recomposed by moving the frames of and above `Handle<Cmd>` to a resumption object `r`, and preserves all frames below frame-4, like in Figure 2.16b. This implementation technique is sometimes called *segmented stack*-based because it splits the stacks into several continous chunks of frames delimited by handler frames.

To search for a handler responding to an effect, cpp-effects maintains a class hierarchy to minimize the number of doing dynamic casting. A concrete handler is a user-defined class `UserDefinedHandler`, whose hierarchy is shown in Figure 2.17. When determining whether a frame can handle the effect `Cmd1`, it only requires casting the frame to `can_invoke_command<Cmd1>` instead of attempting all the concrete handler

```
effect flip() -> bool
effect fail()

func choice(n)
  if (n < 1) raise fail()
  else if (raise flip()) return n
  else choice(n - 1)

func main(n)
  do {
    break [choice(n)]
  } handle { flip() {
    yield join(resume(true), resume(false))
  } fail() {
    yield []
  } }
```

Fig. 2.18: Flipping a coin with effect handling

types `UserDefinedHandler`. To pass a resumption resulting in type `Answer`, it only requires to cast to `command_clause<Answer, Cmd1>`.

This kind of implementation is inefficient in most cases since the heavy overhead of switching and recomposing the stack segments. Despite there are optimizations to progressively improve the performance for some special kind of handlers, the overall overhead of manipulating the stack cannot be ignored and it has shown worse performance than using existing programming constructs to express the same computation or code-transformation-based implementations, but it has shown some advantage over code-transformation-based ones for deeply nested recursive programs [4].

Implementation based on code transformation

Koka [23, 24, 25, 16, 22, 26, 27, 5], Effekt [9], Eff [28], Links JavaScript backend [20] and an OCaml fork [29] uses some kind of code transformation to implement effect handlers.

Taking Effekt [9] for an example, let there be a program in Figure 2.18 doing random choice similarly to Figure 2.1. The code in `choice` flips the coin for at most `n` times, and returns the remaining attempts when succeeding (`flip` gives **`true`**). For each flip, the handler in `main` resumes twice with **`true`** and **`false`**, and collects the various results in an array.

The code transformed in iterated Continuation-Passing-Style (CPS) are given in Figure 2.19. Here the translated functions takes handlers as parameters since Effekt requires explicit passing of the handlers which delegates the work of finding the corresponding handler to programmers, so the work focuses on the transformation of raising an effect. To efficiently find the handler, Koka [5] gives the idea of *evidence vector*.

The code transformation creates continuation at the control-flow operations **`raise`**, **`break`**, **`resume`** and **`yield`** of effect handling. The functions are translated so that it takes additional function parameters representing the computation to be continued after some control-flow operation, and the continuation function are executed at control-flow operations, which resembles CPS transformation used in async/await or generators for effect handling.

This kind of implementation is generally more efficient than the former one in most

```
effect flip() -> bool
effect fail()

// returning a function (k1, k2) -> int
func choice(flip, fail, n)
  func loop(n, k1, k2)
    if (n < 1) return fail(void, k1, k2)
    else return flip(void, func(x, k3) {
      if (x) return k1(n, k3)
      else return loop(n - 1, k1, k3)
    }, k2)
  return loop(n)

func main(n)
  func flip(_, k) {
    return join(k(true), k(false))
  }
  func fail(_, k1, k2) {
    return k2([])
  }
  func liftedFlip(_, k1, k2) {
    flip(void, func(x) { return k1(x, k2) })
  }
  choice(liftedFlip, fail, n, func(x1, k2) {
    return k2([x1])
  }, func(x2) { return x2 })
```

Fig. 2.19: Flipping a coin in CPS

cases, but it requires global code transformation (which means library implementation is not possible for most languages) and still has space for improvement. It is straightforward to observe that the code transformation shown in Figure 2.19 introduces the additional overhead of creating a closure with invocation and return of functions possibly raising an effect. Existing code-transformation-based approaches exhibits similar property of introducing overhead to function calling or returning.

2.3 Importance of tail-resumptive handlers

To encourage the adoption of effect handlers, improving their performance is essential. In particular, there is still space of enhancement for tail resumptive handlers, an area that existing research has largely overlooked. This kind of handler has a substantial position based on its applications and special property which is explained later, whose performance contributes a larger proportion to the application compared to the other cases.

From the discovered applications of effect handling, a fact has been observed that most handlers are tail resumptive [5]. Additionally, these discovered cases of tail resumptive handlers are invoked more frequently in the programs written with consideration of effect handling compared to the other cases. As seen before, tail resumptive handlers are used for shared state and similarly it could be used for contextual access (which is a shared state without setter). For these cases, since contextual access and shared mutable access occurs frequently in the programs future written in effect handling, the handlers are invoked more frequently than the other cases such as exception handling with an yielding handler and coroutines or probabilistic programming with a non-tail-resumptive handler.

For example, it is obvious that a controller or middleware in web application accesses contextual data such as user identity, connection information, session and request parameters more frequently than it throws an exception or performs I/O operations. Almost every controller and middleware need to read the request URL, the query params, the headers and the request body, and to avoid redundant code, the user identity or related entities are often fetched from the data store once and kept in an object associated with the request while the controllers and middlewares reads the fetched entity out from the object, and it often requires obtaining database connection or configuration from the global pool prior to perform I/O operations. To keep the code clear by avoiding adding too much parameters to the controller or middleware function, these data could be accessed with effects as contextual access or shared mutable state, which usually refers to a tail-resumptive handler. It is straightforward to consider that these access are pretty frequent and there are usually multiple accesses to these data in a single controller or middleware. In contrast, exceptions which are used to handle rare conditions such as network failure or unexpected input, are rare cases taking a small proportion of all the requests the server processes, and most I/O operations may require an access to the connection pool or some global lock prior to the execution.

Another case in point is the rendering function of React or other web frontend rendering libraries. The core philosophy of React is to create *functional components* which are functions generating DOM trees, and as an encapsulation of the complex state inside the component such as user inputs and remote data fetched, React adopts effect handling to provide *React hooks* [30] like `useState` in Figure 2.20 which they believes to be better understandable and readable than monads. The invocation of these hooks are actually referring to a tail-resumptive handler since these use cases are for shared mutable state which fully covers all the usage of effect handling in a typical React application.

In the example, the function `FeedbackForm` renders a web form. The view of web form is updated with invoking `FeedbackForm` by React when the state changes. It maintains the state `text`, `isSending` and `isSent` over different invocations by using the function `useState` which returns two effects like `text` and `setText` standing for the getter and setter of the state. These effects are handled by React, and is referring to tail-resumptive handlers like shown in the previous example of shared state in Figure 2.8.

Therefore, since tail resumptive handlers are taking a majority of observed handlers usage, and the usecases of them imposes a higher rate of being invoked, we believe that tail resumptive handlers contributes a larger proportion to the performance of programs utilizing effect handling.

However, current implementations propose a unified approach focusing the support of delimited continuation which is not well optimized for tail resumptive cases. Segmented stack [6, 17, 4, 18, 19, 20, 15, 21] requires recomposing the stack where effects are invoked to enforce the scope semantics of handlers which creates relatively large overhead on every handler invocation. Code transformation [16, 27, 5, 9, 20, 29] creates additional branching code where effectful functions return and it moves part of the stack variables onto the heap which also creates heavy overhead for tail resumptive handlers which has relatively simpler semantics.

It is observed that **`yield`**`(`**`resume`**`(...))` call has a straightforward control flow which could be implemented very close to a function call thus minimizing the overhead. Tail resumption can be considered as nothing more than finding the correct handler by walking the stack and execute it on top of the current execution stack (with tweaks to preserve the handlers' scope semantics) which is extremely simple and efficient. This approach may

```
import { useState } from 'react';

export default function FeedbackForm() {
  const [text, setText] = useState('');
  const [isSending, setIsSending] = useState(false);
  const [isSent, setIsSent] = useState(false);

  async function handleSubmit(e) {
    e.preventDefault();
    setIsSending(true);
    await sendMessage(text);
    setIsSending(false);
    setIsSent(true);
  }

  if (isSent) {
    return <h1>Thanks for feedback!</h1>
  }

  return (
    <form onSubmit={handleSubmit}>
      <p>How was your stay at The Prancing Pony?</p>
      <textarea
        disabled={isSending} value={text}
        onChange={e => setText(e.target.value)}
      />
      {/* ... */}
      {isSending && <p>Sending...</p>}
    </form>
  );
}

// Pretend to send a message.
function sendMessage(text) {
  return new Promise(resolve => {
    setTimeout(resolve, 2000);
  });
}
```

Fig. 2.20: Example usage of React hook [30]

come at the expense of the performance of other types of handlers. However, given the importance of tail resumption, as outlined above, we consider this trade-off to be both reasonable and beneficial. Therefore, the performance of tail resumptive handlers could be further improved which benefits the entire application.

Additionally, for cases other than tail resumptive ones, there are mitigations to avoid the performance degrading or it sounds reasonable to programmers. First, most non-tail-resumptive cases with a single resumption could be rewritten by moving the remaining code to the end of the do-handle block as a mitigation. For example, in Figure 2.21 [3],

```
effect operate(int)

func post_compute(n: int, s: int)
  do {
    for i := range(0, n) {
      operate(i)
    }
    break s
  } handle { operate(x) {
    y := resume()
    yield(abs(x - (503 * y) + 37) % 1009)
  } }
```

Fig. 2.21: Example of non-tail-resumptive handler

```
effect operate(int)

func post-compute(n: int, s: int)
  counter := 0
  y := do {
    for i := range(0, n) {
      operate(i)
    }
    break s
  } handle { operate() {
    counter := counter + 1
    yield(resume())
  } }
  while (counter != 0) {
    counter := counter - 1
    y := abs(x - (503 * y) + 37) % 1009
  }
```

Fig. 2.22: Rewriting remaining code of handler in Figure 2.21

it registers a handler of the effect `operate` which alternates the result value y of the do block by the formula $|(x - 503 \times y + 37) \bmod 1009|$. The do block invokes the handler by n times and evaluates to the value s, so the formula is applied n times on the value s. For example, if $n = 2$, the result value of do-handle is $|(x - 503 \times |(x - 503 \times s + 37) \bmod 1009| + 37) \bmod 1009|$.

The code can be rewritten to eliminate the non-tail-resumptive handler by manually maintaining a counter and apply the formula by the counter like in Figure 2.22. Despite that a formal method of eliminating remaining code in handlers is yet to be discovered, the basic idea behind this rewriting is to push remaining code to the end of do-handle instead of writing it in the handler which is mostly possible. Given this rewriting as a mitigation, the performance is no longer degraded at the cost of writing code not styled in effect handling.

In addition, it is acceptable that the others not being able to be rewritten has worse performance since these cases are rare. These cases together (tail-resumptive, rewritable non-tail-resumptive and non-rewritable cases) fully covers the explored use cases of effect handlers. A first non-rewritable case is yielding whose explored usage only contains the case *exception handling*, and exceptions are expected to be rare in applications.

For example, in Figure 2.9, there is an effect `divZero` which represents for "division by zero" exception. The `div` function detects whether the divisor q is 0, and raises the effect to indicate the exception has occurred. A good application developer should validate user inputs carefully to detect invalid values (such as 0 for the divisor) instead of passing the value directly to the function and let it trigger an exception. Therefore, the performance degrading of yielding handlers are acceptable since the invocations should be considered as internal fault of programs instead of normal code paths.

The other case is multi-shot resumption which is a new complex pattern of programming introduced by effect handling, which is used for Monte Carlo sampling or decision process, whose performance degrading is acceptable since this pattern is new to programmers and the space for improvement could be excepted.

For example, in Figure 2.10, there is an effect `ray` which generates a random ray at each invocation. In addition it also collects the computed pixel by accumulating the result of **`resume`** which is the result of rendering the pixel with the given random ray. By utilizing the power of effect handling, the programmers could write the code of ray-tracing programs more briefly. Since these use cases are relatively new programming paradigms emerging after the introduction of effect handling, it is reasonable for the programmers to expect a temporarily degraded performance.

Chapter 3

eff-unwind: Optimizing for tail-resumptive handlers

We present an implementation for effect handling as a library *eff-unwind* in C++, which prioritizes the performance of **`raise`** and **`resume`** at the cost of the performance of **`yield`**, thereby improving the overall performance of effect handling when most invocations refer to tail-resumptive handlers. Our proposal is based on the observation that exception handling is optimized for the normal path at the expense of the exceptional path. Following a similar structure, we implement raising and tail resuming close to function invocation and return, without recomposing the stack or creating a closure to minimize overhead. This approach incurs the cost of using stack unwinding for yielding, similar to exception handling. Another cost is the support for non-tail resumption required to handle the additional control flow operations of effect handling, achieved through stack preservation and setjmp/longjmp.

Following the previous introduction of effect handling in section 2.1, we consider it as a set of control-flow operators. The explanation to the implementation is given in a typical order of effect handling flow, as shown in Figure 2.5. First, we introduce **`do-handle`** block, which declares an effect handler and a computation. Next, we describe **`raise`**, which invokes an effect handler. We then explain **`resume`**, which passes control from handler to the computation. Following this, we show how a computation ends with **`break`** and continues with the remaining code in the handler. Finally, we illustrate the end of do-handle from a **`yield`** from the handler.

3.1 Implementation of do-handle

3.1.1 Interface of do-handle

The effects are declared as class inheriting from **`effect`**, where there are two template

```
effect Get() -> int
effect Set(int in) -> int
```

(a) Pesudocode

```
struct Get :
  public effect<unit_t, int> {};
struct Set :
  public effect<int, int> {};
```

(b) C++

Fig. 3.1: Code of effect in countdown

parameters to the base class standing for the effect parameter and the result type. For example, for the effects shown in Figure 3.1a, effect handling is used for accessing shared value, and the effect `Get` is defined with no effect parameter but resulting type **`int`** which reads out the value, while the effect `Set` is defined with the effect parameter `in` of type **`int`** and the resulting type **`int`** which sets the value and returns the updated value. It is defined as two classes in eff-unwind as in Figure 3.1b. Effect `Get` declares the effect parameter as `unit_t` to avoid having separate template definition for **`void`** of the base class `effect`, where `unit_t` is a zero-sized struct as a placeholder.

The class `effect` creates two member typedefs for the type parameters, which are `raise_t` for the effect parameter and `resume_t` for the result type. Therefore, the user could use `Effect::raise_t` or `Effect::resume_t` for any effect declarations inheriting from `effect`.

do-handle is implemented as a function call to `do_handle` taking two lambda functions standing for the do block and the handle block respectively, and two template parameters for the result type of do-handle block and the effect. The do lambda does not take any parameters, and the return type of it becomes the return type of the invocation to `do_handle`. The handle lambda takes 3 parameters, the effect parameter, the resume operator and the yield operator.

For example, for the code in Figure 3.2a, it is written as in Figure 3.2b in eff-unwind. The do-handle block on the pseudocode line 3 - 11 are transformed into a function call on C++ code line 4 - 15. Our library does not support handling multiple effects in a single do-handle block for simplicity, so the user has to nest do-handle blocks to handle both `Get` and `Set`. Since the do block becomes a lambda, it uses **`return`** instead of **`break`** for giving its resulting value. The **`resume`** and **`yield`** are given as parameters to the handler, and can be called like a function to invoke the control-flow operations.

In order to distinguish tail resumptive handlers and others, we introduces a special way of writing the tail resumptive handlers' lambda function as in Figure 3.3. The first version in Figure 3.3a, being generic, does not tell the library about the special property of the handler being tail resumptive, while the second one in figure Figure 3.3b, omits the parameter for operators and directly uses **`return`** to represent the same control-flow operation as `yield(resume(s))`. The library could detect that the handler is tail resumptive by type-checking that the handler function only accepts one parameter instead of three.

3.1.2 Tracking the active handlers

We adopt dynamically bound [27, 31, 32] effect handling, which means runtime data structures has to be maintained for keeping which handler is active and thus invoking. Therefore in programs with eff-unwind, there is a global stack containing all the *handler frames* named *handler vector*, each representing an active handler, which effectively tracks all the handlers in scope. The handler frame class is hierarchical, like in cpp-effects [4] for similar reason. The base class `handler_frame_base` is the most generic one for efficient searching corresponding handler, which contains the following fields.

- *id* `id`: A sequential identifier for the frame to assert for bugs on deregistration of the frame.
- *effect typeid* `effect_type`: The typeid of the effect to be handled by the handler of this frame.
- *resumption frame pointer* `resume_fp`: The frame pointer of the caller of current frame, which is used as a boundary of stack unwinding later in section 3.5.
- *handler stack pointer* `handler_sp`: The stack pointer of the current frame pointing

```
func run(n)
  s := n
  return do {
    do {
      break countdown()
    } handle { Set(in) {
      s = in; yield(resume(s))
    } }
  } handle { Get() {
    resume(s)
  } }
```

(a) Pesudocode

```
int run(int n) {
  auto s = n;

  return do_handle<int, Get>([&]() -> int {
    return do_handle<int, Set>([]() -> int {
      return countdown();
    },
    [&](int in, auto resume, auto yield) -> int {
      s = in;
      yield(resume(s));
    });
  },
  [&](unit_t, auto resume, auto yield) -> int {
    yield(resume(s));
  });
}
```

(b) C++

Fig. 3.2: Code of do-handle in countdown

```
[&](unit_t, auto resume, auto yield) -> int {
  yield(resume(s));
}
```

(a) Without optimization

```
[&](unit_t) -> int {
  return s;
}
```

(b) With optimization

Fig. 3.3: Two ways of writing tail resuming in countdown

to the start memory address of the current stack frame, which is used as a boundary for stack copying later in section 3.3.

- *stack of resumption frames* `resumption_frames`: A empty-initialized stack for storing the resumption frames used in break and yield as described later in section 3.3.
- *parent frame offset* `parent_delta`: An offset indicator to skip some of the handler frames to ensure correct stack as described in section 3.2.
- *tail resumptive indicator* `is_tail_resumptive`: An indicator for being tail resumptive or not to enable the optimization for tail resumptive handlers.

Then, a little specialized one, `handler_frame_invoke`, contains an extra template parameter `Effect` standing for the effect to be handled. Thanks to this parameter, the type signature of invoking the effect could be deduced to a function taking a parameter of the effect parameter type `Effect::raise_t` returning a value of the effect result type `Effect::resume_t`. This function signature is defined as a pure virtual member function, so the library could simply cast the handler frame to `handler_frame_invoke<Effect>` without knowing the type of handler function nor the result type of do-handle block at **`raise`**, since the result type of do-handle block is not tied to effect declaration but to a specific do-handle block (or to say, a specific handler definition).

The next level of subclass is `handler_frame_yield` additionally adding the type of the value to **`break`** in do block as a template parameter *yield type* `yield_t`, which is also the type for **`yield`** in the handler since we requires the result type of do block and handler must be the same for simplicity without losing generality. It also contains a field of the yield type named *yield value* `yield_value` holding the evaluation result of the do-handle block after **`break`**. This level is introduced for creating the **`resume`** and **`yield`** operators provided to handlers when invoking, but still do not provide the implementation for invoking an effect handler.

The final level in the hierarchy is `handler_frame` which contains a template parameter of the handler type (since each C++ closure has a different anonymous type), an object of the handler type (which represents for an instance of the closure) and an implementation for invoking the handler.

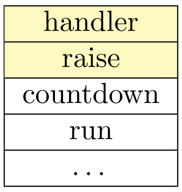
Since `handler_frame` is monomorphized for the template parameters, the handler vector is a C++ standard template library (STL) vector of `std::shared_ptr` holding objects of common abstract type `handler_frame_base` since the base class `handler_frame_base` is not monomorphized for not having any template parameter. The concrete type of the objects in the vector varies basing on the effect and the handler, and it could dynamically cast the references to the value inside smart pointers to get the concrete typed object.

Upon invocation of `do_handle`, it constructs *handler frame* and pushes to the global vector. The construction is mostly trivial as filling the fields as described above, while there are specializations of the function taking different kind of template parameters to distinguish between tail resumptive handlers and others while setting the indicator. The specialization is based on C++ concepts which asserts the number of parameters possibly passed to the handler function to enable or disable the function template, which means the template for tail resumptive handlers are used for the corresponding cases while the template for general handlers are used for others.

It also constructs a `scope_guard` [33] which adds the code related to deregistering the handler to be executed automatically when the frame of `do_handle` goes out of scope. Additionally, it also executes remaining code in the handler as described later in section 3.4.

```
 1 int countdown() {
 2   auto i = raise<Get>({});
 3   while (true) {
 4     i = raise<Get>({});
 5     if (i == 0) {
 6       return i;
 7     } else {
 8       raise<Set>(i - 1);
 9     }
10   }
11   assert(false);
12   return 0;
13 }
```

(a) Code of raise in countdown



(b) Call stack for countdown at raise

Fig. 3.4: raise in countdown

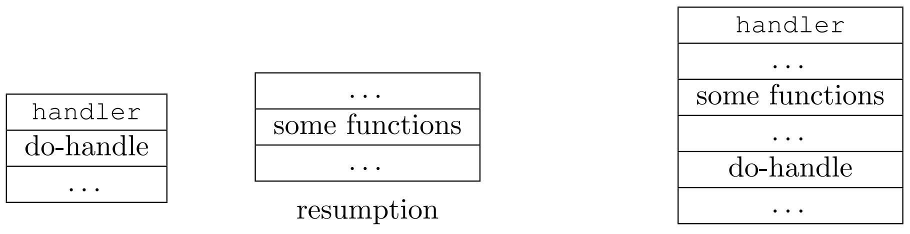
3.2 Implementation of raise

The first step in effect handling is raising an effect, which is achieved using the **raise** operator. Recalling the introduction of effect handling before, **raise** finds the nearest handler of the designated effect in the stack and executes it. In our library, **raise** is a function taking a template parameter of the effect type and a parameter for the effect parameter, which is implemented as finding the handler and executes it as a function call on top of the current calling frame.

Continuing with our previous example in Figure 3.1 and Figure 3.2, we are still missing the `countdown` function which accesses the shared state with the effects defined, so now the code in Figure 3.4a is introduced. On line 2, 4 and 8, it raises the effects by calling **raise**, where it uses the effect `Get` to read the shared value, and updates the value with `Set`. It counts from the initial value down to 0 stepping by 1, and returns the final value (which should be 0).

raise searches through the handler vector from the top (last registered) to the bottom (first registered). If it encounters a frame with parent frame offset $n > 0$, it skips the current frame and the following subsequent $n - 1$ frames, whose reason is explained later in subsection 3.2.1. On each frame, it checks whether the frame could handle the effect by comparing the typeid of the effect raised with the effect typeid field in the frame. If no handler frames are found, the current process is aborted, which is similar to the behavior in exception handling. If found, it goes through the process of invoking the handler, which basically contains the following steps.

1. Cast the frame to a more concrete type `handler_frame_invoke<Effect>` to prepare for the invocation of the handler.
2. Mask part of the handler vector by manipulating the parent frame offset of the last frame in the vector.
3. Construct a *raise context* for the effect and the raise invocation.
4. If the handler is tail resumptive, (by examining the tail resumptive indicator in the handler frame)
   (a) Invokes the handler with the effect argument.



(a) Most other implementations (b) Our implementation

Fig. 3.5: Call stack for executing the handler in countdown

   (b) After the handler returns, return the effect result value from the return value of the handler.
5. Otherwise,
   (a) Save the current execution status by `setjmp` to the raise context.
   (b) Construct a *resume context* and *yield context*.
   (c) Invokes the handler with the effect argument and the two contexts.
   (d) After the handler returns by `longjmp` to the previous saved execution status in step 5a, return the effect result value from the raise context.

The raise context contains the following fields.

- Template parameter `Effect`: The effect type. For accessing the type of effect parameter and effect result.
- jump buffer: A field for storing the execution status in step 5a.
- Effect result value: A field for storing the effect result value for non-tail-resumptive handlers.

The resume context and yield context contain references to the raise context and other things related to **`resume`** and **`yield`**, which is explained in section 3.3 and section 3.5.

Most of the steps are trivial, but skipping frames in searching and the step 2 seem to be inordinary, which is explained as follows in subsection 3.2.1. In addition, the step 5a and the resuming by `longjmp` from the handler seems to also be not straightforward, which is explained later in subsection 3.2.2 and section 3.3.

3.2.1 Masking the handler vector

Differently from the other implementations, our approach are invoking the handler code as a frame on top of the current execution stack (Figure 3.5b) instead of recomposing the stack and put the frame on top of the registration frame (Figure 3.5a). This choice has improved the performance by eliminating the overhead of recomposing the stack [5] while introducing an additional problem of having to mask part of the handler vector in order to keep the scope of handlers correct, since the handlers registered in `foobar` should not be effective for the code in `handler`.

For example, in code in *handler sieve* (Figure 3.6) [3], the function `run` computes the sum of all the primes less than `n` by using trial division. However, the list of all the primes found are not kept in a data structure but as registered effect handlers of `Prime`. When the effect `Prime` is raised with argument `e`, it finds some handler doing trial division over `i`. As on line 11 - 18, the handler behaves as if the current trial division fails (which means $e \mod i = 0$), it yields **`false`** since `e` cannot be a prime; while if it succeeds, it delegates the trial division to the next handler in the stack for some prime smaller than `i`. If the code on line 7 finds some new prime, it adds the prime to the result, and pushes a new

```
 1 struct Prime : public effect<uint64_t, bool> {};
 2
 3 int primes(int i, int n, int a) {
 4   if (i >= n) {
 5     return a;
 6   }
 7   if (raise<Prime>(i)) {
 8     return do_handle<int, Prime>(
 9       [&]() {
10         return primes(i + 1, n, a + i);
11       }, [i](int e) -> bool {
12         if (e % i == 0) {
13           return false;
14         } else {
15           auto val = raise<Prime>(e);
16           return val;
17         }
18       });
19   } else {
20     return primes(i + 1, n, a);
21   }
22 }
23
24 int run(int n) {
25   return do_handle<int, Prime>(
26       [&]() {
27         return primes(2, n, 0);
28       }, [](int) -> bool {
29         return true;
30       });
31 }
```

Fig. 3.6: Handler sieve

handler for the prime as on line 8. Therefore, the stack of handlers in scope also forms a list of all the primes found, and the code uses effect handling to traverse over this list to check if a number is a prime.

Therefore, a typical handler vector of handler sieve could be like in Figure 3.7a for eff-unwind. For the handler of trial division on n, it is written as `do_handle<Prime>[n]` in the figure. When the code raises `Prime` with the number 12, the calling stack could be as in Figure 3.7b at some point. Taking `Prime(7)` for an example, since the do-handle blocks are all in scope, the handler vector is like in Figure 3.7a. If it does not apply the masking technique, the handler `do_handle<Prime>[11]` would be found.

However, the user expects the call stack to be in the shape shown in Figure 3.8a, where the handler for `Prime(7)` is executed directly on top of `primes(7, 12, 10)` where `do_handle<Prime>[11]` and `do_handle<Prime>[7]` is not in scope, and the handlers with $i < 7$ should be found.

In order to address the problem of unwanted handlers being in scope, differently from [5], we create a mask to temporarily deactivate the handlers registered between the frame

...
`do_handle<Prime>[11]` (line 11 - 18)
`do_handle<Prime>[7]` (line 11 - 18)
`do_handle<Prime>[5]` (line 11 - 18)
`do_handle<Prime>[3]` (line 11 - 18)
`do_handle<Prime>[2]` (line 29)

(a) Handler vector

handler for `Prime(7)` (line 11 - 18)
`raise``<Prime>(7)` (line 15)
handler for `Prime(11)` (line 11 - 18)
`raise``<Prime>(12)` (line 7)
`do_handle<Prime>[11]` (line 8)
`primes(11, 12, 28)` (line 7)
...
`primes(8, 12, 17)` (line 7)
`do_handle<Prime>[7]` (line 8)
`primes(7, 12, 10)` (line 7)
...

(b) Calling stack

Fig. 3.7: Runtime data structures for handler sieve (1)

handler for `Prime(7)`
`primes(7, 12, 10)`
...
`do_handle<Prime>[5]`
...

(a) Expected call stack

...
`do_handle<Prime>[11]` (parent frame offset = 2 by **`raise`**`<Prime>(7)`)
`do_handle<Prime>[7]`
`do_handle<Prime>[5]`
`do_handle<Prime>[3]`
`do_handle<Prime>[2]`

(b) Handler vector inside handler for `Prime(7)`

Fig. 3.8: Runtime data structures for handler sieve (2)

of the registration of invoked handler and the frame of invoked handler, which imposes lighter overhead for **`raise`**. As previously introduced, the handler frames contain the parent frame offset fields. When raising an effect, before invoking the handler, it sets the parent frame offset of the last handler to be the distance between the handler found and the last handler, which effectively skips all the handlers from the handler found (including) and the last handler (including). It also sets up a scope guard which restores the parent frame offset for the last handler when the raise function goes out of scope.

Therefore, going back to our previous example in Figure 3.6, when it executes **`raise`**`<Prime>(11)`, it sets the parent frame offset of `do_handle<Prime>(11)` to 1 since there is a frame `do_handle<Prime>(11)` between the last frame and the current frame (inclusive). Therefore, the **`raise`** in handler for `Prime(11)` does not find the handler `do_handle<Prime>[11]` as the searching process of **`raise`** skips 1 frame when reaching `do_handle<Prime>[11]`. Similarly, the **`raise`** in handler for `Prime(7)` skips 2 frames, as shown in Figure 3.8b where the masked frames are in gray background, and finds `do_handle<Prime>[5]` as expected by the user.

3.2.2 setjmp for resuming from handler

To give some background, `setjmp` [34] is a function in C standard library which saves the current execution context into a *jump buffer*. The program could later restore the execution context by `longjmp`, where the execution continues at the construction of the jump buffer variable passed to `longjmp`. As `setjmp` returns twice, for saving the jumping

```
 1 std::jmp_buf solver_error_handler;
 2 
 3 std::array<double, 2> solve_quadratic_equation(
 4   double a, double b, double c)
 5 {
 6     const double discriminant = b * b - 4.0 * a * c;
 7     if (discriminant < 0)
 8         std::longjmp(solver_error_handler, true); // Go to error handler
 9 
10     const double delta = std::sqrt(discriminant) / (2.0 * a);
11     const double argmin = -b / (2.0 * a);
12     return {argmin - delta, argmin + delta};
13 }
14 
15 void show_quadratic_equation_solution(double a, double b, double c)
16 {
17     std::cout << std::format(
18       "Solving␣{}x^^c2^^b2␣+␣{}x␣+␣{}␣=␣0...\n", a, b, c);
19     auto [x_0, x_1] = solve_quadratic_equation(a, b, c);
20     std::cout << std::format(
21       "x^^e2^^82^^81␣=␣{},␣x^^e2^^82^^82␣=␣{}\n\n", x_0, x_1);
22 }
23 
24 int main()
25 {
26     if (setjmp(solver_error_handler))
27     {
28         // Error handler for solver
29         std::cout << "No␣real␣solution\n";
30         return EXIT_FAILURE;
31     }
32 
33     for (auto [a, b, c] : {
34       std::array{1, -3, 2}, {2, -3, -2}, {1, 2, 3}
35     })
36         show_quadratic_equation_solution(a, b, c);
37 
38     return EXIT_SUCCESS;
39 }
```

Fig. 3.9: Example of setjmp and longjmp

buffer and for recovering from `longjmp`, the program distinguishes that by examining the return value of `setjmp`.

For example, in the code in Figure 3.9 borrowed from [34], it solves quadratic equations in `solve_quadratic_equation`, and it uses setjmp/longjmp to handle exceptions instead of using try-catch. On line 26, it sets the jump buffer `solver_error_handler` declared on line 1, which creates a execution context to be restored by `longjmp` later. The return value of `setjmp` is falsy, so it knows that it is saving an execution context on line 26 and proceeds with the logic on line 33. When solving an equation in `solve_quadratic_equation`, if the discriminant is less than 0, the equation does not have an solution which is an exceptional case, and it uses longjmp to restore the execution context in the jump buffer `solver_error_handler`. The code on line 26 gets the control flow and knows it is a resuming from `longjmp` as the return value of `setjmp` becomes truthy, so it proceeds with the error handler defined on line 28.

setjmp is commonly used as a lightweight way of switching execution context between non-local code. However, it has many limitations.[35] If replacing the jumping of longjmp by try-catch would "invoke a non-trivial destructor for any automatic object", the behavior is undefined. An observation for the current implementation of the C standard library shows that longjmp do not execute the non-trivial destructors of the objects for the frames traversed, as it is not aware of non-trivial destructors for being a C construct. If "the function that called setjmp has exited", the behavior is also undefined, which means it only permits jumping downwards the call stack. An observation shows that if the stack frame containing call to `setjmp` is destroyed, the program encounters invalid state.

In order to utilize the properties of setjmp/longjmp, we sets up a jump buffer in step 5a for **`raise`**, which is resumed later by **`resume`** with `longjmp`. The reason behind this is illustrated in section 3.3.

3.3 Implementation of resume

The second step in effect handling is **`resume`** from a handler. Recalling the introduction of effect handling before, **`resume`** passes the control flow back to the do block as if **`raise`** is a function and has just returned. In our implementation, **`resume`** has two different implementations, one for normal **`resume`** and one of the special tail resuming (**`yield`**`(`**`resume`**`(...))`), where the specialization provides performance improvement.

3.3.1 Implementation of normal resumption

Normal **`resume`** is implemented as saving part of the stack frames and the registers and then returning to the callsite of raise. The first part is conducted in order to execute the remaining code in handler after **`break`** (or **`yield`** sometimes), and the second part is about the control-flow semantics of **`resume`**.

As briefly introduced before, in eff-unwind, there is a resume context passed to the effect handler as an argument. The resume context is indeed a class with template parameters and fields to track the components required by performing resumption, which are as follows.

- Template parameter *effect type* `Effect`: The type of the effect.
- Template parameter *yield type* `yield_t`: The type of the result of do-handle block.
- Field *handler frame reference* `handler_frame`: A reference to the handler frame of the invoked handler.
- Field *raise context reference* `ctx`: A reference to the raising context.

The resume context overloads the function call operator (**`operator`**`()`), so in order to perform a resumption, the handler code invokes the resume context similarly to a function, passing a parameter of effect result type, and receiving a value of do-handle result type `yield_t`, like on line 9 in Figure 3.10. Since the resume context is created after a concrete handler is found, it knows the do-handle result type from the specialized `handler_frame_yield`.

First, it sets the resumption value in the raise context. Since the resume context holds

```
struct count : public effect<unit_t, int> {};

int runCounter() {
  auto [x, counter] = do_handle<std::pair<int, int>, count>(
    []() -> int {
      return std::make_pair(counter(3), 0);
    },
    [](unit_t, auto resume, auto yield) {
      auto [x, counter] = resume({});
      yield(std::make_pair(x, counter + 1));
    });
  return x;
}

int counter(int x) {
  raise<count>({});
  if (x == 0) return x;
  else return counter(x - 1);
}
```

Fig. 3.10: Simplified counter in C++

raise context reference, it stores the value in the effect result value field by the reference. The value is passed in this way as **resume** passes control flow back to **raise** by `longjmp` instead of function returning.

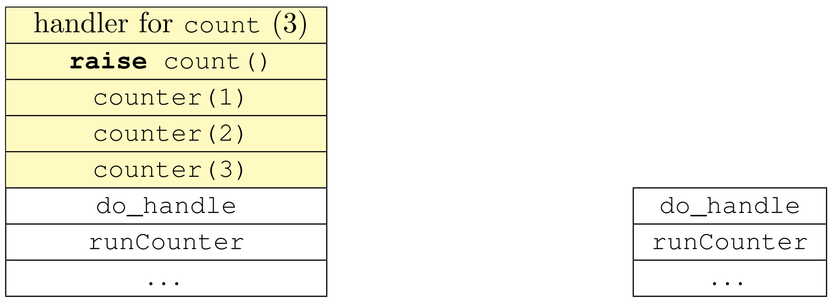
Second, the stack and registers are saved with memory copying and `setjmp`, creating a resumption frame, in order to restore execution of the remaining code in the handler in the future. A resumption frame contains the following fields.

- jump buffer `saved_jmp`.
- Part of the stack preserved `saved_stack`.
- Preserved copy of handler frames `saved_frames`.

The jump buffer is constructed with `setjmp`. The copy of handler frames are copied from the current handler vector. Since the handler vector is containing a list of shared pointers (reference counting pointers) to handler frames, it does not copy the objects but only the reference and increases the reference counter for memory management.

The part of the stack contains the stack frames from the current frame to the frame of the do-handle block (inclusive). The memory address for the boundaries of these frames are read out from the current stack pointer, and the handler stack pointer in the handler frame. Since the remaining code of the handler is expected to be executed after the do block ends, at that point, the frames in the specified range is destroyed as the function returns, which should be preserved to completely restore the execution state as `setjmp` do not preserve stack memory.

For example, considering the code in Figure 2.6, it uses effect handling to count the execution of the function `counter`, and a typical call stack would be as in Figure 3.11a. For the third invocation to the handler of `count`, it is denoted on the top of the stack. It resumes back to **raise** `count()` and then to `counter(1)`, but there are remaining code to be executed before do-handle ends where the stack looks like in Figure 3.11b. The frame "handler for `count` (3)" need to be restored in the same position, and the yielding



(a) Stack when executing handler before resume (b) Stack when executing remaining code

Fig. 3.11: Call stack for simplified counter

for the handler iterates over all the stacks below it as shown later in section 3.5, so the stack highlighted in yellow in Figure 3.11a need to be preserved.

Finally, **`resume`** uses `longjmp` to pass control back to raise, and raise returns the stored resumption value back to its callsite. This skips the non-trivial destructors in the handler frame in order to preserve for the execution of non-tail code, as the frame is restored later for remaining code and the execution of remaining code executes the non-trivial destructors.

Looking back to Figure 3.11a, the frame "handler for `count` (3)" is restored later with the technique explained later in section 3.4 in order to execute the remaining code, so the non-trivial destructors in that frame should not be executed. Therefore, it uses longjmp to resume back to **`raise`** `count()`, skipping the destructors in that frame.

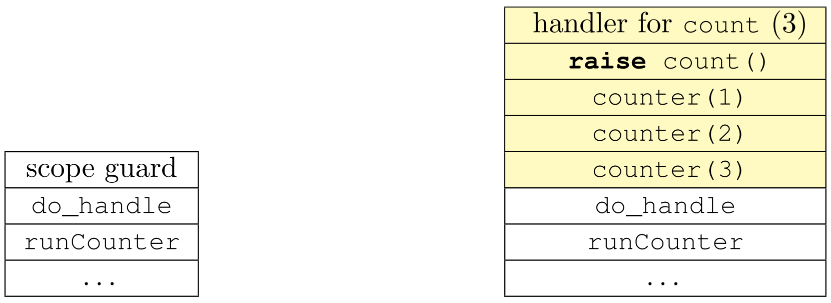
3.3.2 Implementation of tail resumption

Tail resumption (**`yield`**`(`**`resume`**`(_))`) is specialized to be implemented as function returning without any additional process. Recalling the introduction over the tail resumptive handlers, the handlers are special in terms that they do not contain any remaining code after **`resume`** changing the resulting value of do block, since **`yield`** marks the end of the handler and yielding with the value from resumption does not impose any computations between **`break`** and **`yield`**. Therefore, the stack frame for tail resumptive handlers does not need to be preserved, which eliminates the need for using setjmp/longjmp in raising and resuming, and the need for preserving the execution state in resuming.

Therefore, following the style of writing tail resumption as function returning like in Figure 3.3b, the tail resumption is expresses as a simple **`return`** statement. The control flow goes back to **`raise`**, and it directly returns the effect result value to the callsite. Since this implementation skips unnecessarily saving the execution state, we suppose that this technique minimizes the overhead for invoking a tail resumptive handler and imposes performance improvement.

3.4 Implementation of break

Moving on to the third point, the do block concludes with a **`break`**. Recalling the introduction of effect handling before, **`break`** marks the end of the do block, which provides the value of the do block and passes the control flow back to the handler as if **`resume`** is a function and has just returned which continues the *remaining code* in handler after **`resume`**. **`break`** is expressed with a function returning in the do block lambda function as on line 6 of Figure 3.10 for our previous example of simplified counter in Figure 2.6.



(a) Stack when executing break

(b) Stack desired

Fig. 3.12: Call stack for simplified counter at break

With the function returning of the do block, the function `do_handle` gains control flow again, sets the yield value field to the argument provided to **`break`** and declares a scope guard to execute the additional logic for effect handling and returns the result of the do-handle block from the yield value field of the handler frame to conclude it, as in C++ the destructors (scope guards) are executed before the function returns, and scope guards ensures the execution for the logic in case of a stack unwinding used in section 3.5.

To properly conclude a do block, the implementation of the scope guard contains two parts as briefly introduced in section 3.1 before, the execution of remaining code in the handler and the deregistration of the handler frame, while properly maintain the current evaluation result value of the do-handle block in the yield value field of the handler frame. In summary, the implementation of break contains the following steps.

1. Set the yield value field to the argument provided to **`break`** (expressed as **`return`**).
2. Execute the steps in the scope guard.
   (a) Restore the handler vector from the preserved copy of handler frames in the resumption frame.
   (b) Execute the remaining code in the handlers.
   (c) Deregister the handler frame.
3. Return the value from the yield value field as the result of the do-handle block.

Restoring the handler vector is straightforward as copying the value back to the global handler vector. The deregistration of handler frame is also as straightforward as popping the last handler frame from the handler vector since the correspondence between the frame of `do_handle` and the pushing to the handler vector. However, executing the remaining code is not trivial, which is described below.

To execute the remaining code, we have saved the execution state of the handler in a resumption frame with the technique explained in section 3.3. It first pops the frame out from the stack of resumption frames in the current handler frame, and the remaining work is to restore this resumption frame. The current stack is as Figure 3.12a, and we want to restore the stack frames highlighted in yellow in Figure 3.12b which is saved before as in Figure 3.11a. It is obvious that the frame of "scope guard" overlaps with the stack frames we want to restore, and `longjmp` could only jump downwards to a frame recalling the previous introduction of setjmp/longjmp.

Therefore, we first shift the stack upwards in the scope guard like the arrow (1) shows in Figure 3.13, by subtracting the stack pointer by $nsp - sp + |saved\ stack|$. The nsp is the current stack pointer of the "scope guard" frame which points to the top of the frame, sp is the stack pointer of `do_handle`, and $|saved\ stack|$ is the size of the saved stack which contains the frames between "handler for `count`[3]" and "`counter(3)`". The shift forms

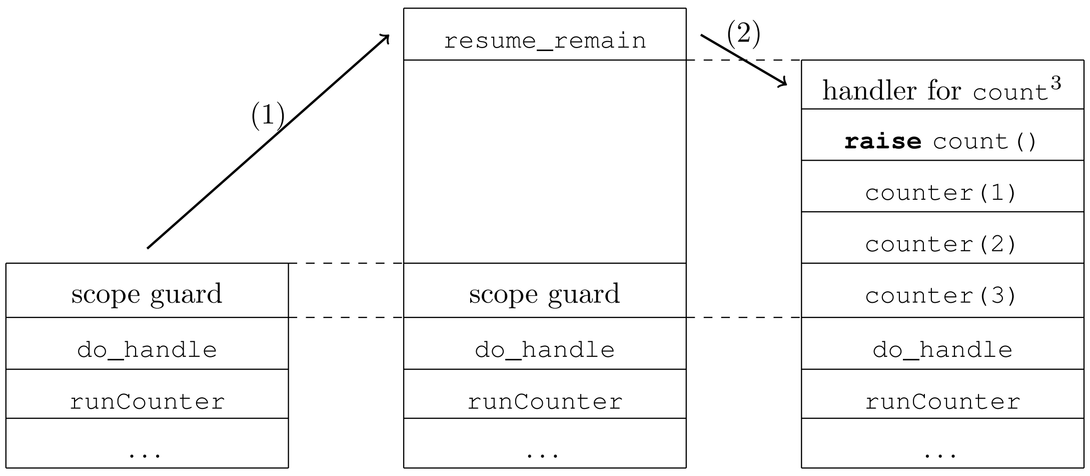


Fig. 3.13: Restore the stack for executing remaining code

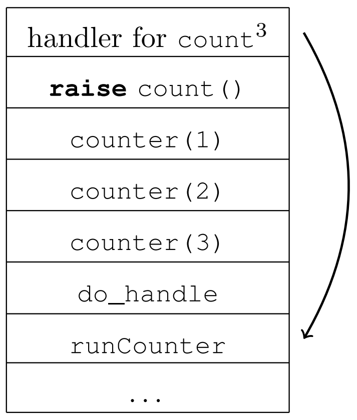


Fig. 3.14: Control flow for yield

the stack shaped in the middle in Figure 3.13, and then it invokes `resume_remain`. Then `resume_remain` jumps to the “handler for `count3`” by using longjmp to switch to the checkpoint saved before with the **`resume`** as in section 3.3, passing control flow back to the remaining code in the handler.

The control flow is passed back to the scope guard later by a **`yield`** from the handler which is demonstrated later in section 3.5. Therefore, it executes all of the remaining codes in handlers one-by-one in the order introduced before in the background as the resumption frames forms a stack.

3.5 Implementation of yield

As a final point, **`yield`** indicates the end of an effect handler and gives return value to the do-handle block, which requires jumping backwards over several frames on the call stack as in Figure 3.14 for our previous example of simplified counting function invocations Figure 2.6.

In eff-unwind, similarly to resume, there is a yield context passed to the effect handler as an argument. The yield context is indeed a class with template parameters and fields to track the components required by performing yielding, which are as follows.

- Template parameter *effect type* `Effect`: The type of the effect.
- Template parameter *yield type* `yield_t`: The type of the result of do-handle block.
- Field *handler frame reference* `handler_frame`: A reference to the handler frame of the invoked handler.

The yield context likely overloads the function call operator (**`operator`**`()`), so in order to perform a yielding, the handler code invokes the yield context similarly to a function, passing a parameter of do-handle block result type, like on line 10 in Figure 3.2b. Since the yield context is created after a concrete handler is found, it knows the do-handle result type from the specialized `handler_frame_yield`.

When the handler calls **`yield`**, it first updates the yield value field of the handler frame to the argument of **`yield`**, and then uses stack unwinding to obtain the execution context of the target frame.

3.5.1 Stack unwinding

To give some background, stack unwinding is a technique introduced by C++ exception handling for restoring the execution context of some functions below the current calling frame, the target of which is similar to longjmp. The execution context contains the registers and the memory, where the memory is not required to be restored since it is going downwards on the stack. Therefore, stack unwinding is about restoring the registers for a frame downwards the stack, without saving these registers beforehand which differs from setjmp/longjmp.

To jump to the target frame, it requires changing the program counter (or named as instruction pointer), and the data registers. We know that the registers are split into two kinds, caller-saved and callee-saved, based on their preservation across function calls. Caller-saved registers are preserved by the caller, and is expected to be clobbered by the callee. Callee-saved registers, in contrast, is preserved by the callee, and is expected to be restored before returning to the caller. Additionally, the return address, which is the program counter of caller, should also be preserved by the callee. Since we are going to a frame downwards (from callee to caller), the caller-saved registers are not concerned but the callee-saved registers and the return address are important.

In C++ (and programs with unwinding), the compiler generates a *unwind table* which tells the place where the function saves callee-saved registers and return address. Therefore, we could restore the registers of caller by using this table and reading from the callee's registers and the memory.

To use stack unwinding, it requires initializing an *unwind context* by calling the function `unw_getcontext`, which parses the unwind table and sets up the essential data structures. Then, an *unwind cursor*, which represents an execution context, is possible to be obtained by calling `unw_init_local` which gives the cursor pointing to the current execution context (and current frame). By using `unw_step`, it moves the cursor framewise downwards the stack. The registers of the execution context in the cursor can be read out by `unw_get_reg` and set by `unw_set_reg`.

Additionally, `unw_get_proc_info` reads out the information (*procedure information*) related to the function containing the instruction which the execution context in unwind cursor. The procedure information includes the boundary of program counter of the function, address of language-specific data area, address of *personality routine* and other information related to the unwind table.

Personality routine is related to the exception semantics of C++. By invoking it with different unwind action `_Unwind_Action` parameter, it changes the unwind cursor passed as one of the parameters instructing the code to be executed in different scenarios. One

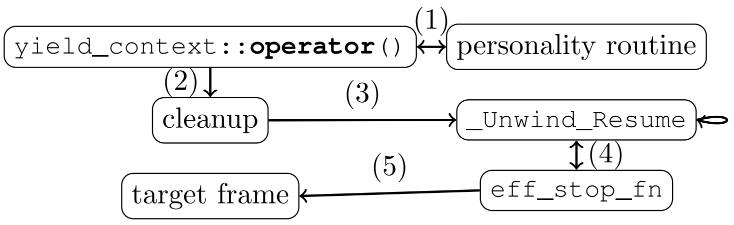


Fig. 3.15: Control flow of yield with destructors

of the scenarios includes executing the non-trivial destructors for the values contained in the frame, where it changes the unwind cursor to the cleanup code of that frame.

Finally, after making the cursor reaching some designated execution context, it *resume*s the execution context specified by the unwind cursor with `unw_resume`, which restores the registers and jumps to the instruction defined in the program counter of the cursor.

3.5.2 Stack unwinding for yielding

Stack unwinding is chosen for two reasons. First, some of other "jumping" techniques such as setjmp/longjmp or bubbling requires additional setup over code path where there are no **`yield`**, which introduces performance penalty over tail-resuming cases. Second, some of these techniques do not interoperate with C++ destructors well, since they only restores registers and do not execute destructors over the frames they jump over, leading to unexpected behavior such as resource leaking. Stack unwinding is a mature technique used in C++ exception handling for a long time, which is optimized for the "normal" path and handles destructors, so it suits the need for yielding well and is thus chosen.

Implementation without invoking non-trivial destructor

To start with simplicity, let all the frames not have a non-trivial destructor. Briefly speaking, it steps downwards the stack frame-by-frame and stops at the frame of the caller of `do_handle`, which marks the returning point of `do_handle`. The process of yielding are as follows.

1. Initialize the unwind context, cursor and exception object.
2. Step the cursor down by a frame.
3. Compare the frame pointer of current cursor to the resumption frame pointer field of the handler frame, which points to the frame of the caller of `do_handle`.
   - If not equals, go back to step 2 to continue the stepping process.
   - If equals, continues with the next step.
4. Sets the returning registers of the cursor to the result of do-handle, which comes from the yield value field of the handler frame. This gives the return value of do-handle block.
5. Resume the caller of `do_handle` by `unw_resume`.

Most parts of the procedure is trivial, but the exception object is initialized a little differently from C++ exception handling. The exception class is initialized to `"XFOXEH␣␣␣"` (`0x58464f5845480000`) which distinguishes itself from C++ exceptions. The other fields could be uninitialized for the current simplified scenario.

Handling non-trivial destructors

To go deeper, if one of the frames stepped over contains a non-trivial destructor, it becomes more complex as destructors of that frame need to be executed. Therefore, between step

2 and 3, it invokes the personality routine defined in the procedure info of the frame to check if there are any cleanup (destructors) associated with the frame, which is marked as the edge (1) in Figure 3.15. If there is a cleanup, the unwinding cursor is updated to point to the execution context of the cleanup code. **`yield`** resumes to that unwinding cursor to execute the cleanup code as the edge (2) in the figure.

However, the cleanup function does not return back to **`yield`**, and actually it cannot since the stack frames above the frame containing the cleanup is no longer valid after resuming to the cleanup. It hands over to the stack unwinding process of C++ as tail calling `_Unwind_Resume` which is marked as the edge (3) in Figure 3.15. `_Unwind_Resume` steps over all the frames below it and expects to reach some frame containing a matching exception handler, which in our case does not exist at all as there is no C++ exception.

Therefore, we introduce a stopping function which is invoked when `_Unwind_Resume` steps at a new frame. The stopping function checks if the current frame is our target frame, and resumes the current cursor when matching to jump out of the C++ stack unwinding process to the returning point of `do_handle`, which is marked as edge (5) in Figure 3.15. Additionally, it sets the returning registers of the cursor before resuming to pass the result value of do-handle block to the caller.

This depicts the entire flow of yielding. Considering the entire implementation, additionally for the exception object, the first private field `private_1` is initialized to the function address of our custom stopping function which is required by C++ stack unwinding ABI. The second private field `private_2` is set to the target frame pointer (the resumption frame pointer field of the handler frame) for storing our target frame which is also required by C++ stack unwinding ABI. The exception cleanup `exception_cleanup` is set to the *yielding frame context* since the field is not used for stack unwinding process of non-exception cases. The yielding frame context is constructed when yielding, and contains the result value of the do-handle block.

Another point is that since **`break`** is implemented as a scope guard in `do_handle`, **`yield`** gives control flow back to the scope guard by the execution of cleanup in `do_handle`, which eventually executes the remaining code of all the handlers.

Chapter 4

Evaluation

In order to illustrate the performance implications of the unbalanced implementation, we conduct evaluation based on well-known applications of effect handlers which each contains tens of lines of code and compare the execution time of our library against the existing works. Our evaluation is constructed in order to answer the following two questions.

- Does our approach improve the performance of the invocation of tail resumptive handlers, and if so, how much?
- How much does our approach worsen the performance of other kinds of effect handlers?

We first takes the test cases from [3], which includes the following.

- *countdown* is the case previously shown in Figure 2.8 where there are only tail resumptive handlers.
- *fibonacci recursive* calculates fibonacci by recursion without using effects, giving insights into the performance of the platform.
- *generator* sums the total number in a binary tree by using first-class resumption. It stores the resumption after the handler exits and resumes the execution later to continuously emit numbers.
- *handler sieve* computes the sum of all primes less than n by trial division, which has been shown in Figure 3.6.
- *iterator* creates the effect *emit* which is triggered over each element of a container, and the tail resumptive handler computes the sum of the emitted values.
- *nqueens* traverses the solution space of a nqueens problem by using multishot handlers.
- *parsing dollars* generates n lines which contains i `$` characters for the i-th line. The function `parse` consumes `$` and yields the count in each line as an effect. The function `sum` receives the yielded count and gives the total count of `$`s in the generated file.
- *product early* computes the product of all the numbers between $1000 \ldots 0$, where the effect *abort* is raised when encountering a 0 and the yielding handler short-circuits the result to be 0.
- *resume nontail* is the case previously shown in Figure 2.21 where a non-tail-resumptive handler is used to change the return value.
- *tree explore* explores a binary tree similarly to *generator*.
- *triples* counts the total number of triples (t_1, t_2, t_3) where $t_1 > t_2 > t_3$ and $t_1 + t_2 + t_3 = s$.

We run these cases over the following implementations.

- *eff-unwind* is our implementation of the proposed approach as a library in C++.

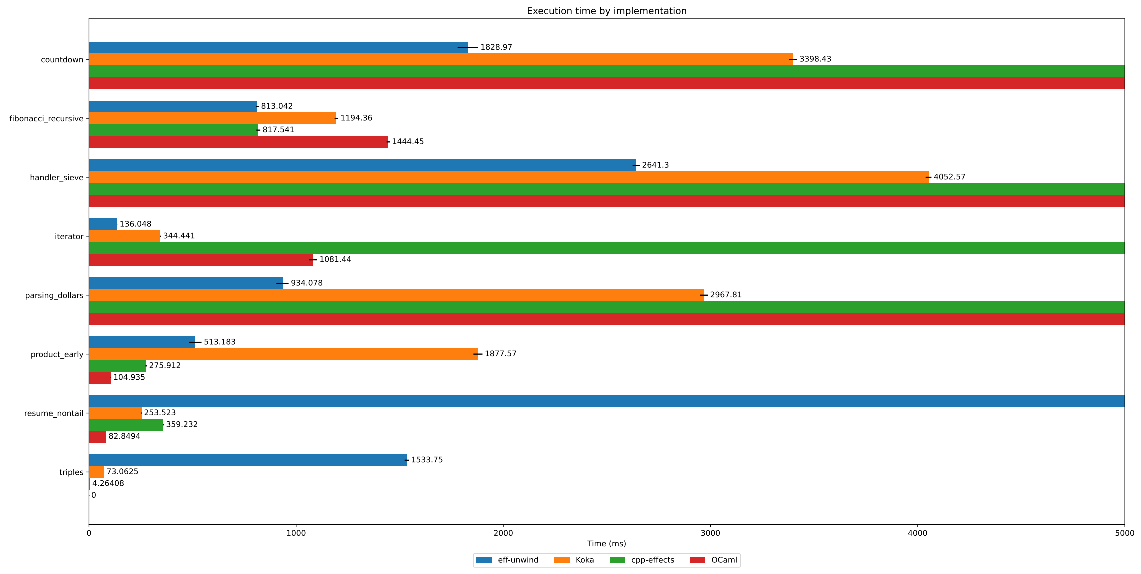


Fig. 4.1: Execution time averaged over 10 runs

Time (ms)	eff-unwind	Koka	cpp-effects	OCaml
countdown	1828.968	3398.433	74886.605	10146.831
fibonacci recursive	813.042	1194.364	817.541	1444.449
handler sieve	2641.296	4052.567	49177.091	8187.453
iterator	136.048	344.441	6413.194	1081.44
parsing dollars	934.078	2967.811	31968.717	6507.325
product early	513.183	1877.568	275.912	104.935
resume nontail	6121.71	253.523	359.232	82.849
triples	1533.755	73.063	N/A	N/A

Table 4.1: Execution time averaged over 10 runs

- *cpp-effects* [4] is an implementation of effect handling in C++ with segmented stack achieved by coroutines.
- *Koka* [5] is a programming language with effect handling support by monadic translation.
- *OCaml* [6] is the first industry programming language with effect handling support. Its implementation is based on stack manipulation.

We omit the cases of *generator* and *tree explore* since we do not support stored resumption after the handler yields and *nqueens* since it has problems with the lifetime of stack frames.

The experiments are conducted on a MacBook Pro with M1 Pro chip (AArch64) and 32 GiB memory running macOS Sequoia (15.2), with Apple clang 16.0.0, CMake 3.31.2, ninja 1.12.1, Koka v2.6.0, OCaml 5.2.1 and boost 1.87.0. We measure the execution time of these programs and calculate mean and standard deviation by 10 times. The results are in Figure 4.1, Table 4.1 and Table 4.2, of which the discussion is given below.

Baseline

fibonacci recursive: Our implementation is about 1.5 times faster than Koka, 1.8 times faster than OCaml and almost the same as cpp-effects. As we are comparing mostly

Time (ms)	eff-unwind	Koka	cpp-effects	OCaml
countdown	49.583	20.315	770.817	107.507
fibonacci recursive	6.393	8.741	9.41	7.717
handler sieve	17.21	14.046	1235.877	78.838
iterator	0.897	2.406	47.577	19.975
parsing dollars	29.458	19.238	596.058	20.175
product early	30.172	21.855	3.039	1.381
resume nontail	54.599	1.126	1.72	0.584
triples	10.081	0.786	N/A	N/A

Table 4.2: Standard deviation of execution time over 10 runs

between different programming languages, this benchmark case reveals the possible performance difference between these different platforms, eventually being a possible baseline.

Tail resumptive handlers

countdown: Our implementation is about 1.9 times faster than Koka, 5.5 times faster than OCaml and 40.9 times faster than cpp-effects. In this use case, there are two effects `get` and `set`, both of which are handled by a tail resumptive handler as shown before in Figure 3.2. This case shows that our implementation achieves better performance than the existing works evaluated in high confidence, but since it is comparing between different programming languages, the conclusion is not definite.

handler sieve: Our implementation is about 1.5 times faster than Koka, 3.1 times faster than OCaml and 18.6 times faster than cpp-effects. As shown before in Figure 3.6, this case uses a tail resumptive handler, and we are achieving a similar performance improvement over *fibonacci recursive*. Considering the fact that the handlers are invoked recursively, and Koka incorporates tail recursion optimization over the recursive call [5] while we, being a C++ library, cannot perform that optimization, our effect handling mechanism is believed to be as efficient as Koka and more efficient than the other implementation compared. Additionally, this reveals a weakness of our implementation being not supportive to tail recursion optimization over effect handlers.

iterator: Our implementation is about 2.5 times faster than Koka, 7.9 times faster than OCaml and 47.1 times faster than cpp-effects. As shown in Figure 4.2, this case has the effect `emit` which emits an integer, and the handler sums all the emitted integers without imposing any computations after resuming, being a tail resumptive handler. It shows that our implementation has performance improvement over the existing works compared on tail resumptive handlers.

parsing dollars: Our implementation is 3.2 times faster than Koka, 7.0 times faster than OCaml and 34.2 times faster than cpp-effects. As shown in Figure 4.3, this case has 3 effects, `read`, `emit` and `stop`. `read` simulates reading from a file, where the file contains n lines, and there are i `$`s on the i-th line, and it is handled by a tail resumptive handler. `emit` is similar to our previous example *iterator*, where the code emits the number of `$`s for each line, and the handler sums the count up to get a total number of dollars for the entire file, which is also tail-resumptive. `stop` is similar to throwing an exception, where the code aborts the execution if it encounters a character which is neither a dollar nor a newline, or if it reaches the end of the simulated input file, where an yielding handler is used. This case is more complex than the previous ones while the invocations to tail resumptive handlers is still dominating, and our implementation shows performance improvement over the existing works compared.

```
effect emit(e: int)

func range(l, r)
  for e in l..r {
    raise emit(e)
  }

func main(n)
  sum := 0
  do {
    range(0, n)
    break s
  } handle { emit(e) {
    sum += e
    yield(resume())
  } }
```

Fig. 4.2: Iterator

Non-tail-resumptive handlers

product early: Our implementation is 1.9 times slower than cpp-effects and 4.9 times slower than OCaml, but 3.7 times faster than Koka. This case is similar to the exception handling case introduced before in Figure 2.9, where there is a single yielding handler aborting the production of a list of numbers when encountering a 0. This case shows that the implementation technique of Koka, which uses bubbling instead of stack unwinding for a deeply recursive call stack, has worse performance than our implementation, but using stack manipulating has better performance than our technique of stack unwinding, which we believe is related to the heavy overhead of parsing the unwinding table and invoking the personality function.

However, if we construct some object with non-trivial destructor in the code of *product early* and output some text to check if the non-trivial destructors are correctly executed, a surprising finding is that the output of cpp-effects's version does not contain the logging from the destructor, while our implementation eff-unwind contains. Therefore, it shows that our implementation successfully maintains the semantics of non-trivial destructors of the objects referenced in effect handling, while cpp-effects does not.

resume nontail: Our implementation is 17.0 times slower than OCaml, 24.1 times slower than Koka and 73.9 times slower than cpp-effects. This case has been introduced before in Figure 2.21, where there is a handler with remaining code. This case shows that our approach of stack copying and setjmp/longjmp has a enormous performance penalty over the existing implementations compared, which means our implementation is extremely biased for tail resumptive handlers.

triples: Our implementation is 21.0 times slower than Koka, but OCaml and cpp-effects do not support multishot handlers. As shown in Figure 4.4, this case has an effect `flip` which is handled by a multishot handler on line 31 - 33 where it permutes over the possible results of flipping a coin. Our implementation technique of stack copying and setjmp/longjmp has successfully supported multishot handlers while OCaml and cpp-effects do not, but our performance is significantly slower than Koka.

```
effect read() -> char_t
effect emit(int)
effect stop()

func newline() { return 10; }
func is_newline(c) { return c == 10; }
func dollar() { return 36; }
func is_dollar(char_t c) { return c == 36; }

func parse() {
  a, c := 0
  while ((c = raise read()) != 0) {
    if (is_dollar(c)) { a += 1 }
    else if (is_newline(c)) {
      raise emit(a)
      a = 0
    } else { raise stop() }
  }
}

func feed(int n)
  i, j := 0
  do {
    parse()
    break
  } handle { read() {
    if (i > n) { raise stop() }
    else if (j == 0) {
      i += 1
      j = i
      yield(resume(newline()))
    } else {
      j -= 1
      yield(resume(dollar()))
    }
  } }

func sum(n)
  s = 0
  return do {
    do { feed(n) } handle { stop() {
      yield()
    } }
    break s
  } handle { emit(e) {
    s += e
    yield(resume())
  } }
```

Fig. 4.3: Parsing dollars

```
effect flip() -> bool
effect fail() -> int

func choice(n)
  while (n >= 1) {
    if (raise flip()) {
      return n
    } else {
      n -= 1
    }
  }
  raise fail()

func triple(n, s)
  i := choice(n)
  j := choice(i - 1)
  k := choice(j - 1)
  if (i + j + k == s) {
    return {i, j, k}
  } else {
    raise fail()
  }

func hash({a, b, c})
  return (53 * a + 2809 * b + 148877 * c) % 1000000007

func run(n, s)
  return do {
    break hash(triple(n, s))
  } handle { flip() {
    v1 := resume(true)
    v2 := resume(false)
    yield((v1 + v2) % 1000000007)
  } fail() {
    yield(0)
  } }
```

Fig. 4.4: Triples

Unsupported cases

generator and *tree explore* is not supported, where it stores the resumption in a handler to be restored later where the handler has already yielded. In our implementation, **`resume`** is only possible to be called within the scope of the handler and cannot be copied or moved as a normal object.

nqueens is not supported since there is problem with vector, a C++ STL object with non-trivial destructors. If multishot handlers are used in conjunction with objects with non-trivial destructors, the timing for invoking the destructors are not trivial to be determined, where memory safely problems such as double freeing arises, which reveals some

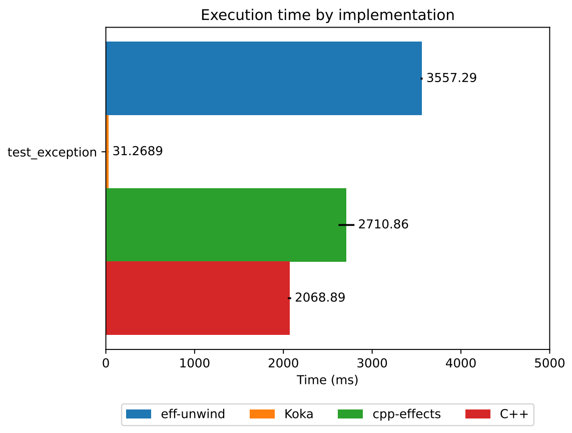


Fig. 4.5: Execution time averaged over 10 runs

challenges for introducing effect handling into C++.

Exception handling

We also conducted comparison against Koka, cpp-effects and vanilla C++ for exception handling [4], whose result is shown in Figure 4.5. In the case, a function is executed for a million times where each invocation throws an exception. The result shows that our implementation has similar performance characteristics like cpp-effects and vanilla C++ but is significantly slower than Koka. This reveals more insights over the performance degrading for non-tail-resumptive cases, despite being unrealistic since exceptions should be an extremely rare case instead of happening at every invocation.

Summary

With the dataset in [3], it shows that our implementation has about 1.5 - 3.2 times faster than state-of-the-art works where invocations to tail resumptive handlers are taking the majority, while being 1.9 - 73.9 times slower than SOTA works for non-tail-resumptive handlers.

We are supporting multishot handlers in comparison to OCaml and cpp-effects, but we do not support first-class resumption cases such as *generator* and *tree explore*.

We are additionally keeping the non-trivial destructor semantics executed over cpp-effects, which better ensures the C++ semantics.

Chapter 5

Conclusion

5.1 Summary

To sum up this thesis, algebraic effect handling [1, 2] is a new programming abstraction for non-local control flows, whose performance need to be improved. Especially, we believe that the performance of tail resumptive handlers takes priority over the others based on the hypothesis that the invocations to such kind of handlers are more frequent than yielding and non-tail-resumptive ones. As a result, we proposes an implementation technique for effect handling which is optimized for tail resumptive handlers at the cost of the performance of others. It enables the efficient execution of **`raise`** and tail resuming with function call and returning by adopting stack unwinding, stack copying and setjmp/longjmp for the operators not used in tail resumptive handlers.

The proposed technique has been carried out as a C++ library, and we have conducted evaluation against SOTA works such as *cpp-effects* [4], *Koka* [5] and *OCaml* [6] based on a dataset containing well-known use cases [3] and a case from existing work [4]. The result shows that our implementation has achieved improved performance over all the SOTA works for tail resumptive cases while has a significant degradation for non-tail-resumptive cases. Additionally, we are the first to propose an implementation technique for C++ which supports multishot effect handlers and we are keeping the semantics of non-trivial destructors in C++ correct compared to existing C++ implementations.

We believe that this implementation technique provides an insight into the implementation of effect handling which illustrates the importance of tail resumptive handlers and shows the effectiveness of optimizing for tail resumptive handlers. Despite that there is no real-world large applications with effect handling, we believe that we have given an explanation of how people would use different kinds of effect handlers, and the eventual focus of implementation based on that hypothesis.

5.2 Future work

There are still a list of unresolved problems for our implementation. First, the library does not support first-class resumption, where resumption object is allowed to be saved and executed later, which empowers cases such as *generator* and *tree explore.* From a first glance, it is possible to save the preserved stack and jump buffer to an object and restore it later, but the execution timepoint of non-trivial destructors of the variables contained in that resumption object need to be considered thoughtfully.

Second, the user has to manually distinguish tail resumptive handlers by using a separate interface to enable the optimization, which is due to our limitation of being a library. We believe that it is trivial to automatically distinguish different kinds of effect handlers if it is implemented as part of the compiler, which Koka [13] has successfully achieved.

Finally, despite we have identified the problem of the execution of non-trivial destructors in multishot handlers, we do not come up with a solution, which blocks the case *nqueens*. It is straightforward to introduce some kind of reference counting to frames and execute the non-trivial destructors as the reference count goes to 0, but first, the programmer may expect that the destructors are executed once for each resumption if the destructors are used for logging or similar scenarios, and second, the trivial workaround imposes a heavy overhead to the program. Therefore, we are leaving this problem to the future.

Publications and Research Activities

(1) Yuze Fu, Tetsuro Yamazaki, Shigeru Chiba. Exploring possibility of using stack unwinding for effect handlers in C++. Programming and Programming Language Workshop (PPL), March 3-5, 2024. (Poster)

References

[1] Gordon D. Plotkin and Matija Pretnar. Handlers of algebraic effects. In Giuseppe Castagna, editor, *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, volume 5502 of *Lecture Notes in Computer Science*, pages 80–94. Springer, 2009.

[2] Gordon D. Plotkin and Matija Pretnar. Handling algebraic effects. *Log. Methods Comput. Sci.*, 9(4), 2013.

[3] Codrin Iftode, KC Sivaramakrishnan, Daniel Hillerström, Filip Koprivec, phischu, Jesse Sigal, Sam Lindley, Jonathan Lindegaard Starup, and Teodoro Freund. effect-handlers/effect-handlers-bench: Benchmark repository of polyglot effect handler examples. `https://github.com/effect-handlers/effect-handlers-bench`, 2024. [Accessed 2024-07-26].

[4] Dan R. Ghica, Sam Lindley, Marcos Maroñas Bravo, and Maciej Piróg. High-level effect handlers in C++. *Proc. ACM Program. Lang.*, 6(OOPSLA2):1639–1667, 2022.

[5] Ningning Xie and Daan Leijen. Generalized evidence passing for effect handlers: efficient compilation of effect handlers to C. *Proc. ACM Program. Lang.*, 5(ICFP):1–30, 2021.

[6] K. C. Sivaramakrishnan, Stephen Dolan, Leo White, Tom Kelly, Sadiq Jaffer, and Anil Madhavapeddy. Retrofitting effect handlers onto ocaml. In Stephen N. Freund and Eran Yahav, editors, *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, pages 206–221. ACM, 2021.

[7] Syouki Tsuyama, Youyou Cong, and Hidehiko Masuhara. An intrinsically typed compiler for algebraic effect handlers. In Gabriele Keller and Meng Wang, editors, *Proceedings of the 2024 ACM SIGPLAN International Workshop on Partial Evaluation and Program Manipulation, PEPM 2024, London, UK, 16 January 2024*, pages 134–145. ACM, 2024.

[8] Daniel Hillerström and Sam Lindley. Shallow effect handlers. In Sukyoung Ryu, editor, *Programming Languages and Systems - 16th Asian Symposium, APLAS 2018, Wellington, New Zealand, December 2-6, 2018, Proceedings*, volume 11275 of *Lecture Notes in Computer Science*, pages 415–435, Wellington, New Zealand, 2018. Springer.

[9] Philipp Schuster, Jonathan Immanuel Brachthäuser, and Klaus Ostermann. Compiling effect handlers in capability-passing style. *Proc. ACM Program. Lang.*, 4(ICFP):93:1–93:28, 2020.

[10] Daniel Hillerström, Sam Lindley, Robert Atkey, and K. C. Sivaramakrishnan. Continuation passing style for effect handlers. In Dale Miller, editor, *2nd International Conference on Formal Structures for Computation and Deduction, FSCD 2017, September 3-9, 2017, Oxford, UK*, volume 84 of *LIPIcs*, pages 18:1–18:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017.

[11] Wenhao Tang, Daniel Hillerström, Sam Lindley, and J. Garrett Morris. Soundly handling linearity. *Proc. ACM Program. Lang.*, 8(POPL):1600–1628, 2024.

[12] Institut National de Recherche en Informatique et en Automatique. Effect handlers, the ocaml language. `https://ocaml.org/manual/5.2/effects.html`, 2024. [Accessed 2024-12-18].

[13] Microsoft Research and Daan Leijen. The koka programming language. `https://koka-lang.github.io/koka/doc/book.html`, 2025. [Accessed 2025-01-03].

[14] The Flix contributors. The flix programming language. `https://flix.dev/`, 2025. [Accessed 2025-01-05].

[15] Luna Phipps-Costin, Andreas Rossberg, Arjun Guha, Daan Leijen, Daniel Hillerström, K. C. Sivaramakrishnan, Matija Pretnar, and Sam Lindley. Continuing webassembly with effect handlers. *Proc. ACM Program. Lang.*, 7(OOPSLA2):460–485, 2023.

[16] Daan Leijen. Implementing algebraic effects in C - "monads for free in c". In Bor-Yuh Evan Chang, editor, *Programming Languages and Systems - 15th Asian Symposium, APLAS 2017, Suzhou, China, November 27-29, 2017, Proceedings*, volume 10695 of *Lecture Notes in Computer Science*, pages 339–363. Springer, 2017.

[17] Andrej Bauer and Matija Pretnar. Programming with algebraic effects and handlers. *J. Log. Algebraic Methods Program.*, 84(1):108–123, 2015.

[18] Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. Binders by day, labels by night: effect instances via lexically scoped handlers. *Proc. ACM Program. Lang.*, 4(POPL):48:1–48:29, 2020.

[19] Lukas Convent, Sam Lindley, Conor McBride, and Craig McLaughlin. Doo bee doo bee doo. *J. Funct. Program.*, 30:e9, 2020.

[20] Daniel Hillerström, Sam Lindley, and Robert Atkey. Effect handlers via generalised continuations. *J. Funct. Program.*, 30:e5, 2020.

[21] Mario Alvarez-Picallo, Teodoro Freund, Dan R. Ghica, and Sam Lindley. Effect handlers for C via coroutines. *Proc. ACM Program. Lang.*, 8(OOPSLA2):2462–2489, 2024.

[22] Daan Leijen. Type directed compilation of row-typed algebraic effects. In Giuseppe Castagna and Andrew D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 486–499. ACM, 2017.

[23] Daan Leijen. Koka: Programming with row polymorphic effect types. In Paul Blain Levy and Neel Krishnaswami, editors, *Proceedings 5th Workshop on Mathematically Structured Functional Programming, MSFP@ETAPS 2014, Grenoble, France, 12 April 2014*, volume 153 of *EPTCS*, pages 100–126, 2014.

[24] Nicolas Wu, Tom Schrijvers, and Ralf Hinze. Effect handlers in scope. In Wouter Swierstra, editor, *Proceedings of the 2014 ACM SIGPLAN symposium on Haskell, Gothenburg, Sweden, September 4-5, 2014*, pages 1–12. ACM, 2014.

[25] Daniel Hillerström and Sam Lindley. Liberating effects with rows and handlers. In James Chapman and Wouter Swierstra, editors, *Proceedings of the 1st International Workshop on Type-Driven Development, TyDe@ICFP 2016, Nara, Japan, September 18, 2016*, pages 15–27. ACM, 2016.

[26] Ningning Xie and Daan Leijen. Effect handlers in haskell, evidently. In Tom Schrijvers, editor, *Proceedings of the 13th ACM SIGPLAN International Symposium on Haskell, Haskell@ICFP 2020, Virtual Event, USA, August 7, 2020*, pages 95–108. ACM, 2020.

[27] Ningning Xie, Jonathan Immanuel Brachthäuser, Daniel Hillerström, Philipp Schuster, and Daan Leijen. Effect handlers, evidently. *Proc. ACM Program. Lang.*, 4(ICFP):99:1–99:29, 2020.

[28] Georgios Karachalias, Filip Koprivec, Matija Pretnar, and Tom Schrijvers. Efficient compilation of algebraic effect handlers. *Proc. ACM Program. Lang.*, 5(OOPSLA):1–

28, 2021.

[29] Oleg Kiselyov and K. C. Sivaramakrishnan. Eff directly in ocaml. In Kenichi Asai and Mark R. Shinwell, editors, *Proceedings ML Family Workshop / OCaml Users and Developers workshops, ML/OCAML 2016, Nara, Japan, September 22-23, 2016*, volume 285 of *EPTCS*, pages 23–58, 2016.

[30] Meta. Built-in react hooks. `https://react.dev/reference/react/hooks`, 2024. [Accessed 2024-12-18].

[31] Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. Effect handlers for the masses. *Proc. ACM Program. Lang.*, 2(OOPSLA):111:1–111:27, 2018.

[32] Yizhou Zhang and Andrew C. Myers. Abstraction-safe effect handlers via tunneling. *Proc. ACM Program. Lang.*, 3(POPL):5:1–5:29, 2019.

[33] Ricardo Abreu and contributors. ricab/scope_guard. `https://github.com/ricab/scope_guard`, 2023. [Accessed 2025-01-05].

[34] cppreference.com contributors. setjmp - cppreference.com. `https://en.cppreference.com/w/cpp/utility/program/setjmp`, 2025. [Accessed 2025-01-06].

[35] cppreference.com contributors. longjmp - cppreference.com. `https://en.cppreference.com/w/cpp/utility/program/longjmp`, 2025. [Accessed 2025-01-06].

Acknowledgements

Time flies, and the past two years of my master’ s studies have passed in the blink of an eye. Here, I would like to express my heartfelt gratitude to everyone who has supported and helped me during this journey.

First and foremost, I am deeply grateful to my professor, Professor Shigeru Chiba. Your education, guidance, encouragement, and support in your courses have made these two years truly rewarding. During this time, I not only gained professional knowledge but also learned the methods of scientific research. Although I may not have achieved significant results, as a student with much to improve, I deeply appreciate the meaningful and enriching two years you have given me.

Next, I extend my sincere thanks to my parents. Your unconditional love has taught me to cherish life, your care and support have enabled me to persevere to this day, and the curiosity and sincerity you instilled in me have strengthened my resolve to pursue my studies.

Lastly, I wish to thank Dr. Yamazaki, my lab colleagues and friends. Thank you for the discussions we shared about knowledge, technology, and hobbies.

These brief two years will forever remain a cherished memory in my heart.