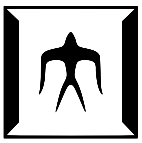

TOKYO INSTITUTE OF TECHNOLOGY

DOCTORAL/MASTER THESIS

Extending Effekt with Bidirectional Effects

Author:
Kazuki NIIMI

Student Number:
19M30318

Supervisor:
Prof. Hidehiko MASUHARA

A thesis submitted in fulfillment of the requirements for the degree of Master of Science

in the

Programming Research Group
Department of Mathematical and Computing Science

April 1, 2021

Declaration of Authorship

I, Kazuki NIIMI, declare that this thesis titled, "Extending Effekt with Bidirectional Effects" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

Date:

TOKYO INSTITUTE OF TECHNOLOGY

Abstract

School of Computing
Department of Mathematical and Computing Science

Master of Science

Extending Effekt with Bidirectional Effects

by Kazuki NIIMI

Algebraic effects and handlers (proposed by Plotkin et al.) enable implementing various computational effects, such as exception handlers and side effects, in a uniform and effect-safe manner. Our goal is to allow manipulation of complex control flow in a fully-fledged language with algebraic effects. To achieve this goal, we introduce bidirectional effects to the Effekt language. Bidirectional effects (proposed by Zhang et al.) are a novel concept that makes it possible to implement bidirectional control flow needed by generators and async-await.

Effekt natively supports algebraic effects and it has practical features such as records, modules, mutable variables, and inputs/outputs. It also has a compiler and a runtime system. Moreover, Effekt has a simple and easily extensible type system by special treatment of effect polymorphism. Effect polymorphism is required to support higher-order functions in languages with algebraic effects. Effekt realizes this without parametric effect polymorphism, which is difficult to understand for users.

We extend the Effekt compiler with bidirectional effects and formalize the extended type system. We also show that this extension allows implementation of generators and communication, which were not possible in the original Effekt language.

Acknowledgements

I would like to thank H. Masuhara and Y. Cong for ideas and discussions. I would also like to thank J. Brachthäuser for many advice of Effekt. I am grateful to the members of the Programming Research Group for useful comments.

Contents

List of Figures

Chapter 1

Introduction

Algebraic effects and handlers (Plotkin and Pretnar, 2013) enable implementing various computational effects, such as exception handlers and side effects, in a uniform and effect-safe manner. Effect safety means that an effect is never accidentally handled by a wrong handler or not handled by any handlers. This is useful for the real world programming such as exceptions of like Java or input/output.

However, some programs cannot be written with plain algebraic effects. Two such examples are generators and async-await, which contain complex control flow and are currently in vogue for programming languages (Zhang et al., 2020).

Our goal is to allow manipulation of complex control flow in a fully-fledged language with algebraic effects. To achieve this goal, we introduce *bidirectional algebraic effects* to Effekt language. Bidirectional algebraic effects (Zhang et al., 2020) are a novel concept that makes it possible to implement bidirectional control flow needed by generators and async-await. The difference between plain algebraic effects and bidirectional algebraic effects is in the direction of effect propagation. With plain algebraic effects, an effect propagates up the dynamic call stack to its handler. With bidirectional algebraic effects, an effect propagates to the location where the initiating effect was raised. This is the opposite direction to plain algebraic effects.

Effekt (Brachthäuser et al., 2020; *Effekt Language: Home*) is a language with algebraic effects. Effekt has practical features such as records, modules, mutable variables, and inputs/outputs. It also has a well-working compiler and runtime system. Moreover, Effekt has simple and easily extensible type system due to special treatment of effect polymorphism unlike Frank (Lindley et al., 2017) and Koka (Leijen, 2017). Effect polymorphism is required to support higher-order functions in languages with algebraic effects. Effekt realizes this without parametric effect polymorphism, which is difficult to understand for users.

Contributions. Our contributions are (i) extending the Effekt compiler with bidirectional effects, (ii) formalize the extended type system, and (iii) showing that this extension allows implementation of generators and communication, which was not possible in the original Effekt language. We proceed as follows:

- Chapter 2 explains algebraic effects, Effekt language, and bidirectional algebraic effects.

- Chapter 3 shows examples that it is difficult to express with plain algebraic effects, and gives how to solve this problem with bidirectional algebraic effects.
- Chapter 4 gives details related to the extension of the compiler and runtime system.
- Chapter 5 shows the formalize and proves the type soundness.
- Chapter 6 shows Effekt programs with bidirectional effects.
- Chapter 7 discusses related work.
- Chapter 8 concludes this paper.

Chapter 2

Background

2.1 Algebraic Effects

Algebraic effects and handlers (Pretnar, 2015; Plotkin and Pretnar, 2013) are an approach to expressing computational effects such as `put/get` for mutable state, `read/print` for input and output, and `throw` for exceptions. These are called *operations* and users define interpretation of them by *handlers* that are like exceptions and handlers of Java.

Figure 2.1 shows an Effekt (explained in 2.2) program that involves exception handling Exceptions are the simplest example with algebraic effects. Operation `DivByZero` is defined (line 1). The function `div` takes two integer and calculate x/y. In an ordinary programming language, dividing an integer by 0 produces an runtime error (such as `ArithmeticException`). In Figure 2.1, `div(x, 0)` raises `DivByZero` that represents "integer x is divided by 0". The operation call `do DivByZero()` in `div` (line 5) raises an effect `DivByZero` with the same name as its operation, if $y = 0$. A raised operation propagates upwards in the dynamic call stack that is similar to exceptions in Java, and must be captured by a handler. In this example, `DivByZero` is handled by the handler in `main` function (line 16-18). The function `div` is passed 6 and 0 (line 14), so `div` raises an effect `DivByZero` and it is handled. The function call `main()` prints "`div by zero!`".

```
1  effect DivByZero() : Unit
2
3  def div(x: Int, y: Int) : Int / { DivByZero } = {
4    if (y == 0) {
5      do DivByZero()
6    }
7    else {
8      x / y
9    }}
10
11 def main() = {
12   try {
13     println(div(6, 0))
14   }
15   with DivByZero {
16     println("div by zero!")
17   }}
```

FIGURE 2.1: An example of algebraic effects (exception)

Function `div` has the type `(Int, Int) → Int / { DivByZero }`. This type means "Given two values of type `Int`, the function produces a value of type `Int` and has an effect `DivByZero`". Traditionally with algebraic effects, effectful functions such as `div` have a type like $\alpha \to \beta/\varepsilon$. In traditional language with algebraic effects such as Koka, this type is read as follows (Brachthäuser et al., 2020).

> "Given a value of type α, the function produces a value of type β and has effects ε."

In the above sentence, "has effects ε" with non-empty effects ε means that the function may raise an effect including ε when it is called. In particular, a function with empty ε is considered *pure*.

A more complex application is input/output. The following example (Figure 2.2) defines operations `Read` and `Print`. Operation `Read` inputs a string like `readLine()` in Java. Operation `Print` outputs a string like `println` in Java. The difference from Figure 2.1 is that the `resume` function is called. This `resume` function represents remaining computation from the operation call of the handled effects. Passing a parameter value v to function `resume` corresponds to regarding the result of an operation as a value v. In Figure 2.2, a `Read` effect is raised (line 5) and handled in function `main` (line 13-15). The `resume` function in the handler of `Read` is called with a parameter `"Kazuki"` and the result value of operation call `do Read()` is `"Kazuki"`. On the other hand, operation `Print` takes a string, which is passed to its handler as `str`. In the handler of `Print` (line 16-19), the passed parameter `str` is printed by `println`. The `resume` function in the handler of `Print` is called with a unit value `()` because we want to treat `Print` operation as a function. The function call `main()` prints "`Hello Kazuki!`".

```
1  effect Read() : String
2  effect Print(str: String) : Unit
3
4  def sayHello() : Unit / { Read, Print } = {
5    val name = do Read()
6    do Print("Hello " ++ name ++ "!")}
7
8  def main() = {
9    try {
10     sayHello()
11   }
12   with Read {
13     resume("Kazuki")
14   }
15   with Print { str =>
16     println(str)
17     resume(())
18   }}
```

FIGURE 2.2: An example of algebraic effects (Read/Print)

There are two advantages of supporting algebraic effects over having individual effects as primitives such as exceptions and input/output. First, users can write highly modular programs. Programmers can separate interfaces and implementations by effects and handlers. Implementations provided by handlers of interfaces as effects can be replaced easily. In Figure 2.2, handlers of `Read` and `Print` can be replaced with other handlers such as communicating or discarding the parameter of `Print`. This can be applied to dependency injection (Fowler, 2004). Second, programs with algebraic effects are strictly typed. This means effect safety: all effects are handled by correct handlers. An effect can never be accidentally handled by the wrong handler or not handled by any handlers.

2.2 Effekt Language

Effekt (Brachthäuser et al., 2020; *Effekt Language: Home*) is a language with algebraic effects. Its syntax is similar to Scala's one. Effekt is fully-fledged language with following reasons. First, it has features such as record, module, mutable variable, and input/output. Second, it has well working compiler and runtime system.

Effekt takes a unique approach to effect polymorphism Effect polymorphism is required to support higher-order functions in languages with algebraic effects, because a passed function may perform arbitrary effects, and we must be explicit about those effects in the type of the function. Figure 2.3 defines a higher-order function `myMap` (line 5-10) that takes a list and a block (anonymous function in Effekt) and calls the block on each element of the

list, and collects these result. In the `main` function, `myMap` is called with a list and a block that includes operation call `Print(a)` (line 14). The type of the block (called X) at line 14 is:

```
String → Unit / { Print }
```

The type of the block (called X_f) of `myMap` is:

```
String → Unit / {}
```

X and X_f are different in effects ε, so Figure 2.3 has a contradiction. Hence we have to add `Print` to ε of X_f to solve the contradiction. However it is not possible to specialize signatures of higher-order functions to any use-sites without effect polymorphism.

Therefore Koka supports parametric effect polymorphism.

```
String → Unit / {e}
```

Here, `e` can be replaced with any effect(s). This feature is difficult to understand, as Biernacki et al., 2019 refers. Some languages including Koka offer syntactic sugar to hide effect polymorphism to users. However, this hiding often breaks and details of effect polymorphism to users.

Effekt realizes effect polymorphism simply by giving the different interpretation, which is called *contextual effect polymorphism*, for a type $\alpha \to \beta / \varepsilon$:

> "Given a value of type α, the function produces a value of type β and *requires the calling context to handle effects* ε"
>
> (Brachthäuser et al., 2020)

In Effekt, effects are considered as inputs to a function. Function `sayHello` in Figure 2.2 takes hidden parameter `Read` and `Print`. These are revealed in System Ξ, which is a core language translated from Effekt. Figure 2.5 is a translated System Ξ program from Figure 2.2. Function `sayHello` has two effects as parameters used in its body. An effect type represent *capability* that a computation needs from its context. This makes Effekt more lightweight and simpler than other languages with algebraic effects.

To support both contextual effect polymorphism and effect safety, Effekt offers blocks as second-class (Osvald et al., 2016). Second-class values cannot be returned and stored in data structures. If blocks are first-class, type system cannot track all effects which are raised in a block. The following example illustrates this problem.

```
def returnit() { f: Unit } : Unit / {} = f
val g = returnit { do Exception() }
g() // raises Exception
```

Here, we define a function `returnit` which returns the block as a type `Unit → Unit / {}`. Then we bind the result of `returnit` with a block raising `Exception` to a variable `g`. The type of `g` is `Unit → Unit / {}`, hence a function call `g()` raises no effects in the type system. However `g()` raises an effect `Exception` in fact and a runtime error is occurred. To prevent this problem,

Effekt treats blocks as second-class values. Effekt gets an advantage of begin lightweight while providing type safety and effect polymorphism in place of the limitation of expressiveness.

```
1  import immutable/list
2
3  effect Print(str: String): Unit
4
5  def myMap[A, B](l: List[A]) { f: A => B } : List[B] =
6    l match {
7      case Nil() => Nil[B]()
8      case Cons(a, rest) =>
9        Cons(f(a), myMap(rest) { a => f(a) })
10   }
11
12 def main() = {
13   try {
14     myMap(["A", "B", "C"]) { a => Print(a) };
15     ()
16   }
17   with Print { str =>
18     println(str)
19     resume(())
20   }}
```

FIGURE 2.3: Operation call in a block

```
A
B
C
```

FIGURE 2.4: The result of Figure 2.3

```
1  import effekt
2
3  record Read(Read)
4  record Print(Print)
5
6  def sayHello = (Read, Print) => {
7    val name = Read.Read();
8    Print.Print(infixConcat(
9      infixConcat("Hello ", name), "!"))
10 }
11
12 def main = () => {
13   handle {
14       (Read, Print) => {
15           sayHello(Read, Print)
16       }
17   } with ([{
18       Read: (resume) => {
19           resume("Kazuki")
20       }
21   }, {
22       Print: (str, resume) => {
23           println(str);
24           resume(())
25       }
26   }])
27 }
```

FIGURE 2.5: A translated System Ξ program from Figure 2.2

2.3 Bidirectional Effects

In Effekt, many kinds of programs can be expressed by algebraic effects. However it is hard to write programs with complex control-flow. For example, generator, async-await, and communication. Modern software uses event-driven programming frequently. Callback functions are a conventional pattern for event-driven, but callbacks without constraint make software complex. Features such as generator and async-await is a solution of this problem because these are useful for writing structured event-driven programs (Zhang et al., 2020). These features involve *bidirectional* control flow. In languages with algebraic effects, it is impossible to implement these *bidirectional* program with keeping effect safety.

Bidirectional algebraic effects (Zhang et al., 2020) are a novel concept that makes it possible to implement bidirectional control flow in a simple way. The difference between plain algebraic effects and bidirectional algebraic effects is in the propagating direction of effect that is raised in the handler. With plain algebraic effects, an effect propagates up the dynamic call stack

to its handler. With bidirectional algebraic effects, an effect propagates to the location where the initiating effect was raised. This is the opposite direction to plain algebraic effects.

Bidirectional effects behaves as Figure 2.6. There are two differences between plain and bidirectional algebraic effects. First, operation calls may raise effects that are listed in its operation definition. Effect `Replace` is listed in the definition of effect `Yield` (line 1). An operation call `do Yield()` may raise a effect `Replace`. Second, function `resume` takes a block parameter. This block is evaluated in the location where the initiating effect was raised. If an effect is raised while this block is executed, the operation call of initiating effect raises the effect that are defined in its operation definition. A `Yield` effect is raised in function `iter` (line 6). This effect is handled in the handler in function `func` (line 13-17). In the `Yield` handler, a `Replace` effect is raised inside the block parameter of `resume` (line 14). So the `Yield` effect raised in this block is handled in function `iter` (line 7-8). The function call `main()` prints "`Replace handler at iter`".

Let us write this expression with plain algebraic effect. This expression raising the `Replace` effect (line 14) is written as line 16. A raised effect propagates up the call stack, so it is handled the handler in function `main` (line 22-23). The function call `main()` prints "`Replace handler at main`" with plain algebraic effects.

```
1  effect Yield(): Unit / { Replace }
2  effect Replace(x: Int): Unit
3
4  def iter() = {
5    try {
6      do Yield() }
7    with Replace { x =>
8      println("Replace handler at iter") }}
9
10 def func() = {
11   try {
12     iter() }
13   with Yield {
14     resume { do Replace(42) }
15     // With plain algebraic effects
16     // do Replace(42); resume(())
17   }}
18
19 def main() = {
20   try {
21     func() }
22   with Replace { x =>
23     println("Replace handler at main") }}
```

FIGURE 2.6: An example of bidirectional effects (Yield and Replace)

As the next examples, how to implement generator and communication program with bidirectional effects can be described in the following way. Figure 2.7 is a generator program taken from Zhang et al. In this program, function `iter` takes a integer list and returns a modified list. It `Yields` each elements of the given list. List modification performed by effects `Replace` and `Reject` in the handler of effect `Yield`. In the block of the parameter of function `resume` in this handler, raising `Replace` means *replacing the element with* x and raising `Reject` means *removing the element*. These effects, listed at the definition of `Yield` (line 3), are raised at the location of operation call of `Yield` (line 11). Therefore these are handled by `iter` (line 10-15). In function `main`, function `iter` is called with a list `lst` (line 20). Function `iter` raises an effect `Yield` (line 11) and handled at line21-24. In the handler of `Yield`, if `x < 0`, effect `Reject` is raised; otherwise effect `Replace` is raised with `x * 2` (line 23). These effects are handled in function `iter` and handlers override or drop elements (line 14-15). The result of `lst` is showed at the bottom of Figure 2.7.

The advantage of bidirectional effects over plain algebraic effects in this example is type-safety. Figure 2.8 is an implementation of Figure 2.7 with plain algebraic effects. Effects `Replace` and `Reject` are defined as a data type `Modification` (line 3-6). A value of `Modification` is passed from the handler of `Yield` to the operation call of `Yield` (line 13 and 27). Even if a programmer forgets to handle the result of `do Yield(x)` (line 16-19), the program is passed compiling and it works. Because this violates the type (effect) safety, this is not intended behavior. We want a type error that a modification (replace or reject) is not handled. Using bidirectional effects, if a programmer forgets to handle effects from `do Yield(x)` (line 14-15), the compiling is failed.

```
1  effect Replace(x: Int): Unit
2  effect Reject(): Unit
3  effect Yield(x: Int): Unit / { Replace, Reject }
4
5  def iter(lst: List[Int]): List[Int] / { Yield } =
6    lst match {
7      case Nil() => Nil()
8      case Cons(x, xs) =>
9        val xsp = iter(xs)
10       try {
11         do Yield(x)
12         Cons(x, xsp)
13       }
14       with Replace { xp => Cons(xp, xsp) }
15       with Reject { xsp }
16   }
17
18 def main() = {
19   val lst = [0, 1, 3, -2, -8, 9]
20   val lst2 = try { iter(lst) }
21   with Yield { x =>
22     resume {
23       if (x < 0) do Reject() else do Replace(x * 2) }
24   }
25   println(lst2)
26 }
```

```
Cons(0, Cons(2, Cons(6, Cons(18, Nil()))))
```

FIGURE 2.7: An example of generator and the result of it

```
1  effect Yield(x: Int): Modification
2
3  type Modification {
4    Replace(x: Int);
5    Reject()
6  }
7
8  def iter(lst: List[Int]): List[Int] / { Yield } =
9    lst match {
10     case Nil() => Nil()
11     case Cons(x, xs) =>
12       val xsp = iter(xs)
13       val mod = do Yield(x)
14
15       // Mistake!
16       // mod match {
17       //  case Replace(x) => Cons(x, xsp)
18       //  case Reject => xsp
19       //}
20       Cons(x, xsp)
21   }
22
23 def main() = {
24   val lst = [0, 1, 3, -2, -8, 9]
25   val lst2 = try { iter(lst) }
26   with Yield { x =>
27     resume( if (x < 0) Reject() else Replace(x * 2) )
28   }
29   println(lst2)
30 }
```

FIGURE 2.8: An implementation of Figure 2.7 with plain algebraic effects

An yet another example, consider a communication program Figure 2.9 taken from Zhang et al. This program denotes the exchange effects between function `pinger` and `ponger`. Function `pinger` raises an effect `Ping` that is handled by function `ponger` and function `ponger` raises an effect `Pong` that is handled by function `pinger`. Figure 2.10 shows how pong-pong program to works. In this figure, a box denotes the function context in the call stack. A line denotes the function call. A dotted box denotes the handler and a dotted line denotes the flow of raised effect. Function `main` calls `pinger` with `0` and `5`. This means that functions `pinger` and `ponger` repeat to exchange effects 5 times. Function `pinger` raises an effect `Ping` (line 8) and it is handled by the handler in function `main` (line 30-33). In the block of the parameter of `resume` in it, function `ponger` is called (line 32). This function call is evaluated at `do Ping()` (line 8), so the context of `ponger` is above the context of `pinger` in the call stack. Function `ponger` raises an effect `Pong` (line 19) and it is handled by

the handler if function `pinger` (line 11-14). In this handler, function `pinger` is called with `i + 1` and `N`, that increments the counter `i`. The continuation is the same as before untile `i = 0`.

This program cannot be implemented using plain algebraic effects, because the handler would send an other effect to the location where initiating effect was raised. More practical examples are explained in Chapter 6.

```
1  effect Ping(): Unit / { Pong }
2  effect Pong(): Unit / { Ping }
3
4  def pinger(i: Int, N: Int): Unit / {Ping, Console} = {
5    println("enter pinger")
6    println(i)
7    try {
8      if (i < N) do Ping()
9      else ()
10   }
11   with Pong {
12     println("enter Pong handler")
13     resume { pinger(i + 1, N) }
14   }}
15
16 def ponger(): Unit / {Pong, Console} = {
17   println("enter ponger")
18   try {
19     do Pong()
20   }
21   with Ping {
22     println("enter Ping handler")
23     resume { ponger() }
24   }}
25
26 def main() = {
27   try {
28     pinger(0, 10)
29   }
30   with Ping {
31     println("enter Ping handler@main")
32     resume { ponger() }
33   }}
```

```
enter pinger
0
enter Ping handler@main
enter ponger
enter Pong handler
enter pinger
1
enter Ping handler
enter ponger
enter Pong handler
enter pinger
2
enter Ping handler
enter ponger
enter Pong handler
```

```
enter pinger
3
enter Ping handler
enter ponger
enter Pong handler
enter pinger
4
enter Ping handler
enter ponger
enter Pong handler
enter pinger
5
```

FIGURE 2.9: An example of communication (ping and pong) and the result of it

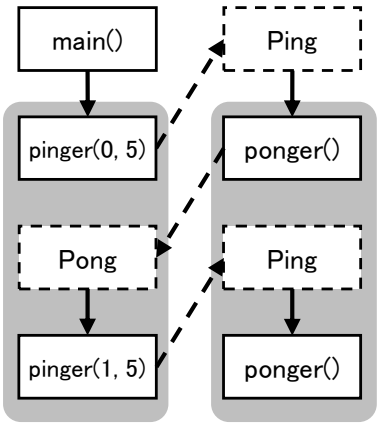

FIGURE 2.10: Ping pong

Chapter 3

Problems and Proposal

As we mentioned, there is a problem that we cannot express bidirectional control flow in languages with plain algebraic effects, such as Effekt and Koka. This prevents us from implementing generator, async-await, and communication. These are particularly useful in real-world software, which is often event-driven.

To solve this problem, we introduce bidirectional algebraic effects to Effekt language.

We chose Effekt language for the following reasons. First, Effekt is practical because it has many practical features, including records, and it has a well-working compiler and runtime system. Second, Effekt takes a unique approach to effect polymorphism, and we wonder the synergy that is occurred when we mix this approach and bidirectional effects.

There are two major tasks to introduce bidirectional algebraic effects to Effekt.

1. Adding effects to signature of operations.
   Example: `effect Yield(x: Int): Unit / {Replace, Reject}`

2. Allowing `resume` to be passed a computation.
   Example: `resume { do Fail() }`

In Chapter 4, we describes details of compiler and runtime system implementation to work programs with bidirectional effects well. Then in Chapter 5, we describes details of formalization to ensure the type safety.

Chapter 4

Implementation

In this chapter, we first explain the structure of the compiler and runtime system of Effekt. Then we explain how to implement bidirectional algebraic effects.

4.1 Effekt Compiler

The compiler of Effekt [1] consists of multiple modules (`Phase` in the code of Effekt compiler). The module `Parser` analyzes a given program and generates an AST (Abstract Syntax Tree). The module `Namer` collects the definitions of identifiers such as effects, variables, and parameters. It generates a mapping from source identifiers to symbol which is a structure with a name and a type. For example, `Yield` $\rightarrow$ `EffectOp(Yield, [], [x: Int], Unit, [Replace, Reject])` represents an operation `Yield` has a parameter `x` with `Int`, returns a value of `Unit`, and may raise effects `Replace` and `Reject`. The module `Typer` infers the type and effect of expressions and outputs the type of blocks, functions, and resumptions. The module `Transformer` transforms an AST of Effekt into an AST of System Ξ. Finally, the AST is translated to program of the target language such as JavaScript and Chez Scheme (*(chez (chez scheme))*).

We modified `Parser` to accept the extended syntax, `Namer` to register an effectful operation with the context, and `Typer` to resolve the type of effectful operations.

4.2 Modification of the Extension

Extending Effekt with bidirectional effects needs four modifications. First, we modify the declaration and grammar of operations for the parser to accept an effectful type (i.e. $\alpha \rightarrow \beta/\varepsilon$) instead of a plain type (i.e. $\alpha \rightarrow \beta$). The changes are shown in following figure. We express added lines and deleted lines by using "+" and "-".

```
// /shared/src/main/scala/effekt/source/Tree.scala
case class Operation(id: IdDef, tparams: List[Id],
-   params: List[ValueParams], ret: ValueType) extends Definition {
+   params: List[ValueParams], ret: Effectful) extends Definition {
    type symbol = symbols.EffectOp
```

[1] https://github.com/effekt-lang/effekt

```
}
```

Note that classes including `Operation` in `Tree.scala` represent the AST of Effekt. In `Tree.scala`, we have changed the return type `ret` of operations from `ValueType` to `Effectful`, which is a value type with an effect.

```
// /shared/src/main/scala/effekt/symbols/symbols.scala
case class EffectOp(name: Name, tparams: List[TypeVar],
     params: List[List[ValueParam]], ret: Option[Effectful],
     effect: UserEffect) extends Fun
+  {
+    def otherEffects: Effects = ret.get.effects - effect
+    def isBidirectional: Boolean = otherEffects.nonEmpty
+  }
```

Note that classes including `EffectOp` in `symbols.scala` represent the AST of System Ξ. In `symbols.scala`, we add two methods. The method `otherEffects` returns the additional effects that the operation may raise. The method `isBidirectional` serves as a flag telling us whether the operation is bidirectional.

We modify `Parser` along with the modification of AST.

```
// /shared/src/main/scala/effekt/Parser.scala
   lazy val effectOp: P[Operation] =
-    idDef ~ maybeTypeParams ~ some(valueParams) ~/ (‘:‘ ~/> valueType)
+    idDef ~ maybeTypeParams ~ some(valueParams) ~/ (‘:‘ ~/> effectful)
      ^^ Operation
```

We also modify the parser so that `resume` accepts a block.

```
// /shared/src/main/scala/effekt/Parser.scala
   lazy val resumeExpr: P[Expr] =
-    (‘resume‘ ^^^ IdRef("resume")) ~ valueArgs
+    (‘resume‘ ^^^ IdRef("resume")) ~ args
      ^^ { case r ~ args => Call(r, Nil, List(args)) withPositionOf r }
```

Here, `args` means values and blocks.

As a second step, we modify `Namer` to register an operation with effectful type with the context.

```
// /shared/src/main/scala/effekt/Namer.scala
    val name = Context.freshTermName(id)
    Context scoped {
      val tps = tparams map resolve
-     val tpe = Effectful(resolve(ret), Effects(List(effectSym)))
-     val op = EffectOp(Name(id), tps, params map resolve, Some(tpe),
-       effectSym)
+     val Effectful(tpe, otherEffs) = resolve(ret)
+     val retResolved = Effectful(tpe, otherEffs + effectSym)
+     val op = EffectOp(Name(id), tps, params map resolve,
+       Some(retResolved), effectSym)
      Context.define(id, op)
      op
    }
```

In `Namer.scala`, the modification is to accept effectful types as `otherEffs`.

Thirdly, we modify `Typer` to determine the appropriate type of `resume`. If an operation `Eff` has an effectful type `a → b / { Eff, E1, E2, ... En }`, the resume function in the handler of the operation has type:

```
(() → b / { E1, E2, ..., En }) → t
```

Here, `t` is the result type of try-with expression corresponding to `resume`. This violates the specification that block values are second-class values in Effekt. However, it is not a problem, because the block value is immediately used in a function call of the form *`Eff()`*`(E1, E2, ..., En)` (we use an italic font for the block). That is, the block value cannot be used as a first-class value. When an operation does not have an effectful type, the type of `resume` is `b → t`, which is the same as the original Effekt.

```
// /shared/src/main/scala/effekt/Typer.scala
   case d @ source.OpClause(op, params, body, resume) =>
     val effectOp = d.definition
-    val bt = Context.blockTypeOf(effectOp)
-    val ps = checkAgainstDeclaration(op.name, bt.params, params)
-    val resumeType = BlockType(Nil, List(List(effectOp.ret.get.tpe)),
-      ret / Pure)
+
+    val BlockType(_, pms, tpe / effs) = Context.blockTypeOf(effectOp)
+    val ps = checkAgainstDeclaration(op.name, pms, params)
+
+    val resumeType = if (effectOp.isBidirectional) {
+      // resume { e }
+      BlockType(Nil, List(List(BlockType(Nil, List(Nil),
+        tpe / effectOp.otherEffects))), ret / Pure)
+    } else {
+      // resume(v)
+      BlockType(Nil, List(List(tpe)), ret / Pure)
+    }
```

In `Typer.scala`, we modify the typer of handlers named `OpClause` to infer `resumeType` (the type of `resume`) whether a effect operation `effectOp` is bidirectional, that is, whether the operation has effects. If the operation is bidirectional, the typer infers that the type of `resume` is `(() → tpe / effectOp.otherEffects) → ret`. Otherwise, the typer infers that the type is `tpe → ret` as before.

Finally, we modify the runtime system of Effekt to work correctly with effectful operations. An effectful operation call `Eff()` is evaluated as `Eff()(E1, E2, ..., En)` by modifying the implementation of operation in runtime system. By this modification, a block that is passed to `resume` can be evaluated in the location of the initiating operation call. Needless to say, if effects are raised in the block, these propagate from the location of the initiating operation call upwards in the dynamic call stack in the same manner as plain algebraic effects.

```
// /lib/effekt_runtime.js
   for (var op in h) {
     const impl = h[op];
     cap[op] = function() {
       const args = Array.from(arguments);
-      return shift(p)(k => impl.apply(null, args.concat([k])))
+      const arity = impl.length - 1
+      const oargs = args.slice(0, arity)
+      const caps = args.slice(arity)
+      var r = shift(p)(k => impl.apply(null, oargs.concat([k])))
+      // resume { caps => e }
+      if (caps.length > 0) {
+        return r.then(f => f.apply(null, caps))
+      }
+      // resume(v)
+      else {
+        return r
+      }
     }
   }
```

Here, the modification is to run both form `resume { e }` and `resume(v)`.

We leave `resume()` for backward compatibility, that is, we can still pass values to `resume`. This does not require a separate typing rule because we can express `resume(x)` as `val tmp = x; resume { tmp }`.

Chapter 5

Formalization

In this chapter, we give a formalization of Effekt with bidirectional effects.

In the following formalization, we highlight changes from the original language in gray. Note that we give only changed definitions and proofs.

5.1 Extended Effekt

Figure 5.1 gives the specification of extended Effekt. At the level of syntax, a key addition is ε in the effect signature, which represents the set of effects to be raised by an operation. We modify effect environment accordingly. We also consider this effect in typing rules related to the operation F. Specially, because an operation call `do F(e)` may raise effects, we add the union of ε to the conclusion of EFFECTCALL. Moreover, we modify `resume` to take a block which results τ_0 and raises ε.

Syntax:

Statements $s ::= \ldots$

$\quad$ **effect** $F(x : \tau) : \tau \,/\, \varepsilon \,;\, s$ effect declaration

Syntax of Types:

Effect Environment $\Sigma ::= \emptyset \mid \Sigma, F : \tau \to \tau \,/\, \varepsilon$

Typing rules:

$$\frac{\Gamma|\Delta|\Sigma, F : \tau_1 \to \tau_0 \,/\, \varepsilon_1 \vdash s_2 : \tau_2|\varepsilon_2}{\Gamma|\Delta|\Sigma \vdash \textbf{effect}\ F(x_1 : \tau_1) : \tau_0 \,/\, \varepsilon_1 \,; s_2 : \tau_2|\varepsilon_2}\ [\textsc{Effect}]$$

$$\frac{\Sigma(F) = \tau_1 \to \tau_0 \,/\, \varepsilon \qquad \Gamma \vdash e_1 : \tau_1}{\Gamma|\Delta|\Sigma \vdash \textbf{do}\ F(e_1) : \tau_0|\{F\} \cup \varepsilon}\ [\textsc{EffectCall}]$$

$$\frac{\Sigma(F) = \tau_1 \to \tau_0 \,/\, \varepsilon_1 \qquad \Gamma|\Delta|\Sigma \vdash s : \tau|\varepsilon \qquad \Gamma, x_1 : \tau_1|\Delta, \mathsf{resume} : (() \to \tau_0/\varepsilon_1) \to \tau/\phi|\Sigma \vdash s' : \tau|\varepsilon_0}{\Gamma|\Delta|\Sigma \vdash \textbf{try}\ \{s\}\ \textbf{with}\ F\ \{(x_1 : \tau_1) \Rightarrow s'\} : \tau|(\varepsilon \backslash \{F\}) \cup \varepsilon_0}\ [\textsc{Try}]$$

FIGURE 5.1: An extension of Effekt (Brachthäuser et al., 2020)

5.2 Extended System Ξ

Figure 5.2 defines the syntax of extended System Ξ. We add union types v which is used in the proofs to deal with both value types τ and block types σ. We also add a new member c to the block environment Δ to memorize *capabilities*. *Capabilities* are implementation of operations, namely handlers.

In the typing rules, we use block types σ instead of value types τ for the parameter of k (`resume` in Effekt) and the result of operation (**cap** and F). This is because we let the block passed to `resume` to be evaluated in the location of an operation call. The passed block is evaluated immediately by the translation from operation call `do F(e)` to `F(e)(F1, ..., Fn)` as we mentioned in Chapter 4. This is reflected in the translation rule of operation call **do** $F(e_1)$. We add two new typing rules CAPCALL and CAPVAR for distinction between blocks and capabilities. The difference of these is the result type, that is, value types τ of blocks and block types σ of capabilities.

In the reduction rules, we make a change to the (cap) rule. Specifically, we changed the argument y of resumption k to block value f for which `resume` takes a block parameter.

Syntax of Types:

$$\text{Union Types} \quad v ::= \tau \mid \sigma$$

$$\text{Block Environment} \quad \Delta ::= \emptyset \mid \Delta, f : \tau \mid \Delta, c : \tau \to \sigma$$

Typing rules:

$$\frac{\Xi = \Xi_1, l : \tau, \Xi_2 \qquad \Gamma, x_1 : \tau_1 | \Delta, k : \sigma_0 \to \tau | \Xi_1 \vdash s : \tau}{\Gamma | \Delta | \Xi \vdash \mathbf{cap}_l \{(x_1 : \tau_1, k : \sigma_0 \to \tau) \Rightarrow s\} : \tau_1 \to \sigma_0} \text{[CAP]}$$

$$\frac{\text{c is a cap} \qquad \Gamma | \Delta | \Xi \vdash c : \tau \to \sigma_0 \qquad \Gamma | \Delta | \Xi \vdash e : \tau}{\Gamma | \Delta | \Xi \vdash c(e) : \sigma_0} \text{[CAPCALL(new rule)]}$$

$$\frac{\begin{array}{c}\Gamma \mid \Delta, F : \tau_1 \to \sigma_0 \mid \Xi \vdash s : \tau \\ \Gamma, x : \tau_1 \mid \Delta, k : \sigma_0 \to \tau \mid \Xi \vdash s' : \tau\end{array}}{\Gamma \mid \Delta \mid \Xi \vdash \mathbf{handle}\ \{F \Rightarrow s\}\ \mathbf{with}\ \{(x, k) \Rightarrow s'\} : \tau} \text{[HANDLE]}$$

$$\frac{\Delta(c) = \tau \to \sigma}{\Gamma \mid \Delta \mid \Xi \vdash c : \tau \to \sigma} \text{[CAPVAR(new rule)]}$$

Reduction Rules:

$$(cap) \quad \#_l \cdot H_l \cdot (\mathbf{cap}_l\{(x, k) \Rightarrow s\})(v) \to s[x \mapsto v, k \mapsto \{f \Rightarrow \#_l \cdot H_l \cdot f\}]$$

where f is a block

Translation Effect to System Ξ:

$$\mathcal{S}[\mathbf{effect}\ F(x_1 : \tau_1) : \tau_0 / \varepsilon_1 ; s] = \mathcal{S}[s]$$

$$\mathcal{S}[\mathbf{do}\ F(e_1)] = F(e_1)\ (F_1, ..., F_n) \quad \text{where } F : \tau_1 \to \tau_0 / \{F_1, ..., F_n\}$$

FIGURE 5.2: An extension of SystemΞ (Brachthäuser et al., 2020)

5.3 Soundness Proofs of Extended Effekt and System Ξ

In this section, we try to give the soundness of extended System Ξ translated from extended Effekt. One goal of this soundness is progress and preservation. Note that we use the same numbering of theorems and lemmas as Brachthäuser et al., 2020.

> THEOREM 4.2 (PROGRESS OF SYSTEM Ξ) If $\emptyset \mid \emptyset \mid \emptyset \vdash s : \tau$, then s is a value v or $s \mapsto s'$.
>
> THEOREM 4.3 (PRESERVATION OF SYSTEM Ξ) If $\emptyset \mid \emptyset \mid \emptyset \vdash s : \tau$ and $s \mapsto s'$, then $\emptyset \mid \emptyset \mid \emptyset \vdash s' : \tau$.

To give this proof, some lemmas and theorems are given. However Lemma A.5$_{cap}$ have a problem and this problem prevents the proof of Theorem A.6. We explain this with proofs. Remaining lemma and theorems can be found in Appendix A.

5.3.1 Lemma A.5$_{cap}$

> (CONTEXT PLUGGING FOR CAPABILITIES) Given $E : \sigma \to v$ and $\lceil E \rceil = \Xi'$, if $\Gamma \mid \Delta \mid \Xi, \Xi' \vdash s : \sigma$ where $s = (\mathbf{cap}_l\{(x,k) \Rightarrow s''\})(v)$, then $\Gamma \mid \Delta \mid \Xi \vdash E[s] : v$.

Case $E = \square$**:** Consider $E = \sigma \to \sigma$ and $\Xi' = \emptyset$, if $\Gamma \mid \Delta \mid \Xi, \emptyset \vdash s : \sigma$, then obviously $\Gamma \mid \Delta \mid \Xi \vdash E[s] : \sigma$.

Case $E = \mathbf{val}\ x = E'; s'$**:** From $\lceil E' \rceil = \lceil E \rceil = \Xi'$, $E' : \sigma \to v''$, and the induction hypothesis, we obtain:

$$\Gamma \mid \Delta \mid \Xi \vdash E'[s] : v'' \tag{5.1}$$

$$\Gamma, x : v'' \mid \Delta \mid \Xi \vdash s' : \tau \tag{5.2}$$

We apply (VAL) to 5.1 and 5.2 and obtain:

$$\Gamma, \mid \Delta \mid \Xi \vdash E[s] : \tau \tag{5.3}$$

Case $E = \#_l\{E'\}$ From the premise and $\Xi' = \lceil E \rceil = l, \lceil E' \rceil$, we obtain:

$$\Gamma \mid \Delta \mid \Xi, l : v, \Xi'' \vdash s : v \tag{5.4}$$

where $\Xi'' = \lceil E' \rceil$. From $E' : \sigma \to v$ and 5.4, we obtain:

$$\Gamma \mid \Delta \mid \Xi, l : v \vdash E'[s] : v \tag{5.5}$$

We apply [DELIMIT] to 5.5 and obtain:

$$\Gamma \mid \Delta \mid \Xi \vdash E[s] : v \tag{5.6}$$

□

The problem of Lemma A.5$_{cap}$ is the type of $E[s]$ is v (τ or σ) in 5.6. This prevents the proof of Theorem A.6.

5.3.2 Theorem A.6

(PRESERVATION IN CONTEXT) $\Gamma \mid \Delta \mid \emptyset \vdash E[s] : \tau$, $E : v' \to \tau$, and $s \to s'$, then $\Gamma \mid \Delta \mid \emptyset \vdash E[s'] : \tau$

Case (CAP) We have $E : \sigma' \to \tau$ and $\#_l \cdot H_l \cdot (\mathbf{cap}_l\{(x,k) \Rightarrow s''\})(v) \to s''[x \mapsto v, k \mapsto \{f \Rightarrow \#_l \cdot H_l \cdot f\}]$.
We use Lemma A.4$_{cap}$ to obtain:

$$\Gamma \mid \Delta \mid \Xi, \Xi' \vdash (\mathbf{cap}_l\{(x,k) \Rightarrow s''\})(v) : \sigma'' \quad \textit{with } \lceil \#_l\{H_l\} \rceil = \Xi' \tag{5.7}$$

From rule [CAP-CALL], we have:

$$\Gamma \mid \Delta \mid \Xi, \Xi' \vdash \mathbf{cap}_l\{(x,k) \Rightarrow s''\} : \tau_1 \to \sigma'' \tag{5.8}$$

$$\Gamma \mid \Delta \mid \Xi, \Xi' \vdash v : \tau_1 \tag{5.9}$$

From rule [CAP], we have:

$$\Gamma, x : \tau_1 \mid \Delta, k : \sigma'' \to \tau' \mid \Xi \vdash s'' : \tau' \tag{5.10}$$

We show:

$$\Gamma \mid \Delta \mid \Xi \vdash s''[x \mapsto v, k \mapsto \{f \Rightarrow \#_l \cdot H_1[f]\}] : \tau' \tag{5.11}$$

using Lemma A1, A2, and following derivations:

$$\Gamma \vdash v : \tau_1 \tag{5.12}$$

$$\Gamma \mid \Delta \mid \Xi \vdash \{f \Rightarrow \#_l \cdot H_1[f]\} : \sigma'' \to \tau' \tag{5.13}$$

5.12 equals to 5.9. To derive 5.13, we show:

$$\Gamma \mid \Delta, f : \sigma'' \mid \Xi \vdash \#_l \cdot H_1[f] : \tau' \tag{5.14}$$

We apply (BLOCK) to 5.14 and get 5.13.

We apply block context weakening to 5.7 and $\Gamma \mid \Delta \mid \Xi' \vdash f : \sigma''$, replace $(\mathbf{cap}_l\{(x,k) \Rightarrow s''\})(v)$ with f, and get:

$$\Gamma \mid \Delta, f : \sigma'' \mid \Xi, \Xi' \vdash f : \sigma'' \tag{5.15}$$

Finally, we apply Lemma A.5$_{cap}$ to 5.15 and get:

$$\Gamma \mid \Delta, f : \sigma'' \mid \Xi \vdash \#_l \cdot H_l[f] : v'' \tag{5.16}$$

To use 5.16, we must replace τ' with v'' in the proof. We must modify the result type of rule (BLOCK). This leads that the result type of all derivations becomes v, and it means that a block is treated as a first-class value. We cannot change block type and we take an another approach to avoid this disruption in the future.

Chapter 6

Examples

In this chapter, we present several examples to show the practical use of bidirectional effects. These programs are fully under the type-safe and effect-safe, that is, users can know their mistake at compile time. The type and effect safety reduces bugs and removes the need for verbose tests.

6.1 Client-server Communication

Client-server communications are common along with popularization of the Internet, such as HTTP.

Figure 6.1 is a chat system where the user can post messages and add reactions. A record and functions in lines 3-6 are definitions of a communication library whose details are hidden. In lines 8-13, effects related to the communication are defined. These are `Connected` meaning that a session between a server and a client is established, `Message` meaning data sent from a client to a server (request) and `Response` meaning data sent from a server to a client (response). A definition of an effect can have multiple operations and these are handled by one identifier of the effect. Effect `Message` has two operations `message` and `reaction`. Operation `message` meaning a message of the chat and returns the ID of one. This has `Response` as the requirements of a capability and it means the corresponding response to a client.

Function `server` (line 15-28) is a implementation of the server side. This function listens to wait connections from clients (line 17), waits requests from a connected client (line 19) as effects `Message`, and handles requests from a client (line 21-27). The handler of `Message.message` makes a response, increments `num_of_msg` that denotes the total number of created messages, and returns the number. The handler of `Message.reaction` makes a response with message `id` and the `name` of a reaction (e.g. `+1`). This handler really adds a reaction named `name` to the message of ID `id` in the production program.

Function `client` denotes a implementation of the client side. This function connects to a server `localhost` (line 32), sends two messages, and adds reactions to each sent message (line 33-36). The handler of `Response.response` prints a message from the server (line 38-42) and the handler of `Connected` just prints "`Connected`" (line 43-45).

The flow of execution can be described in the following way. Note that functions `server` and `client` are executed as one program. This is useful for debugging. Of course these functions are separated into individual programs

in the production usage. This program firstly executes function `main` which calls function `server`, and this function calls function `wait`. Then, function `wait` internally calls function `client`. In function `client`, effects `Message` are raised and these effects are handled in function `server`. The handler of `Message` raises an effect `response` back to the location where effect `Message` is raised. This effect `response` handled in function `client`.

Processes in the server and the client are totally separated into two functions, and these are written in natural way.

```
1  module chat
2
3  record Client()
4  def connect(host: String) : Unit / { Connected } = ...
5  def wait(c: Client) { cli: Unit } : Unit = ...
6  def accept { f: Client => Unit } : Unit = ...
7
8  effect Connected(): Unit
9  effect Message {
10   def message(msg: String): Int / { Response }
11   def reaction(id: Int, name: String): Unit / { Response } }
12 effect Response {
13   def response(msg: String): Unit }
14
15 def server() : Unit / { Console } = {
16   var num_of_msg = 0
17   accept { c =>
18     try {
19       c.wait { client() }
20     }
21     with Message {
22       def message(msg) = resume {
23         do response(msg);
24         num_of_msg = num_of_msg + 1;
25         num_of_msg }
26       def reaction(id, name) = resume {
27         do response("React to "++show(id)++" with "++show(name)) }
28     } } }
29
30 def client(): Unit / { Console, Message } = {
31   try {
32     connect("localhost")
33     var id = do message("Hello")
34     do reaction(id, "+1")
35     id = do message("World")
36     do reaction(id, "+1")
37   }
38   with Response {
39     def response(msg) = {
40       println("Server: " ++ show(msg))
41       resume(())
42     } }
43   with Connected {
44     println("Connected")
45     resume(()) } }
46
47 def main() = server()
```

FIGURE 6.1: An application of chat

```
Connected
Server: Hello
Server: React to 1 with +1
Server: World
Server: React to 2 with +1
```

FIGURE 6.2: The result of Figure 6.1

6.2 Token Ring

Figure 6.3 is a more interesting example of communication implementing token-ring. Token-ring is an implementation of ring network in which a token is passed around. Each node is connected to the next node and the last node is connected to the first node. As a result, nodes are connected like a ring. The node that has the token, has the right to send data. The node adds data to the token and gives it to the next node. The receiver node takes data from the token.

Figure 6.5 shows an example of token ring. If we assume that node 0 sends data to node 1, an empty token flows node 0 to node 1. Node 1 adds data to the token and the token with data flows from node 1 to node2, then to node 0. Lastly, node 0 obtains data from the token.

In Figure 6.3, effect `Token` is used to transfer data `Frame` which has source and destination indices as well as the message (line 4-5). Effect `Terminate` is used by function `node` to tell that it is the last node (line 7). Figure 6.6 shows that how functions `node` and `tokenRing` are called and how the token flows. Function `tokenRing` called in `main` calls function `node`. Function `node` either calls the next node (line 12) or raises an effect `Terminate` if the current node is the last one (line 14). Effect `Terminate` is handled in `tokenRing` (line 39-40). This flow is shown in Figure 6.6 (left). Lines denote function call and dotted lines denote flows of raised effect. Next, the handler of `Terminate` raises `Token` back to the last node (line 40). This is the place where bidirectional effects are used. The handler of `Token` (line 15-33) catches effects in each nodes and raises new effect `Token` (line 30, 32, and 33) An effect `Token` is eventually handled in `tokenRing` (line 41-44) and function `tokenRing` is called to repeat the flow as described above (line 43). This flow is shown in Figure 6.6 (right).

Figure 6.4 shows the result of Figure 6.3. Output with a number and a value shows the token in each node. For example, `2 Some(Frame(5, 3, I'm 3!))` means the token has the frame and in node 2. If the token is in the target node, the node outputs the source node and the message. In the result, node 3 sends "I'm 3!" to node 5 and we can see this by the flow of the data through node 3 to node 5.

```
1  module tokenring
2  import immutable/option
3
4  record Frame(destination: Int, source: Int, message: String)
5  effect Token(data: Option[Frame]): Unit
6
7  effect Terminate(): Unit / { Token }
8
9  def node(id: Int, N: Int) : Unit / { Token, Terminate, Console } = {
10   try {
11     if (id < N) {
12       node(id + 1, N) }
13     else {
14       do Terminate() } }
15   with Token { data =>
16     println(id); println(data)
17
18     val newData = data match {
19       case None() => None[Frame]()
20       case Some(frame) =>
21         if (frame.destination == id) {
22           println(" Data sent from: "); println(frame.source);
23           println(" Message: "); println(frame.message)
24           None[Frame]() }
25         else { data } }
26
27     newData match {
28       case None =>
29         if (id == 3) {
30           do Token(Some(Frame(5, id, "I'm 3!"))) }
31         else {
32           do Token(None) }
33       case Some(_) => do Token(newData) } } }
34
35 def tokenRing(N: Int,  loopCount: Int,  loopNum: Int,
36               data: Option[Frame]) : Unit / { Console } = {
37   try {
38     node(0, N) }
39   with Terminate {
40     resume { do Token(data) } }
41   with Token { data =>
42     if (loopCount < loopNum) {
43       tokenRing(N, loopCount + 1, loopNum, data) }
44     else { () } } }
45
46 def main() = tokenRing(5, 0, 2, None[Frame]())
```

FIGURE 6.3: An application of token ring

```
5 None()
4 None()
3 None()
2 Some(Frame(5, 3, I'm 3!))
1 Some(Frame(5, 3, I'm 3!))
0 Some(Frame(5, 3, I'm 3!))
5 Some(Frame(5, 3, I'm 3!))
 Data sent from: 3
 Message: I'm 3!
4 None()
3 None()
2 Some(Frame(5, 3, I'm 3!))
1 Some(Frame(5, 3, I'm 3!))
0 Some(Frame(5, 3, I'm 3!))
5 Some(Frame(5, 3, I'm 3!))
 Data sent from: 3
 Message: I'm 3!
4 None()
3 None()
2 Some(Frame(5, 3, I'm 3!))
1 Some(Frame(5, 3, I'm 3!))
0 Some(Frame(5, 3, I'm 3!))
```

FIGURE 6.4: The result of Figure 6.3 (processed)

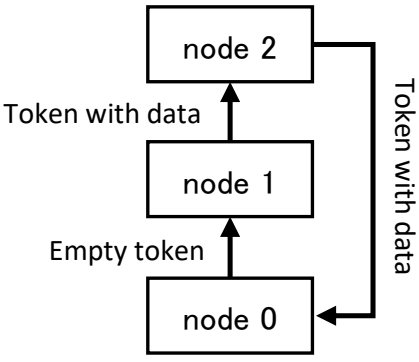

FIGURE 6.5: An example of token ring

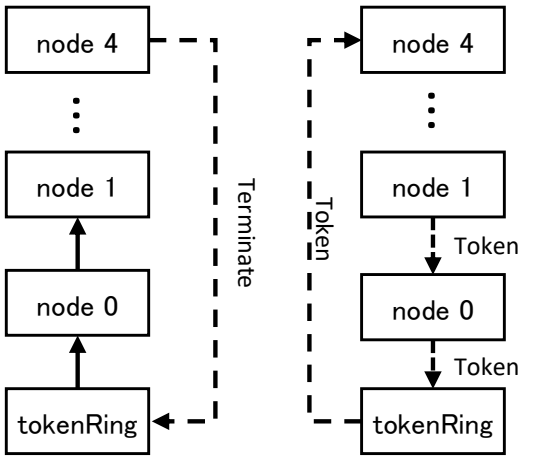

FIGURE 6.6: An example of token ring

Chapter 7

Related works

7.1 Shallow Handlers

There are two kinds of plain handlers. One is *deep handlers* used in languages explained before, such as Effekt and Koka. The other is *shallow handlers*. The difference between shallow handlers and deep handlers is whether the resumption includes try-with clause. For example, consider the following code:

```
1 try {
2   do Exn()
3   do Exn()
4 } with Exn {
5   resume(42)
6 }
```

As we explained in Chapter 2, `resume` in *deep handlers* is the computation after evaluating `do Exn()` in line 2, that is, `resume` is `try { □; do Exn() } with Exn { resume(42) }` . Note that a symbol □ represents a "hole", i.e., the evaluation position. If the same effect is raised after resumption, the effect is handled by the same handler.

On the other hand, `resume` in *shallow handlers* does not include try-with clause. In our particular example, `resume` is just `do Exn()`. This means the second `do Exn()` is handled by the outer handler.

Shallow handlers are useful for programs with mutually recursive functions that raise effects to each other, such as `ping` and `pong` mentioned in Chapter 2. It has been shown that using shallow handlers to implement countdown and pipes leads to better performance in terms of execution time Hillerström and Lindley, 2018. Using bidirectional effects, we can implement such programs while keeping track of effects of operation calls. For example, if the bidirectional effect `Eff` is defined as:

```
effect Eff() : Unit / { Read, Print }
```

Then we know that an operation call `do Eff()` may raise effects `Read` and `Print`. This sequencing of effects can be used for providing readable error messages. For example, if a user forget to write the handler of `Print` in the following program:

```
def main() : Unit / {} = {
  try {
    Eff()
  } with Eff { resume { () }
  } with Read { resume(()) }
}
```

The compiler with bidirectional effects reports the following error:

```
[error] test.effekt:5:1: Unhandled effects: Print
def main() : Unit / {} = {
```

In addition, bidirectional effects are easier to reason about, because they are based on deep handlers, which have a simpler semantics compared to shallow handlers (Kammar et al., 2013; Lindley et al., 2017);

7.2 Session Types

Session types (Honda et al., 1998) are a type discipline for communication-based programming. Users can write how nodes to communicate and what data is transferred with types. As we saw in chapter 6, bidirectional effects can also express programs with communication. Bidirectional effects do not support features of session types, for example orders of communication, subtyping, and preventing deadlocks. One of advantage of bidirectional effects is that users can write communication programs type safely with familiar grammar such as Java.

Chapter 8

Conclusion

We extended Effekt language with bidirectional algebraic effects to allow manipulation of complex control flow in a potentially practical language with algebraic effects. We found that implementation of bidirectional effect to the compiler and the runtime system of Effekt was easy thanks to the carefully designs of these. However, it was not possible to prove soundness of the extended Effekt by simply modifying the original proof because blocks (lambdas) are not first-class values. Blocks should be treated as pseudo first-class values because blocks are passed through `resume` and a operation call. We showed that some programs with bidirectional control flow can be implemented by extended Effekt, for example generator and communication.

As future work, there are three topics remaining. First, we complete formalization. To finish the soundness proof, we modify the formalize and proofs. Second, we modify the compiler and runtime system to work communication examples really. We need to modify the compiler to generate a server and a client program separately, and the runtime system to let these program communicate over TCP. Third, we measure performances such as compiling time and execution time to examine effects of this extension.

Bibliography

Biernacki, Dariusz, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski (Dec. 2019). "Binders by Day, Labels by Night: Effect Instances via Lexically Scoped Handlers". In: *Proc. ACM Program. Lang.* 4.POPL. DOI: 10.1145/3371116. URL: https://doi.org/10.1145/3371116.

Brachthäuser, Jonathan Immanuel, Philipp Schuster, and Klaus Ostermann (Nov. 2020). "Effects as Capabilities: Effect Handlers and Lightweight Effect Polymorphism". In: *Proc. ACM Program. Lang.* 4.OOPSLA. DOI: 10.1145/3428194. URL: https://doi.org/10.1145/3428194.

(chez (chez scheme)). https://www.scheme.com/.

Effekt Language: Home. https://effekt-lang.org/.

Fowler, Martin (2004). *Inversion of Control Containers and the Dependency Injection pattern*. https://martinfowler.com/articles/injection.html.

Hillerström, Daniel and Sam Lindley (2018). "Shallow effect handlers". In: *Asian Symposium on Programming Languages and Systems*. Springer, pp. 415–435.

Honda, Kohei, Vasco T Vasconcelos, and Makoto Kubo (1998). "Language primitives and type discipline for structured communication-based programming". In: *European Symposium on Programming*. Springer, pp. 122–138.

Kammar, Ohad, Sam Lindley, and Nicolas Oury (2013). "Handlers in Action". In: *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*. ICFP '13. Boston, Massachusetts, USA: Association for Computing Machinery, 145–158. ISBN: 9781450323260. DOI: 10.1145/2500365.2500590. URL: https://doi.org/10.1145/2500365.2500590.

Leijen, Daan (2017). "Type Directed Compilation of Row-Typed Algebraic Effects". In: *Proceedings of Principles of Programming Languages (POPL'17), Paris, France*. URL: https://www.microsoft.com/en-us/research/publication/type-directed-compilation-row-typed-algebraic-effects/.

Lindley, Sam, Conor McBride, and Craig McLaughlin (2017). *Do be do be do*. arXiv: 1611.09259 [cs.PL].

Osvald, Leo, Grégory Essertel, Xilun Wu, Lilliam I. González Alayón, and Tiark Rompf (Oct. 2016). "Gentrification Gone Too Far? Affordable 2nd-Class Values for Fun and (Co-)Effect". In: *SIGPLAN Not.* 51.10, 234–251. ISSN: 0362-1340. DOI: 10.1145/3022671.2984009. URL: https://doi.org/10.1145/3022671.2984009.

Plotkin, Gordon and Matija Pretnar (2013). "Handling Algebraic Effects". In: *Logical Methods in Computer Science* 9.4. Ed. by AndrzejEditor Tarlecki. ISSN: 1860-5974. DOI: 10.2168/lmcs-9(4:23)2013. URL: http://dx.doi.org/10.2168/LMCS-9(4:23)2013.

Pretnar, Matija (2015). "An Introduction to Algebraic Effects and Handlers. Invited tutorial paper". In: *Electronic Notes in Theoretical Computer Science* 319. The 31st Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXI)., pp. 19 –35. ISSN: 1571-0661. DOI: https://doi.org/10.1016/j.entcs.2015.12.003. URL: http://www.sciencedirect.com/science/article/pii/S1571066115000705.

Zhang, Yizhou, Guido Salvaneschi, and Andrew C. Myers (Nov. 2020). "Handling Bidirectional Control Flow". In: *Proc. ACM Program. Lang.* 4.OOPSLA. DOI: 10.1145/3428207. URL: https://doi.org/10.1145/3428207.

Appendix A

Soundness Proofs of extended Effekt and System Ξ

A.0.1 Lemma A.1

(EXPRESSION VALUE SUBSTITUTION) Given a statement $\Gamma, x : \tau' \mid \Delta \mid \Xi \vdash s : \tau$ and a value $\Gamma \vdash v : \tau'$, we have $\Gamma \mid \Delta \mid \Xi \vdash s[x \mapsto v] : \tau$.

Case (CAP) Straightforward

Case (CAPCALL) Straightforward

Case (HANDLE) Straightforward

Case (CAPVAR) Straightforward

A.0.2 Lemma A.2

(BLOCK VALUE SUBSTITUTION) Given a statement $\Gamma \mid \Delta, f : \sigma \mid \Xi \vdash s : \tau$ and a block value $\Gamma \mid \Delta \mid \Xi \vdash w : \sigma$, we have $\Gamma \mid \Delta \mid \Xi \vdash s[f \mapsto w] : \tau$.

Case (CAP) Straightforward

Case (CAPCALL) Straightforward

Case (HANDLE) Straightforward

Case (CAPVAR) Straightforward

A.0.3 Lemma A.2′

(CAPABILITY VALUE SUBSTITUTION)
Given a statement $\Gamma \mid \Delta, c : \tau \to \sigma \mid \Xi \vdash s : \tau$ and a block value $\Gamma \mid \Delta \mid \Xi \vdash w : \tau \to \sigma$, we have $\Gamma \mid \Delta \mid \Xi \vdash s[c \mapsto w] : \tau$.

Case (CAP) Straightforward

Case (CAPCALL) Straightforward

Case (HANDLE) Straightforward

Case (CAPVAR) Straightforward

A.0.4 Lemma A.3

(LABEL CONTEXT WEAKENING) If $\Gamma \mid \Delta \mid \Xi \vdash s : \tau$ then for $l \notin \Xi$ and $\Xi = \Xi_1, \Xi_2$ we have $\Gamma \mid \Delta \mid \Xi_1, l : \tau', \Xi_2 \vdash s : \tau$.

Case (CAP) Straightforward

Case (CAPCALL) Straightforward

Case (HANDLE) Straightforward

Case (CAPVAR) Straightforward

A.0.5 Lemma A.4$_{cap}$

(CORRESPONDENCE OF LABELS FOR CAPABILITIES) If $\Gamma \mid \Delta \mid \Xi \vdash E[s] : v$ where $E : \sigma' \to v$ and $s = (\mathbf{cap}_l\{(x, k) \Rightarrow s''\})(v)$, then $\Gamma \mid \Delta \mid \Xi, \Xi' \vdash s : \sigma'$ with $\lceil E \rceil = \Xi'$.

This proof is the same as the original proof. Note that if $E = \Box$, then $v = \sigma$, otherwise $v = \tau$.

□

A.0.6 Theorem 5.1

(TRANSLATION PRESERVES WELL-TYPEDNESS) If $\Gamma \mid \Delta \mid \Sigma \vdash s : \tau \mid \varepsilon$, then $\Gamma \mid \mathcal{T}[\Delta] + \mathcal{T}[\varepsilon] \mid \emptyset \vdash \mathcal{S}[s] : \tau$.

Case (EFFECT) Straightforward

Case (EFFECTCALL) From the assumptions of [EFFECTCALL], we have:

$$\Gamma \vdash e_1 : \tau_1 \tag{A.1}$$
$$\Sigma(F) = \tau_1 \to \tau_0 / \varepsilon' \tag{A.2}$$

From (A.2), we know that the type of F in the translated block context is $\tau_1 \to \tau_0$.
From [BLOCKVAR] and $\Delta'(F) = \tau_1 \to \tau_0$, we derive:

$$\Gamma \mid \Delta' \mid \emptyset \vdash F : \tau_1 \to \tau_0 \tag{A.3}$$
$$\text{where } \Delta' = \mathcal{T}[\Delta] + \mathcal{T}[\{F\} \cup \varepsilon']$$

Finally we apply [CALL] and obtain:

$$\Gamma \mid \mathcal{T}[\Delta] + \mathcal{T}[\{F\} \cup \varepsilon'] \mid \emptyset \vdash F(e_1) : \tau_0$$

Case (TRY) From the assumptions of [TRY], we have:

$$\Sigma(F) = \tau_1 \to \tau_0 / \varepsilon_1 \tag{A.4}$$

$$\Gamma|\Delta|\Sigma \vdash s : \tau|\varepsilon \tag{A.5}$$

$$\Gamma, x_1 : \tau_1|\Delta, \mathsf{resume} : (() \to \tau_0/\varepsilon_1) \to \tau/\phi|\Sigma \vdash s' : \tau|\varepsilon_0 \tag{A.6}$$

From (A.4), the type of F in the translated block context is $\tau_1 \to \tau_0$.

From (A.5) and the IH, we have $\Gamma \mid \mathcal{T}[\Delta] + \mathcal{T}[\varepsilon] \mid \emptyset \vdash \mathcal{S}[s] : \tau$. By applying weakening, we can derive:

$$\Gamma \mid \mathcal{T}[\Delta] + \mathcal{T}[(s \backslash \{F\}) \cup \varepsilon_0], F : \tau_1 \to \tau_0 \mid \emptyset \vdash \mathcal{S}[s] : \tau \tag{A.7}$$

Similarly, from (A.6) and the IH, we have $\Gamma, x : \tau_1 \mid \mathcal{T}[\Delta] + \mathcal{T}[\varepsilon_0], \mathsf{resume} : (() \to \tau_0) \to \tau \mid \emptyset \vdash \mathcal{S}[s'] : \tau$, and by weakening, we obtain:

$$\Gamma, x : \tau_1 \mid \mathcal{T}[\Delta] + \mathcal{T}[(s \backslash \{F\}) \cup \varepsilon_0], r : ((\mathcal{T}[\varepsilon_0]) \to \tau_0) \to \tau \mid \emptyset \vdash \mathcal{S}[s'] : \tau \tag{A.8}$$

Lastly, using (A.7), (A.8), and [HANDLE], we derive:

$$\Gamma \mid \mathcal{T}[\Delta] + \mathcal{T}[(s \backslash \{F\}) \cup \varepsilon_0] \mid \emptyset \vdash \textbf{handle}\ \{F \Rightarrow \mathcal{S}[s]\}\ \textbf{with}\ \{(x, \mathit{resume}) \Rightarrow \mathcal{S}[s']\} : \tau \tag{A.9}$$

□