Contextual Effect Polymorphism Meets Bidirectional Effects (Extended Abstract)

Kazuki Niimi
Tokyo Institute of Technology
Tokyo, Japan
niimi.k.ab@prg.is.titech.ac.jp

Youyou Cong
Hidehiko Masuhara
Tokyo Institute of Technology
Tokyo, Japan
cong@c.titech.ac.jp,masuhara@acm.org

Jonathan Immanuel Brachthäuser
EPFL
Lausanne, Switzerland
jonathan.brachthauser@epfl.ch

1 Introduction

Algebraic effects and handlers [4, 5] offer a uniform and modular way to support user-defined effects. In the past decade, researchers have been exploring the design space of effect handler calculi, to make good balance between expressiveness, efficiency, and ease of reasoning [2, 7, 8]. There is also a line of work on the applications of effect handlers in specific domains, giving rise to specially-designed syntax, semantics, and implementations [1, 3, 9].

This paper presents an extension of the Effekt language [2] with *bidirectional effects* [9]. Effekt is a research language with native support for effect handlers, based on a unique mechanism called *contextual effect polymorphism*. Specifically, Effekt treats the latent effects of a function as a requirement on its calling context, which makes reasoning of effect-polymorphic functions easier. Bidirectional effects are a variant of effects that can express bidirectional control flow. More precisely, where traditional effect handlers allow control flow to be transferred from the call of an effect operation to the handler, bidirectional effects now symmetrically allow control flow to be transferred in the other direction. We study the combination of contextual effect polymorphism and bidirectional effects both from a theoretical point of view and from an implementation perspective.

Our specific contributions can be summarized as follows.

- We formalize a bidirectional-effects extension of System Ξ, the core language of the Effekt language, and prove its type soundness.
- We implement extended System Ξ by extending the Effekt compiler and runtime system.
- We present two example programs written in extended Effekt that make use of bidirectional effects.

2 Background

We start by briefly reviewing relevant concepts.

2.1 Contextual Effect Polymorphism

Effekt [2] is a language with native support for effect handlers, exploring a programmer-centric approach to effect polymorphism. Unlike traditional languages, where one has to explicitly declare effect-polymorphic functions through effect abstraction (as in $\forall\epsilon.A \to B/\epsilon$), Effekt allows one to use functions effect-polymorphically without requiring such explicit abstraction. This is achieved by employing a *contextual* reading of function effects: a function with latent effects ϵ is understood as requiring its calling context to handle ϵ. The contextual reading, however, poses a challenge to effect safety, because it invalidates the traditional reasoning about the purity of programs. For this reason, Effekt does not support first-class functions. More precisely, Effekt treats functions as *blocks*, which can be passed to, but cannot be returned from, a function.

2.2 Bidirectional Effects

Bidirectional effects [9] are a variation of algebraic effects that can express bidirectional control flow. Unlike ordinary algebraic effects, where one can only transfer control from an initiating effect to a handler, bidirectional effects additionally allow one to transfer control from a handler back to the initiating effect. More specifically, one can handle an effect performed at the handler site in the context where the initiating effect was performed. This ability is useful for implementing examples like generators with concurrent modification of data structures, or async-await with asynchronously raised exceptions.

To ensure effect safety, operations of bidirectional effects are decorated with an extra effect annotation, telling us what effects will be performed by their handlers. For instance, the effect operation

```
effect Yield(a: Int): Unit / { Cleanup }
```

models a generator, yielding values of type `Int`. Annotating the effect operation with an extra effect `Cleanup` allows the handler for `Yield` to signal a cleanup at the call site of `Yield`.

To properly handle effects performed by handlers, the argument of resumptions captured by bidirectional effect handlers is evaluated after β-reduction. For instance, the following program

```
def iter(lst: List[Int]): List[Int] / { Yield } = ...
  // performs Yield and handles Cleanup

def main() = {
```

TyDe '21, August 22, 2021, Online
2021. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00

```
val lst = [0, 6, 4, 2, 8, 9]
val lst2 = try { iter(lst) }
with Yield { x =>
  resume { if (x > 5) do Cleanup() else () }}
println(lst2) }
```

uses a generator `iter` and drops numbers greater than 5. Calling the resumption **`resume`** with the block `{ if (x > 5) do Cleanup() else () }` allows the handler to perform `Cleanup` back to function `iter`, where it needs to be handled.

3 Formalization

We now describe how to extend Effekt with bidirectional effects, and how to adopt its core language System Ξ [2] to the extended setting. Roughly speaking, the necessary changes include (i) augmenting the type of operations with an effect component, and (ii) allowing resumptions to be passed a block (instead of a value, which cannot be a function). Note that a block passed to a resumption represents a thunked computation. We need the thunking because we wish to delay the evaluation of resumption arguments.

3.1 Extended Effekt

Figure 1 presents the specification of extended Effekt. A key addition is ε in the effect signature, which represents the set of effects to be performd by the handler of an operation. This extra component appears both in the syntax of effect declarations and in the typing rules of effect-related constructs. As in Brachthäuser et al. [2], a typing judgment carries three environments: Γ for values, Δ for blocks, and Σ for effects. To keep track of the effects that go backwards, we modify the typing rule [EffectCall] for operation calls. Specifically, we take a union of effects in the conclusion, including the operation F that is being called, and the effects ε that will potentially be performed by the handler. To allow bidirectional control-flow transfers, we also modify the typing rule [Try] for handlers. Here, we make the continuation **`resume`** receive a block (essentially a function) of type $(() \to \tau_0/\varepsilon_1)$ as an argument. Conceptually, this block is evaluated at the call site of the effect operation.

3.2 Extended System Ξ

Figure 2 defines the syntax of extended System Ξ, the core language of Effekt with bidirectional effects. The modifications are mainly made in order to allow resumptions to be called on blocks. In the typing rules [Cap] and [Handle], we find that the parameter type of k (**`resume`** in Effekt) is a block type σ, instead of a value type τ as in the original System Ξ. In the reduction rule (*cap*), we see that a block f passed to a resumption k is evaluated in the captured evaluation context H_l. Note that the parameters $\overline{F}$ of a block call $f(\overline{F})$ are specified by the translation of an effect call, namely $\mathcal{S}[\![\mathbf{do}\ F(e_1)]\!]$.

Syntax:

Statements

$s ::= \ \ldots$

$\quad\quad \mathbf{effect}\ F(x:\tau):\tau\ /\varepsilon\ ;\ s$ effect declaration

Syntax of Types:

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

Typing rules:

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

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

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

Figure 1. Extended Effekt (modifications in gray)

We prove the soundness of extended System Ξ by showing the following two theorems.

Theorem 3.1 (Progress). *If $\emptyset \mid \emptyset \mid \emptyset \vdash s : \tau$, then s is a value v or $s \mapsto s'$.*

Theorem 3.2 (Preservation). *If $\emptyset \mid \emptyset \mid \emptyset \vdash s : \tau$ and $s \mapsto s'$, then $\emptyset \mid \emptyset \mid \emptyset \vdash s' : \tau$.*

4 Examples

We modified the Effekt compiler and runtime[1] according to the formalization discussed in the previous section. Here, we present two examples written in extended Effekt.

4.1 Client-server Communication

Bidirectional communication commonly occurs in client-server systems. As a concrete example, let us consider a chat system, where the user can post messages and add reactions. In Figure 3, we define two effects necessary for implementing the chat system. The `Message` effect includes operations for sending data to the server, and the `Response` effect has an operation for responding to the client's action. We see that the operations `message` and `reaction` have `Response` as their effect, because each action has a corresponding response from the server. Together with these effects, we define the

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

Typing rules:

$$\frac{\Xi = \Xi_1, l : \tau, \Xi_2 \qquad \Gamma, x_1 : \tau_1 | \Delta, k : (\overline{\sigma} \to \tau_0) \to \tau | \Xi_1 \vdash s : \tau}{\Gamma | \Delta | \Xi \vdash \mathbf{cap}_l \{(x_1 : \tau_1, k : (\overline{\sigma} \to \tau_0) \to \tau) \Rightarrow s\} : (\tau_1, \overline{\sigma}) \to \tau_0} \text{[Cap]}$$

$$\frac{\Gamma \mid \Delta, F : (\tau_1, \overline{\sigma}) \to \tau_0 \mid \Xi \vdash s : \tau \qquad \Gamma, x : \tau_1 \mid \Delta, k : (\overline{\sigma} \to \tau_0) \to \tau \mid \Xi \vdash s' : \tau}{\Gamma \mid \Delta \mid \Xi \vdash \mathbf{handle}\ \{F \Rightarrow s\}\ \mathbf{with}\ \{(x, k) \Rightarrow s'\} : \tau} \text{[Handle]}$$

Reduction Rules:

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

Translation of Effect Types:

$$\mathcal{T}[\![F]\!] = (\tau_1, F_1, ..., F_n) \to \tau_0 \quad \text{where } \Sigma(F) = \tau_1 \to \tau_0 / F_1, ..., F_n$$

Translation of Statements:

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

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

Figure 2. Typing, reduction, and translation of System Ξ^+

`client` and `server` functions. The former sends a message while raising the `Message` effect, which is to be handled by the latter. In the course of handling, `server` performs the `Response` effect, which is listed in the signature of `message`. The behavior of the chat system is depicted in Figure 4.

```
effect Message {
  def message(msg: String): Int / { Response }
  def reaction(id: Int): Unit / { Response } }
effect Response {
  def response(msg: String): Unit }

def client(): Unit / { Message } = { ... }
def server() : Unit = try { ... } with Message {
  ... resume { ... do Response("Hi back") ... } ... }
```

Figure 3. Implementation of chat system

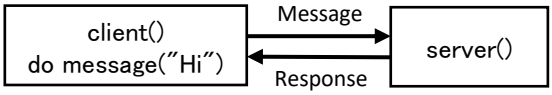

Figure 4. Behavior of chat system

4.2 Token Ring

Bidirectional effects are also useful for implementing a token-ring network [6]. As shown in Figure 6 (left), a token-ring network consists of multiple hosts that send around a token in one direction. We implement such a network by declaring two effects (Figure 5): `Token`, which transfers data `Frame`, and `Terminate`, which represents the last node. A token ring can now be described in terms of two functions `node` and `tokenRing`. The former either calls the next node or performs the `Terminate` effect if the current node is the last one. The latter handles the `Terminate` effect performd by `node` and itself performs the `Token` effect back to the last node. Figure 6 (middle and right) shows how these functions make `Token` flow through nodes.

```
record Frame(...)
effect Token(data: Option[Frame]): Unit

effect Terminate(): Unit / { Token }
```

Figure 5. Implementation of token ring

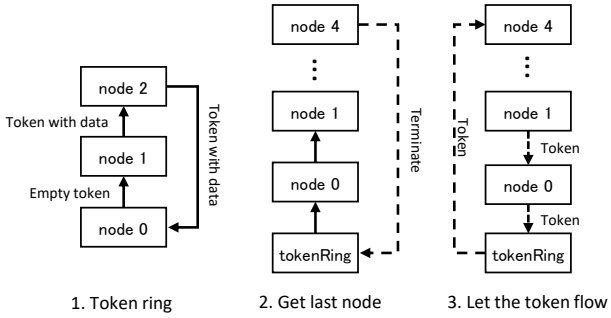

Figure 6. Behavior of token ring

5 Conclusion and Future Work

We extended the Effekt language with bidirectional algebraic effects. Specifically, we proved the soundness of the extended core language, and presented two examples written in extended Effekt.

While the soundness theorems in Section 3 were established in pencil-and-paper proofs, we fully mechanized a variation of the extended language in Coq. A mechanization of the calculus as presented here is under way.

References

[1] Danel Ahman and Matija Pretnar. 2021. Asynchronous Effects. *Proc. ACM Program. Lang.* 5, POPL, Article 24 (Jan. 2021), 28 pages. https://doi.org/10.1145/3434305

[2] Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. 2020. Effects as Capabilities: Effect Handlers and Lightweight Effect Polymorphism. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 126 (Nov. 2020), 30 pages. https://doi.org/10.1145/3428194

[3] Stephen Dolan, Spiros Eliopoulos, Daniel Hillerström, Anil Madhavapeddy, K. C. Sivaramakrishnan, and Leo White. 2018. Concurrent System Programming with Effect Handlers. In *Trends in Functional Programming*, Meng Wang and Scott Owens (Eds.). Springer International Publishing, Cham, 98–117.

[4] Gordon Plotkin and Matija Pretnar. 2013. Handling Algebraic Effects. *Logical Methods in Computer Science* 9, 4 (Dec 2013), 1 – 36. https://doi.org/10.2168/lmcs-9(4:23)2013

[5] Matija Pretnar. 2015. An Introduction to Algebraic Effects and Handlers. Invited tutorial paper. *Electronic Notes in Theoretical Computer Science* 319 (2015), 19 – 35. https://doi.org/10.1016/j.entcs.2015.12.003 The 31st Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXI).

[6] Norman C Strole. 1987. The IBM token-ring network—A Functional Overview. *IEEE Network* 1, 1 (1987), 23–30.

[7] Ningning Xie, Jonathan Immanuel Brachthäuser, Daniel Hillerström, Philipp Schuster, and Daan Leijen. 2020. Effect Handlers, Evidently. *Proc. ACM Program. Lang.* 4, ICFP, Article 99 (Aug. 2020), 29 pages. https://doi.org/10.1145/3408981

[8] Yizhou Zhang and Andrew C. Myers. 2019. Abstraction-Safe Effect Handlers via Tunneling. *Proc. ACM Program. Lang.* 3, POPL, Article 5 (Jan. 2019), 29 pages. https://doi.org/10.1145/3290318

[9] Yizhou Zhang, Guido Salvaneschi, and Andrew C. Myers. 2020. Handling Bidirectional Control Flow. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 139 (Nov. 2020), 30 pages. https://doi.org/10.1145/3428207