Session-Typed Effect Handlers (Extended Abstract)

WENHAO TANG, The University of Edinburgh, United Kingdom

We explore how to design a novel effect system for algebraic effects and handlers inspired by session types, especially the notions of channels and protocols.

1 INTRODUCTION

Effect handlers [Plotkin and Pretnar 2013] and session types [Honda 1993] are two hot topics in programming language research nowadays. Algebraic effects [Plotkin and Power 2003] and effect handlers offer an expressive and modular way to implement computational effects. To enable safe and scalable usage of effect handlers, the type system is usually extended with an effect system to track the effects used by programs. Session types provide static guarantees that concurrent programs respect communication protocols. Intuitively, the interaction between effects and handlers can also be described as communication protocols. Can we design effect systems for effect handlers using the idea of session types?

As background, in the past ten years, many approaches to effect systems for effect handlers have been proposed. The mainstream ones are row-based effect systems as implemented in Links [Hillerström and Lindley 2016] and Koka [Leijen 2017], which track effects as row types and achieve compositionality via row polymorphism. Eff infers more precise effect types via effect subtyping [Bauer and Pretnar 2013; Karachalias et al. 2020; Pretnar 2014] at the cost of explicit constraints in types. Frank [Lindley et al. 2017] simplifies row-based effect systems using the idea of contextual effects. Effekt [Brachthäuser et al. 2022, 2020] further develops contextual effect polymorphism, totally getting rid of row variables. Moreover, some effect systems also support named handlers (also called lexically-scoped handlers) [Biernacki et al. 2020; Brachthäuser et al. 2022; de Vilhena and Pottier 2023; Xie et al. 2022; Zhang and Myers 2019], which associate effects with their handlers to solve the effect encapsulation problem and ease program reasoning. However, there is still no ultimate effect system for algebraic effects and handlers.

We explore the possibilities of session-typed effect handlers, a novel design discipline towards effect systems. Specifically, we propose a core calculus $\lambda^{\boxtimes}_{\text{eff}}$ with a novel effect system inspired by session types, especially the notions of *channels* and *protocols*. In $\lambda^{\boxtimes}_{\text{eff}}$, all effects are invoked on channels and all handlers are installed on channels. $\lambda^{\boxtimes}_{\text{eff}}$ tracks channels as second-class values to avoid escaping. Different channels do not interfere with each other; effects in one channel must and can only be handled in the dual channel. Channels provide a generalised version of named handlers [Biernacki et al. 2020; Xie et al. 2022]. Channels carry protocols which are similar to session types and specify how effects and handlers interact with each other. Different from conventional effect systems for effect handlers [Hillerström et al. 2016; Leijen 2017], protocols in $\lambda^{\boxtimes}_{\text{eff}}$ not only track which effects are used, but also track which handlers are installed. Furthermore, the rich structure of protocols allow us to easily track more involved control flows like bidirectional effects [Zhang et al. 2020] in a more precise manner.

To be clear, there has been work on extending session-typed π-calculus with effects and one-shot handlers [Qian 2022], comparing the expressiveness of some specific forms of session types and effect types [Orchard and Yoshida 2016], encoding effects and one-shot handlers with coroutines [Kawahara and Kameyama 2020; Phipps-Costin et al. 2023], and encoding session-typed communication with effect handlers [Kammar et al. 2013; Lindley et al. 2017]. This work is about

This is an extended abstract for the student research competition of POPL 2024.

something quite different, namely exploring new effect systems for effect handlers instead of encoding some restricted forms of them via concurrent processes or vice versa. This has a non-negligible influence on our design choice; for example, it is important for $\lambda^{\boxtimes}_{\text{eff}}$ to support standard features shared by other calculi with algebraic effects and handlers like multi-shot handlers.

The main advantages of the effect system of $\lambda^{\boxtimes}_{\text{eff}}$ are:

- Channels generalise handler names in the sense that different handlers can share one channel, while different named handlers must have their own names.
- Channels easily support shallow handlers, while previous systems named handlers do not.
- Protocols can track which handlers are installed and in which order they are installed.
- Protocols can track involved control flows like bidirectional effects flexibly.

Section 2 gives an overview of $\lambda^{\boxtimes}_{\text{eff}}$ with several examples showing how channels and protocols work in practice. Section 3 shows the syntax, kinding rules, subtyping rules, typing rules, and operational semantics for $\lambda^{\boxtimes}_{\text{eff}}$. Section 4 lists future work.

2 OVERVIEW

This section shows examples of $\lambda^{\boxtimes}_{\text{eff}}$ including unidirectional effects and bidirectional effects.

2.1 Unidirectional Effects

$\lambda^{\boxtimes}_{\text{eff}}$ gives us generalised named handlers which can share names and specify the order of handlers.

Channels give names to handlers. Channels give us named handlers. Consider two operations $\mathsf{Choose} : () \twoheadrightarrow \mathsf{Bool}$ and $\mathsf{Print} : \mathsf{String} \twoheadrightarrow ()$. We obtain a named handler for Choose by creating a single channel with it. We use the syntactic sugar $\mathbf{resume}\, r\, M \equiv (\mathbf{force}\ r)\, (\mathbf{thunk}\ M)$ and $\mathbf{resume}\, r\, V \equiv (\mathbf{force}\ r)\, (\mathbf{thunk}\ (\mathbf{return}\, V))$.

$$\begin{aligned}&\mathbf{fork}_c^{\oplus\{\mathsf{Choose}\}}\ (\mathbf{if}\ (\mathbf{do}_c\ \mathsf{Choose})\ \mathbf{then}\ 20\ \mathbf{else}\ 22)\\ &\qquad\qquad (\mathbf{handle}_{\overline{c}}\ \bigcirc_{\overline{c}}\ \mathbf{with}\ \{\mathsf{Choose}\ r \mapsto \mathbf{resume}\, r\ \mathsf{true} + \mathbf{resume}\, r\ \mathsf{false}\})\end{aligned}$$

The **fork** primitive is used to connect computations and handlers. We name the two computations taken by **fork** as M and N. The $\mathbf{fork}_c^{\oplus\{\mathsf{Choose}\}}\ M\ N$ binds the channel $c : \oplus\{\mathsf{Choose}\}$ in M, and its dual channel $\overline{c} : \&\{\mathsf{Choose}\}$ in N. The protocol type $\oplus\{\mathsf{Choose}\}$ allows unlimited invocation of operation Choose, and its dual $\&\{\mathsf{Choose}\}$ requires deep handling of Choose. Operation invocation $\mathbf{do}_c$ and handling $\mathbf{handle}_{\overline{c}}$ take channels and respect their protocols. The hole $\bigcirc_{\overline{c}}$ is a special variable bound in N which will be filled with **thunk** M during evaluation. This program evaluates to 42.

Channels can be shared by different handlers. One advantage of channels over named handlers is that different handlers can share the same channel. For example, we can create a channel with two operations and handle them separately.

$$\begin{aligned}&\mathbf{fork}_c^{\oplus\{\mathsf{Choose}\,;\,\mathsf{Print}\}}\ (\mathbf{if}\ (\mathbf{do}_c\ \mathsf{Choose})\ \mathbf{then}\ \mathbf{do}_c\ \mathsf{Print}\ \text{"42"}\ \mathbf{else}\ \mathbf{do}_c\ \mathsf{Print}\ \text{"24"})\\ &\qquad\qquad (\mathbf{handle}_{\overline{c}}\ (\mathbf{handle}_{\overline{c}}\ \bigcirc_{\overline{c}}\ \mathbf{with}\ \{\mathsf{Choose}\ r \mapsto \mathbf{resume}\, r\, \mathsf{true}\})\\ &\qquad\qquad\ \ \mathbf{with}\ \{\mathsf{Print}\, s\, r \mapsto \mathsf{write}(s); \mathbf{resume}\, r\, ()\})\end{aligned}$$

Different channels do not interfere with each other. Viewing effectful programs as computation trees, one way to understand channels is that they extract subtrees from them. The handlers in dual channels are restricted to deal with these subtrees instead of the whole computation tree.

Protocols specify the structure of handling. Another benefit provided by protocol types is that we can distinguish between different orders of handlers. For example, we can specify whether we want the local-state semantics or global-state semantics [Pauwels et al. 2019] in types. When using the

syntax . to connect two $\&\{\ldots\}$, the handler of the former appear inside the handler of the latter. We use the syntactic sugar $M \rhd_c H \equiv \mathbf{handle}_c\, M\ \mathbf{with}\ H$.

$$\begin{array}{ll} \mathbf{fork}_c^{\oplus\{\mathsf{Choose}\,;\,\mathsf{Fail}\}.\oplus\{\mathsf{Get}\,;\,\mathsf{Put}\}} & (\ldots) \\ & (\bigcirc_{\overline{c}} \rhd_{\overline{c}} \{\mathbf{return}\ \ldots \mathsf{Choose} \ldots \mathsf{Fail} \ldots\}) \rhd_{\overline{c}} \{\mathsf{Get} \ldots \mathsf{Put} \ldots\} \\ \mathbf{fork}_c^{\oplus\{\mathsf{Get}\,;\,\mathsf{Put}\}.\oplus\{\mathsf{Choose}\,;\,\mathsf{Fail}\}} & (\ldots) \\ & (\bigcirc_{\overline{c}} \rhd_{\overline{c}} \{\mathsf{Get} \ldots \mathsf{Put} \ldots\}) \rhd_{\overline{c}} \{\mathbf{return}\ \ldots \mathsf{Choose} \ldots \mathsf{Fail} \ldots\} \end{array}$$

2.2 Bidirectional Effects

$\lambda^{\boxtimes}_{\text{eff}}$ also supports bidirectional effects. It gives us a more fine-grained way to specify bidirectional interaction between effects and handlers in a similar style to session types.

Protocols specify bidirectional effects. Bidirectional effects enable bidirectional control flow which allows us to transfer control from handlers back to the initial effects. The protocol types of $\lambda^{\boxtimes}_{\text{eff}}$ naturally tracks bidirectional control flow. For example, consider two operations $\mathsf{Ask} : () \twoheadrightarrow \mathsf{String}$ which asks for a string and $\mathsf{Raise} : \forall\alpha.\mathsf{String} \twoheadrightarrow \alpha$ which raises an exception. (Extending $\lambda^{\boxtimes}_{\text{eff}}$ with value polymorphism is standard.) The channel $c : \oplus\{\mathsf{Ask} : \&\{\mathsf{Raise}\}\}$ allows programs to invoke the Ask operation as long as they guarantee to handle exceptions. Its dual channel $\overline{c} : \&\{\mathsf{Ask} : \oplus\{\mathsf{Raise}\}\}$ requires programs to handle Ask, but also allows them to resume with exceptions which are handled in the original channel. We can write a server that might crash and a client that deals with the exception as follows.

$$\begin{array}{ll} \mathbf{fork}_c^{\oplus\{\mathsf{Ask}:\&\{\mathsf{Raise}\}\}} & (\mathbf{handle}_c\ (\mathbf{do}_c\ \mathsf{Ask}; \mathbf{return}\ ())\ \{\mathsf{Raise}\, s\, r \mapsto \mathsf{write}(\text{"error:"} +\!\!+ s)\}) \\ & (\mathbf{handle}_{\overline{c}}\ \bigcirc_{\overline{c}}\ \{\mathsf{Ask}\ r \mapsto \mathbf{resume}\ r\ (\mathbf{do}_{\overline{c}}\ \mathsf{Raise}\ \text{"server crashes"})\}) \end{array}$$

Protocols can specify deeper bidirectional behaviours. Compared with the effect system for bidirectional effects in Zhang et al. [2020], $\lambda^{\boxtimes}_{\text{eff}}$ can express deeper bidirectional behaviours naturally. For example, the channel

$$c : \oplus\{\mathsf{Ask} : \&\{\mathsf{Response} : \oplus\{\mathsf{Ask} : \&\{\mathsf{Response}, \mathsf{Raise}\}\}\}\}$$

indicates that the client only needs to deal with server crashing in the second round of asking. [1]

Recursive protocols. Recursive types are needed to encode endless bidirectional communication. For instance, the ping-pong example in Zhang et al. [2020] is specified by $\mu\alpha.\oplus\{\mathsf{Ping} : \&\{\mathsf{Pong} : \alpha\}\}$. We do not have recursive types in $\lambda^{\boxtimes}_{\text{eff}}$ currently, but it should be standard to extend $\lambda^{\boxtimes}_{\text{eff}}$ with recursive types.

3 THE CORE CALCULUS

The explicit core calculus $\lambda^{\boxtimes}_{\text{eff}}$ is based on call-by-push-value [Levy 2004] extended with constructs for algebraic effects and handlers [Kammar et al. 2013]. We assume readers' familiarity with call-by-push-value and effect handlers.

3.1 Syntax and Kinding

The syntax of $\lambda^{\boxtimes}_{\text{eff}}$ is formally shown in Figure 1. Everything relevant to channels is highlighted. Thunk $\mathbf{thunk}\ M$ captures the used channels Δ in its type $\downarrow^{\Delta} C$. We have channel abstraction $\lambda c^S.\ M$ (which is given type $\forall c^S.C$) and application $M\ c$. Protocol types S have a very similar structure to session types and describe protocols specifying the effectful behaviours of computations. We have empty (or end) protocol type $\diamond$, protocol sequencing $S.S$ representing nested handlers, selection

[1] Another example can be found in https://thwfhk.github.io/files/session-handlers-popl24-src-poster.pdf

$$
\begin{array}{lrl}
\text{Value types} & A, B & ::= \downarrow^{\Delta} C \\
\text{Computation types} & C, D & ::= \uparrow A \mid A \to C \mid \forall c^{S}.\, C \\
\text{Protocol types} & S & ::= \oplus\{\overrightarrow{\ell \triangleright A \twoheadrightarrow B : S}\} \mid \&\{\overrightarrow{\ell \triangleright A \twoheadrightarrow B : S}\} \mid S.S \mid \diamond_{Y} \mid \overline{S} \\
\text{Types} & T & ::= A \mid C \mid E \mid F \mid S \\
\text{Kinds} & K & ::= \mathsf{Type} \mid \mathsf{Comp} \mid \mathsf{Handler} \mid \mathsf{Session}^{Y} \\
\text{Polarity} & Y & ::= \oplus \mid \& \\
\text{Handler types} & F & ::= C \Rightarrow D \\
\text{Type contexts} & \Gamma & ::= \cdot \mid \Gamma, x : A \\
\text{Channel contexts} & \Delta & ::= \cdot \mid \Delta, c : S \\
\text{Channels} & c & ::= c \mid \overline{c} \\
\text{Values} & V, W & ::= x \mid \mathbf{thunk}\ M \\
\text{Computations} & M, N & ::= \mathbf{force}\ V \mid \lambda x^{A}.\, M \mid M\ V \mid \mathbf{let}\ x \leftarrow M\ \mathbf{in}\ N \\
 & & \quad \mid \mathbf{return}\ V \mid \mathbf{do}_{c}\ \ell\ V \mid \mathbf{handle}_{c}\ M\ \mathbf{with}\ H \\
 & & \quad \mid \mathbf{fork}^{S}_{c}\ M\ N \mid \lambda c^{S}.\, M \mid M\ c \\
\text{Handlers} & H & ::= \{\mathbf{return}\ x \mapsto M\} \mid \{\ell\ p\ r \mapsto M\} \uplus H
\end{array}
$$

Fig. 1. Syntax of $\lambda^{\boxtimes}_{\text{eff}}$.

$\oplus\{\overrightarrow{\ell \triangleright A \twoheadrightarrow B : S}\}$ representing effects invocation, and its dual choice $\&\{\overrightarrow{\ell \triangleright A \twoheadrightarrow B : S}\}$ representing effects handling. The $\ell \triangleright A \twoheadrightarrow B : S$ means the label ℓ has signature $A \twoheadrightarrow B$ and continues as protocol type S. We sometimes omit operation signatures and empty protocol types. For unidirectional effects, we only need single-level selection and choice of forms $\oplus\{\overrightarrow{\ell}\}$ and $\&\{\overrightarrow{\ell}\}$. We interpret them as unlimited invocation and deep handling of operations in $\overrightarrow{\ell}$, respectively. In other words, $\oplus\{\overrightarrow{\ell}\}$ means that operations $\overrightarrow{\ell}$ might be invoked, and $\&\{\overrightarrow{\ell}\}$ means that handlers for $\overrightarrow{\ell}$ are installed. The dual operation $\overline{S}$ basically swaps selection and choice. It is defined for well-kinded protocol types as follows.

$$
\begin{array}{rcl}
\overline{\diamond_{\oplus}} & = & \diamond_{\&} \\
\overline{\diamond_{\&}} & = & \diamond_{\oplus} \\
\overline{S_1.S_2} & = & \overline{S_1}.\overline{S_2} \\
\overline{\oplus\{\ell_i \triangleright A_i \twoheadrightarrow B_i : S_i\}_i} & = & \&\{\ell_i \triangleright A_i \twoheadrightarrow B_i : \overline{S_i}\}_i \\
\overline{\&\{\ell_i \triangleright A_i \twoheadrightarrow B_i : S_i\}_i} & = & \oplus\{\ell_i \triangleright A_i \twoheadrightarrow B_i : \overline{S_i}\}_i
\end{array}
$$

It is obvious from definition that $\overline{\overline{S}} = S$. For brevity, we always consider $\overline{\overline{c}}$ and c to be identical.

All protocol types S are required to satisfy the kinding judgement $\vdash S : \mathsf{Session}^{Y}$ which is formally defined in Figure 2.

3.2 Typing

The subtyping relations of protocol types is defined in Figure 3. We write $\vdash S_1 \equiv S_2$ for $\vdash S_1 \leqslant S_2$ and $\vdash S_2 \leqslant S_1$. For brevity, we omit operation signatures and assume that the same operations always have the same signatures. Following the subtyping relation of session types [Gay and Hole 2005], $\oplus$ is contravariant and $\&$ is covariant. Additionally, we have two subtyping rules for the sequencing of $\oplus$ and $\&$, and two equivalence rules showing the transitivity of alternated $\oplus$ and $\&$.

The typing rules of $\lambda^{\boxtimes}_{\text{eff}}$ are shown in Figure 4. The most non-trivial typing rules are T-Fork, T-Handle and T-Handler. The T-Fork rule binds $c : S$ in M, and its dual $\overline{c} : \overline{S}$ and a special variable

$\boxed{\vdash S : \mathsf{Session}^{Y}}$

$$\textsc{K-Select}\quad \frac{[\vdash S_i : \mathsf{Session}^{\&} \qquad \vdash A_i : \mathsf{Type} \qquad \vdash B_i : \mathsf{Type}]_i \qquad \mathsf{unique}(\overrightarrow{\ell_i})}{\vdash \oplus\{\ell_i \triangleright A_i \twoheadrightarrow B_i : S_i\}_i : \mathsf{Session}^{\oplus}}$$

$$\textsc{K-Choice}\quad \frac{[\vdash S_i : \mathsf{Session}^{\oplus} \qquad \vdash A_i : \mathsf{Type} \qquad \vdash B_i : \mathsf{Type}]_i \qquad \mathsf{unique}(\overrightarrow{\ell_i})}{\vdash \&\{\ell_i \triangleright A_i \twoheadrightarrow B_i : S_i\}_i : \mathsf{Session}^{\&}}$$

$$\textsc{K-Seq}\quad \frac{\vdash S_1 : \mathsf{Session}^{Y_1} \qquad \vdash S_2 : \mathsf{Session}^{Y_2}}{\vdash S_1.S_2 : \mathsf{Session}^{Y_1}} \qquad \textsc{K-Empty}\quad \frac{}{\vdash \Diamond_Y : \mathsf{Session}^{Y}}$$

Fig. 2. Kinding rules for the protocol types of $\lambda^{\boxtimes}_{\text{eff}}$.

$\boxed{\vdash E_1 \leqslant E_2}$ $\boxed{\vdash E_1 \equiv E_2}$

$$\vdash \&\{\overrightarrow{\ell_i : E_i}; \overrightarrow{\ell_j : E'_j}\} \leqslant \&\{\overrightarrow{\ell_i : E_i}\}.\&\{\overrightarrow{\ell_j : E'_j}\} \qquad \vdash \oplus\{\overrightarrow{\ell_i : E_i}; \overrightarrow{\ell_j : E'_j}\} \leqslant \oplus\{\overrightarrow{\ell_i : E_i}\}.\oplus\{\overrightarrow{\ell_j : E'_j}\}$$

$$\vdash \oplus\{\overrightarrow{\ell_i : E_i}\}.\&\{\overrightarrow{\ell_j : E'_j}\} \equiv \oplus\{\overrightarrow{\ell_i : E_i.\&\{\overrightarrow{\ell_j : E'_j}\}}\} \qquad \vdash \&\{\overrightarrow{\ell_i : E_i}\}.\oplus\{\overrightarrow{\ell_j : E'_j}\} \equiv \&\{\overrightarrow{\ell_i : E_i.\oplus\{\overrightarrow{\ell_j : E'_j}\}}\}$$

$$\Diamond_{\oplus} \equiv \oplus\{\} \qquad \Diamond_{\&} \equiv \&\{\}$$

Fig. 3. Subtyping rules for the protocol types of $\lambda^{\boxtimes}_{\text{eff}}$.

$\bigcirc_{\overline{c}}$ in N. The thunk variable $\bigcirc_{\overline{c}}$ has protocol type $\Diamond_{\&}$ on $\overline{c}$ which intuitively means no handler has been installed yet. Thus, in order to use it in N, we need to install all handlers required by $\overline{S}$ out of it, which is important for effect safety. The T-Handle rule removes labels $\overrightarrow{\ell_i}$ from the protocol type S of channel c using a special operator $\mathsf{split}(S, \overrightarrow{\ell_i})$. These labels $\overrightarrow{\ell_i}$ are removed from the protocol type of M since the handler H is installed out of M. The T-Handler rule uses S_i as the protocol type of channel c for the parameter of the continuation of each label ℓ_i. Notice that the continuation r_i still captures $c : S$, because deep handlers are recursively installed on the continuations.

The T-Handle and T-Handler rules both use a meta function $\mathsf{split}(S, \overrightarrow{\ell_i})$ to remove the operation labels $\overrightarrow{\ell_i}$ appropriately from E. For $\mathsf{split}(S, \overrightarrow{\ell_i}) = (S_r, \overrightarrow{\ell_i : S_i})$, it finds the $\&\{\overrightarrow{\ell_i : S_i}, \cdots\}$ in all branches of the first-two levels of S that contain all $\overrightarrow{\ell_i}$, extracts the sets $\overrightarrow{\ell_i : S_i}$, checks that all sets are identical, returns the set $\overrightarrow{\ell_i : S_i}$ as the second component and returns the remaining session type as the first component. It is formally defined in Figure 5.

3.3 Semantics

The semantics of $\lambda^{\boxtimes}_{\text{eff}}$ is shown in Figure 6. It follows from the generative semantics of named handlers [Biernacki et al. 2020].

$$\boxed{\Gamma \vdash V : A} \quad \boxed{\Delta \mid \Gamma \vdash M : C} \quad \boxed{\Delta \mid \Gamma \vdash_c H : C \Rightarrow D}$$

$$\text{T-Var}\ \frac{}{\Gamma, x : A \vdash x : A} \qquad \text{T-Thunk}\ \frac{\Delta \mid \Gamma \vdash M : C}{\Gamma \vdash \mathbf{thunk}\ M : \downarrow^{\Delta} C} \qquad \text{T-Force}\ \frac{\Gamma \vdash V : \downarrow^{\Delta'} C \qquad \Delta' \subseteq \Delta}{\Delta \mid \Gamma \vdash \mathbf{force}\ V : C}$$

$$\text{T-Abs}\ \frac{\Delta \mid \Gamma, x : A \vdash M : C}{\Delta \mid \Gamma \vdash \lambda x^{A}.\ M : A \to C} \qquad \text{T-App}\ \frac{\Delta \mid \Gamma \vdash M : A \to C \qquad \Gamma \vdash V : A}{\Delta \mid \Gamma \vdash M\ V : C} \qquad \text{T-CAbs}\ \frac{\Delta, c : S; \Gamma \vdash M : C}{\Delta \mid \Gamma \vdash \lambda c^{S}.\ M : \forall c : S.C}$$

$$\text{T-CApp}\ \frac{\Delta, d : S; \Gamma \vdash M : \forall c : S.C}{\Delta, d : S; \Gamma \vdash M\ d : C[d/c]} \qquad \text{T-CSub}\ \frac{\Delta, c : S' \mid \Gamma \vdash M : C \qquad S \leqslant S'}{\Delta, c : S \mid \Gamma \vdash M : C} \qquad \text{T-Return}\ \frac{\Gamma \vdash V : A}{\Delta \mid \Gamma \vdash \mathbf{return}\ V : \uparrow A}$$

$$\text{T-Do}\ \frac{(c : \oplus\{\ell \triangleright A \twoheadrightarrow B, \ldots\}) \in \Delta \qquad \Gamma \vdash V : A}{\Delta \mid \Gamma \vdash \mathbf{do}_c\ \ell\ V : \uparrow B} \qquad \text{T-Seq}\ \frac{\Delta \mid \Gamma \vdash M : \uparrow A \qquad \Delta \mid \Gamma, x : A \vdash N : C}{\Delta \mid \Gamma \vdash \mathbf{let}\ x \leftarrow M\ \mathbf{in}\ N : C}$$

$$\text{T-Fork}\ \frac{\Delta, c : S; \Gamma \vdash M : \uparrow A \qquad \Delta, \overline{c} : \overline{S} \mid \Gamma, \bigcirc_{\overline{c}} : \downarrow^{\Delta,\, \overline{c}:\Diamond\&} (\uparrow A) \vdash N : C \qquad c, \overline{c} \notin A, C}{\Delta \mid \Gamma \vdash \mathbf{fork}^{E}_{c}\ M\ N : C}$$

$$\text{T-Handle}\ \frac{\begin{array}{c}\mathsf{dom}(H) = \overrightarrow{\ell_i} \qquad (S_r, \_) = \mathsf{split}(S, \overrightarrow{\ell_i}) \\ \Delta,\ c : S_r\ ; \Gamma \vdash M : \uparrow A \qquad \Delta, c : S; \Gamma \vdash_c H : \uparrow A \Rightarrow \uparrow B\end{array}}{\Delta, c : S; \Gamma \vdash \mathbf{handle}_c\ M\ \mathbf{with}\ H : D}$$

$$\text{T-Handler}\ \frac{\begin{array}{c}H = \{\mathbf{return}\ x \mapsto M\} \uplus \{\ell_i\ p_i\ r_i \mapsto N_i\}_i \qquad \Delta, c : S; \Gamma, x : A \vdash M : \uparrow B \\ (\_, \{\ell_i \triangleright A_i \twoheadrightarrow B_i : S_i\}_i) = \mathsf{split}(S, \overrightarrow{\ell_i}) \\ [\Delta, c : S; \Gamma, p_i : A_i, r_i : \downarrow^{\Delta,\, c:S} (\downarrow^{\Delta,\, c:S_i} (\uparrow B_i) \to \uparrow B) \vdash N_i : \uparrow B]_i\end{array}}{\Delta, c : S; \Gamma \vdash_c H : \uparrow A \Rightarrow \uparrow B}$$

Fig. 4. Typing rules for $\lambda^{\boxtimes}_{\text{eff}}$.

$$\begin{array}{rcl}\mathsf{split}(\&\{\overrightarrow{\ell_i : S_i}, \overrightarrow{\ell_j : S'_j}\}, \overrightarrow{\ell_i}) & = & (\&\{\overrightarrow{\ell_j : S'_j}\}, \overrightarrow{\ell_i : S_i}) \\ \mathsf{split}(\oplus\{\overrightarrow{\ell_j : S'_j}\}, \overrightarrow{\ell_i}) & = & (\oplus\{\overrightarrow{\ell_j : S'_j}\}, \overrightarrow{\ell_i : S_i}) \\ & & \text{where } (S'_j, \overrightarrow{\ell_i : S_i}) = \mathsf{split}(S_j, \overrightarrow{\ell_i}) \\ \mathsf{split}(\_, \_) & = & \text{fail, otherwise}\end{array}$$

Fig. 5. Definition of split.

4 FUTURE WORK

Future work includes:

$$
\begin{array}{lll}
\text{E-App} & (\lambda x.M)\, V \rightsquigarrow M[V/x] & \\
\text{E-CApp} & (\lambda c.M)\, l \rightsquigarrow M[l/c] & \\
\text{E-Force} & \textbf{force}\ (\textbf{thunk}\ M) \rightsquigarrow M & \\
\text{E-Seq} & \textbf{let}\ x \leftarrow \textbf{return}\ V\ \textbf{in}\ N \rightsquigarrow N[V/x] & \\
\text{E-Ret} & \textbf{handle}_l\ (\textbf{return}\ V)\ \textbf{with}\ H \rightsquigarrow N[V/x] & \text{where } (\textbf{return}\ x \mapsto N) \in H \\
\text{E-Op} & \textbf{handle}_l\ \mathcal{E}[\textbf{do}_{\overline{l}}\ \ell\ V]\ \textbf{with}\ H \rightsquigarrow N[V/p, (\lambda y.\textbf{handle}_l\ \mathcal{E}[\textbf{force}\ y]\ \textbf{with}\ H)/r] & \\
 & & \text{where } \ell \notin \mathsf{bl}(\mathcal{E}, l) \text{ and } (\ell\ p\ r \mapsto N) \in H \\
\text{E-Fork} & \textbf{fork}_c\ M\ N \rightsquigarrow (N[M/\bigcirc_{\overline{c}}])[l/c, \overline{l}/\overline{c}] & \text{where } l \text{ is fresh} \\
\text{E-Lift} & \mathcal{E}[M] \rightsquigarrow \mathcal{E}[N], & \text{if } M \rightsquigarrow N
\end{array}
$$

Channel instances $l \ldots$

Evaluation contexts $\mathcal{E} ::= [\,] \mid \textbf{let}\ x \leftarrow \mathcal{E}\ \textbf{in}\ N \mid \textbf{handle}_l\ \mathcal{E}\ \textbf{with}\ H$

$$
\begin{array}{rr}
\mathsf{bl}([\,], l) = \emptyset & \mathsf{bl}(\textbf{let}\ x \leftarrow \mathcal{E}\ \textbf{in}\ N, l) = \mathsf{bl}(\mathcal{E}) \\
\mathsf{bl}(\textbf{handle}_l\ \mathcal{E}\ \textbf{with}\ H, l) = \mathsf{bl}(\mathcal{E}) \cup \mathsf{dom}(H) & \mathsf{bl}(\textbf{handle}_{l'}\ \mathcal{E}\ \textbf{with}\ H, l) = \mathsf{bl}(\mathcal{E})
\end{array}
$$

Fig. 6. Small-Step Operational Semantics of $\lambda^{\boxtimes}_{\text{eff}}$

- The current $\lambda^{\boxtimes}_{\text{eff}}$ is specially developed from the conventional point of view of algebraic effects and deep handlers. Channels are unlimited and effects can be arbitrarily invoked and handled. However, session types naturally track more fine-grained information. For instance, $\oplus\{\ell_1 : \oplus\{\ell_2\}\}$ can be interpreted as sequential usage of ℓ_1 and ℓ_2 exactly once. We are exploring how to develop another version of $\lambda^{\boxtimes}_{\text{eff}}$ with linear channels and shallow handlers which could precisely encode session-typed communication on both type and term level.
- It would be interesting to extend $\lambda^{\boxtimes}_{\text{eff}}$ with concurrent semantics. Since effects and handlers are intrinsically non-concurrent, it would be interesting to explore the notion of *non-blocking* effects and their relation to asynchronous effects [Ahman and Pretnar 2021].
- Consider polymorphism and recursive types.
- It would be interesting to explore the usage of multiparty session types [Honda et al. 2008].
- In addition to bidirectional effects, we plan to extend $\lambda^{\boxtimes}_{\text{eff}}$ with more variants of algebraic effects and handlers such as n-ary handlers [Lindley et al. 2017] and higher-order effects [Piróg et al. 2018; Poulsen and van der Rest 2023; van den Berg and Schrijvers 2023; van den Berg et al. 2021].

REFERENCES

Danel Ahman and Matija Pretnar. 2021. Asynchronous effects. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–28. https://doi.org/10.1145/3434305

Andrej Bauer and Matija Pretnar. 2013. An Effect System for Algebraic Effects and Handlers. In *Algebra and Coalgebra in Computer Science*, Reiko Heckel and Stefan Milius (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–16.

Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. 2020. Binders by day, labels by night: effect instances via lexically scoped handlers. *Proc. ACM Program. Lang.* 4, POPL (2020), 48:1–48:29. https://doi.org/10.1145/3371116

Jonathan Immanuel Brachthäuser, Philipp Schuster, Edward Lee, and Aleksander Boruch-Gruszecki. 2022. Effects, capabilities, and boxes: from scope-based reasoning to type-based reasoning and back. *Proc. ACM Program. Lang.* 6, OOPSLA1 (2022), 1–30. https://doi.org/10.1145/3527320

Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. 2020. Effects as capabilities: effect handlers and lightweight effect polymorphism. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 126:1–126:30. https://doi.org/10.1145/3428194

Paulo Emílio de Vilhena and François Pottier. 2023. A Type System for Effect Handlers and Dynamic Labels. In *Programming Languages and Systems - 32nd European Symposium on Programming, ESOP 2023, Held as Part of the European Joint*

Conferences on Theory and Practice of Software, ETAPS 2023, Paris, France, April 22-27, 2023, Proceedings (Lecture Notes in Computer Science, Vol. 13990), Thomas Wies (Ed.). Springer, 225–252. https://doi.org/10.1007/978-3-031-30044-8_9

Simon J. Gay and Malcolm Hole. 2005. Subtyping for session types in the pi calculus. *Acta Informatica* 42, 2-3 (2005), 191–225. https://doi.org/10.1007/S00236-005-0177-Z

Daniel Hillerström and Sam Lindley. 2016. Liberating Effects with Rows and Handlers *(TyDe 2016)*. Association for Computing Machinery, New York, NY, USA, 15–27. https://doi.org/10.1145/2976022.2976033

Daniel Hillerström, Sam Lindley, and KC Sivaramakrishnan. 2016. Compiling Links Effect Handlers to the OCaml Backend. ML Workshop.

Kohei Honda. 1993. Types for Dyadic Interaction. In *CONCUR '93, 4th International Conference on Concurrency Theory, Hildesheim, Germany, August 23-26, 1993, Proceedings (Lecture Notes in Computer Science, Vol. 715)*, Eike Best (Ed.). Springer, 509–523. https://doi.org/10.1007/3-540-57208-2_35

Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2008. Multiparty asynchronous session types. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, George C. Necula and Philip Wadler (Eds.). ACM, 273–284. https://doi.org/10.1145/1328438.1328472

Ohad Kammar, Sam Lindley, and Nicolas Oury. 2013. Handlers in action. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, Greg Morrisett and Tarmo Uustalu (Eds.). ACM, 145–158. https://doi.org/10.1145/2500365.2500590

Georgios Karachalias, Matija Pretnar, Amr Hany Saleh, Stien Vanderhallen, and Tom Schrijvers. 2020. Explicit effect subtyping. *J. Funct. Program.* 30 (2020), e15. https://doi.org/10.1017/S0956796820000131

Satoru Kawahara and Yukiyoshi Kameyama. 2020. One-Shot Algebraic Effects as Coroutines. In *Trends in Functional Programming - 21st International Symposium, TFP 2020, Krakow, Poland, February 13-14, 2020, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 12222)*, Aleksander Byrski and John Hughes (Eds.). Springer, 159–179. https://doi.org/10.1007/978-3-030-57761-2_8

Daan Leijen. 2017. Type Directed Compilation of Row-Typed Algebraic Effects. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL '17)*. Association for Computing Machinery, New York, NY, USA, 486–499. https://doi.org/10.1145/3009837.3009872

Paul Blain Levy. 2004. *Call-By-Push-Value: A Functional/Imperative Synthesis*. Semantics Structures in Computation, Vol. 2. Springer.

Sam Lindley, Conor McBride, and Craig McLaughlin. 2017. Do Be Do Be Do. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL 2017)*. Association for Computing Machinery, New York, NY, USA, 500–514. https://doi.org/10.1145/3009837.3009897

Dominic A. Orchard and Nobuko Yoshida. 2016. Effects as sessions, sessions as effects. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, Rastislav Bodík and Rupak Majumdar (Eds.). ACM, 568–581. https://doi.org/10.1145/2837614.2837634

Koen Pauwels, Tom Schrijvers, and Shin-Cheng Mu. 2019. Handling Local State with Global State. In *Mathematics of Program Construction - 13th International Conference, MPC 2019, Porto, Portugal, October 7-9, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11825)*, Graham Hutton (Ed.). Springer, 18–44. https://doi.org/10.1007/978-3-030-33636-3_2

Luna Phipps-Costin, Andreas Rossberg, Arjun Guha, Daan Leijen, Daniel Hillerström, KC Sivaramakrishnan, Matija Pretnar, and Sam Lindley. 2023. Continuing WebAssembly with Effect Handlers. *Proc. ACM Program. Lang.* 7, OOPSLA2 (2023), 460–485. https://doi.org/10.1145/3622814

Maciej Piróg, Tom Schrijvers, Nicolas Wu, and Mauro Jaskelioff. 2018. Syntax and Semantics for Operations with Scopes. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science* (Oxford, United Kingdom) *(LICS '18)*. Association for Computing Machinery, New York, NY, USA, 809–818. https://doi.org/10.1145/3209108.3209166

Gordon D. Plotkin and John Power. 2003. Algebraic Operations and Generic Effects. *Appl. Categorical Struct.* 11, 1 (2003), 69–94. https://doi.org/10.1023/A:1023064908962

Gordon D. Plotkin and Matija Pretnar. 2013. Handling Algebraic Effects. *Log. Methods Comput. Sci.* 9, 4 (2013).

Casper Bach Poulsen and Cas van der Rest. 2023. Hefty Algebras: Modular Elaboration of Higher-Order Algebraic Effects. *Proc. ACM Program. Lang.* 7, POPL (2023), 1801–1831. https://doi.org/10.1145/3571255

Matija Pretnar. 2014. Inferring Algebraic Effects. *Log. Methods Comput. Sci.* 10, 3 (2014). https://doi.org/10.2168/LMCS-10(3:21)2014

Zesen Qian. 2022. *Concurrency And Races In Classical Linear Logic*. Ph. D. Dissertation. Aarhus University.

Birthe van den Berg and Tom Schrijvers. 2023. A Framework for Higher-Order Effects & Handlers. *CoRR* abs/2302.01415 (2023). https://doi.org/10.48550/arXiv.2302.01415 arXiv:2302.01415

Birthe van den Berg, Tom Schrijvers, Casper Bach Poulsen, and Nicolas Wu. 2021. Latent Effects for Reusable Language Components. In *Programming Languages and Systems - 19th Asian Symposium, APLAS 2021, Chicago, IL, USA, October 17-18, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 13008)*, Hakjoo Oh (Ed.). Springer, 182–201. https://doi.org/10.1007/978-3-030-89051-3_11

Ningning Xie, Youyou Cong, Kazuki Ikemori, and Daan Leijen. 2022. First-class names for effect handlers. *Proc. ACM Program. Lang.* 6, OOPSLA2 (2022), 30–59. https://doi.org/10.1145/3563289

Yizhou Zhang and Andrew C. Myers. 2019. Abstraction-safe effect handlers via tunneling. *Proc. ACM Program. Lang.* 3, POPL (2019), 5:1–5:29. https://doi.org/10.1145/3290318

Yizhou Zhang, Guido Salvaneschi, and Andrew C. Myers. 2020. Handling bidirectional control flow. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 139:1–139:30. https://doi.org/10.1145/3428207