Abstracting Effect Systems for Algebraic Effect Handlers

TAKUMA YOSHIOKA, Kyoto University, Japan
TARO SEKIYAMA, National Institute of Informatics & SOKENDAI, Japan
ATSUSHI IGARASHI, Kyoto University, Japan

Many effect systems for algebraic effect handlers are designed to guarantee that all invoked effects are handled adequately. However, respective researchers have developed their own effect systems that differ in how to represent the collections of effects that may happen. This situation results in blurring what is required for the representation and manipulation of effect collections in a safe effect system.

In this work, we present a language λ_{EA} equipped with an effect system that abstracts the existing effect systems for algebraic effect handlers. The effect system of λ_{EA} is parameterized over *effect algebras*, which abstract the representation and manipulation of effect collections in safe effect systems. We prove the type-and-effect safety of λ_{EA} by assuming that a given effect algebra meets certain properties called *safety conditions*. As a result, we can obtain the safety properties of a concrete effect system by proving that an effect algebra corresponding to the concrete system meets the safety conditions. We also show that effect algebras meeting the safety conditions are expressive enough to accommodate some existing effect systems, each of which represents effect collections in a different style. Our framework can also differentiate the safety aspects of the effect collections of the existing effect systems. To this end, we extend λ_{EA} and the safety conditions to *lift coercions* and *type-erasure semantics*, propose other effect algebras including ones for which no effect system has been studied in the literature, and compare which effect algebra is safe and which is not for the extensions.

CCS Concepts: • **Theory of computation** → **Type theory**; **Control primitives**; • **Software and its engineering** → **Control structures**; *Functional languages*.

Additional Key Words and Phrases: algebraic effects and handlers, effect algebras, type-and-effect systems

ACM Reference Format:
Takuma Yoshioka, Taro Sekiyama, and Atsushi Igarashi. 2024. Abstracting Effect Systems for Algebraic Effect Handlers. *Proc. ACM Program. Lang.* 8, ICFP, Article 252 (August 2024), 30 pages. https://doi.org/10.1145/3674641

1 Introduction

1.1 Background: Effect Systems for Algebraic Effect Handlers

Algebraic effect handlers [Plotkin and Pretnar 2009, 2013] enable implementing user-defined computational effects, such as mutable states, exceptions, backtracking, and generators, and structuring programs with them in a modular way. A significant aspect of algebraic effect handlers is compositionality. Because of the algebraicity inherited from algebraic effects [Kammar et al. 2013; Plotkin and Power 2003], they allow composing multiple effects easily, unlike some other approaches to user-defined effects, such as monads [Moggi 1991; Wadler 1998]. Another benefit of algebraic effect handlers is to separate the interfaces and implementations of effects. For example, the manipulation of mutable states is expressed by two operations to set a new state and get the current state. While a program manipulates states via these operations, their implementation can be determined

Authors' Contact Information: Takuma Yoshioka, Kyoto University, Kyoto, Japan, yoshioka@fos.kuis.kyoto-u.ac.jp; Taro Sekiyama, National Institute of Informatics & SOKENDAI, Tokyo, Japan, tsekiyama@acm.org; Atsushi Igarashi, Kyoto University, Kyoto, Japan, igarashi@kuis.kyoto-u.ac.jp.

ACM 2475-1421/2024/8-ART252
https://doi.org/10.1145/3674641

dynamically by installing *effect handlers*. This separation of interfaces from implementations allows writing effectful programs in a modular manner.

A key property expected in a statically typed language with algebraic effect handlers is *type-and-effect safety*. In the presence of effect handlers, type safety ensures that the type of an operation is matched with that of its implementation provided by an effect handler. *Effect safety* [Brachthäuser et al. 2020][1] states that every operation call is handled appropriately (i.e., it is performed under an effect handler that provides the called operation with an implementation). Ensuring effect safety is crucial to guarantee the safety of programs as an "unhandled operation" makes programs get stuck.

Several researchers have proposed type-and-effect systems (effect systems for short) to guarantee type-and-effect safety. The effect systems in the literature are classified roughly into two groups according to how they represent collections of effects that programs may invoke. Certain effect systems adapt *sets* to represent such collections [Bauer and Pretnar 2013; Forster et al. 2017; Kammar et al. 2013; Kammar and Pretnar 2017; Saleh et al. 2018; Sekiyama et al. 2020]. Another approach is using *rows* [Biernacki et al. 2019; Hillerström and Lindley 2016; Leijen 2017; Xie et al. 2022], which allow manipulating the collections of effects in a more structured manner. For example, the effect system of Hillerström and Lindley [2016] can represent the presence and absence of effects in rows, and that of Leijen [2017] allows the duplication of effects with the same name in one row.

However, several issues are posed by the current situation that the effect systems in the different styles have been studied independently. First, it blurs what manipulation of effect collections are indispensable to give an effect system. Second, it is unclear what property an effect system requires for effect collections and their manipulation to guarantee effect safety. The lack of clarity in these matters causes the problem that designers of new effect systems grope in the dark for the representations of effect collections, and even if they come up with an appropriate representation, they need to prove the desired properties, such as effect safety, from scratch. The third issue is that, when extending languages with new features, one needs to build the metatheory for each of the representations.

1.2 Our Work

This work aims to reveal the essence of safe effect systems for effect handlers. Because we are interested in the shared nature of such effect systems, we avoid choosing one concrete representation of effect collections. Instead, we provide an effect system that abstracts over the representations of effect collections and can derive concrete effect systems by instantiating them.

More specifically, our effect system is parameterized over constructors and manipulations of effect collections. In general, effect systems for algebraic effect handlers require two kinds of manipulation. One is subeffecting, which overapproximates effects to adjust the effects of different expressions. The other is the removal of effects. An installed effect handler removes the effect it handles and forwards the remaining effects to outer effect handlers. We formulate such manipulation of effect collections required by effect systems as *effect algebras*[2] and ensure that our effect system relies only on the manipulations allowed on them.

However, some effect algebras make the effect system unsound. For instance, the effect system with an effect algebra that allows subeffecting to remove some effects may typecheck unsafe programs (e.g., ones that cause unhandled effects). To prevent the use of such effect algebras, we formalize *safety conditions*, which are sufficient conditions on effect algebras to guarantee

[1]The notion of effect safety itself and its importance have been recognized before the name was coined [Kammar et al. 2013].
[2]The name "effect algebra" has been used to specify algebraic structures found in quantum mechanics [Foulis and Bennett 1994] or to specify an algebraic structure in sequencing effects [Ivaskovic et al. 2020; Katsumata 2014], but we decided to use this name because the present work is far from quantum mechanics and is easily distinguished from the work on sequencing effects (see Section 9 for a comparison between them).

effect safety; we call effect algebras meeting the conditions *safe*. We prove that the effect system instantiated with any safe effect algebra enjoys effect safety as well as type safety—therefore, one can ensure the safety of their effect systems only by showing the safety of the corresponding effect algebras. Furthermore, we also show what kind of unsafe programs each condition excludes.

To show that our framework is expressive enough to capture the essence shared among sound effect systems in the literature, we provide three instances of our effect system. The instances represent effect collections by sets and two styles of rows—called *simple rows* [Hillerström and Lindley 2016] and *scoped rows* [Leijen 2017]. We define effect algebras for these three instances and prove their safety, which means that all the instances satisfy type-and-effect safety. We also show that these instances indeed model the existing effect systems [Hillerström and Lindley 2016; Leijen 2017; Pretnar 2015].

Once it turns out that all the instances satisfy the desired property, what are differences among them? How can they be compared? To answer these questions, we make two changes on the language: introduction of *lift coercions* [Biernacki et al. 2018, 2019] and employment of a *type-erasure semantics* [Biernacki et al. 2019].

Lift coercions are a construct to prevent an operation call from being handled by the closest effect handler, introduced to avoid *accidental handling*, that is, unintended handling of operation calls. To reason about the effect of lift coercions soundly, effect collections should be able to express how many effect handlers need to be installed on effectful computation. Effect collections represented by sets or simple rows cannot express it because they collapse multiple occurrences of the same effect into one. Thus, the instances with sets or simple rows result in being unsound. By contrast, scoped rows can encode the number of necessary effect handlers due to the ability to duplicate effects. To enhance the importance of being able to represent the number of necessary effect handlers in the presence of lift coercions, we propose a new instance where effect collections are represented by *multisets*. Because multisets record the multiplicities of the elements they contain, it is expected that the instance with multisets, as well as that with scoped rows, satisfies type-and-effect safety even in the presence of lift coercions. We show that it is the case by providing an additional safety condition for lift coercions, proving that any instance of the effect system enjoys type-and-effect safety if it meets the new safety condition as well as the original ones, and showing that the effect algebra for scoped rows and the one for multisets meet both the additional and original safety conditions.

The second change is to adopt a type-erasure semantics, which differs from the original semantics in the effect comparison in the dynamic search for effect handlers: the original semantics takes into account what type parameters effects accompany to identify effects, while the type-erasure semantics does not. This nature of type-erasure semantics makes the instances with sets and multisets unsound because it is in conflict with the nature of sets and multisets that the order of elements is ignored. The row-based instances can be adapted to the type-erasure semantics by restricting the commutativity in rows. Even for sets and multisets, we can give type-and-effect safe instances based on them if we admit restriction on swapping elements.

These extensions demonstrate the benefit of the abstraction brought by effect algebras: it enables a formal comparison among different forms of effect collections. Specifically, the abstraction clarifies what effect collections make the effect system *unsound*. As shown in Section 7, effect algebras where the composition of effect collections is idempotent (resp. commutative) allow typechecking programs causing unhandled effects in adopting lift coercions (resp. the type-erasure semantics). This kind of formal comparison helps one find how influential the different representations of effect collections are on effect safety.

The contributions of this work are summarized as follows.

- We introduce an abstract effect system for algebraic effect handlers. It abstracts over effect algebras, which characterize the representation and manipulation of effect collections in the effect system.
- We define safety conditions that enforce the effect manipulation allowed by effect algebras to be safe.
- We prove that effect systems instantiated by safe effect algebras are type-and-effect safe.
- We extend the effect system to lift coercions and type-erasure semantics, define an additional safety condition for each of them, and prove that an instance of each extension is type-and-effect safe provided that the effect algebra in the instance meets the specified conditions.
- We give four examples of safe effect algebras and their variants for the type-erasure semantics.

The effect system presented in this paper supposes *deep* effect handlers, but we also have adapted the system to *shallow* effect handlers [Kammar et al. 2013]; readers interested in the formulation for shallow effect handlers are referred to the supplementary material.

The rest of this paper is organized as follows. Section 2 reviews algebraic effect handlers and the existing effect systems, and overviews our approach. Section 3 introduces our type-and-effect language and effect algebras. We also show the instances based on sets and rows as their examples. Section 4 presents our calculus with the abstract effect system. Section 5 states safety conditions, explains their necessities, and proves the type-and-effect safety of the calculus under the safe conditions. Section 6 shows that some existing effect systems can be modeled soundly by the corresponding instances of our calculus. Section 7 extends our language and the safety conditions to lift coercions and type-erasure semantics and Section 8 compares the effect algebras given in the paper. Section 9 describes additional related works and Section 10 concludes this paper with future works. This paper only states certain key properties. All the auxiliary lemmas, proofs, and full definition are given in the supplementary material.

2 Overview

This section reviews algebraic effect handlers and the existing effect systems for them, and provides an overview of our approach to abstracting the effect systems.

2.1 Review: Algebraic Effects and Handlers

Algebraic effect handlers are a means to implement user-defined effects in a modular way. The interface of effects consists of operations, and their behavior is specified by effect handlers.

For example, consider the following program that uses effect Choice (this paper uses ML-like syntax to describe programs):

$$\textbf{effect}\ \mathsf{Choice} :: \{\mathsf{decide} : \mathsf{Unit} \Rightarrow \mathsf{Bool}\}$$

$$\textbf{handle}_{\mathsf{Choice}}$$

$$\quad \textbf{let}\ x = \textbf{if}\ \mathsf{decide}\ ()\ \textbf{then}\ 20\ \textbf{else}\ 10\ \textbf{in let}\ y = \textbf{if}\ \mathsf{decide}\ ()\ \textbf{then}\ 5\ \textbf{else}\ 0\ \textbf{in}\ x - y$$

$$\textbf{with}\ \{\ \textbf{return}\ z \mapsto z\} \uplus \{\mathsf{decide}\ z\ k \mapsto \mathtt{max}\ (k\,\mathsf{true}, k\,\mathsf{false})\}$$

The first line declares *effect label* Choice with only one operation decide. As indicated by its type, decide takes the unit value and returns a Boolean. The program invokes the operation in the third line, determines numbers x and y depending on the results, and returns $x - y$ finally. To install an effect handler, we use the handling construct **handle**–**with**.

In general, an expression $\textbf{handle}_l\ e\ \textbf{with}\ h$ means that an expression e is executed under effect handler h, which interprets the operations of effect label l invoked by e; we call e a *handled expression*. An effect handler consists of one return clause and possibly several operation clauses. A return

clause $\{\,\mathbf{return}\ x \mapsto e_r\}$, which corresponds to $\{\,\mathbf{return}\ z \mapsto z\}$ in the example, is executed when a handled expression evaluates to a value, which the body e_r references by x. An operation clause takes the form $\{\mathsf{op}\ x\ k \mapsto e\}$, which determines the implementation of operation op. When an operation op is called with an argument v under an effect handler with operation clause $\{\mathsf{op}\ x\ k \mapsto e\}$, the reduction proceeds as follows. First, the remaining computation from the point of the operation call up to the **handle**–**with** construct installing the effect handler is captured; such a computation is called a *delimited continuation*. Then, the body e of the corresponding operation clause is executed by passing the argument v as x and the delimited continuation as k.

In the example, the delimited continuation for the first call to decide is

$$\begin{aligned}&\mathbf{handle}_{\mathsf{Choice}}\\ &\quad\mathbf{let}\ x = \mathbf{if}\ \square\ \mathbf{then}\ 20\ \mathbf{else}\ 10\ \mathbf{in}\ \mathbf{let}\ y = \mathbf{if}\ \mathsf{decide}\ ()\ \mathbf{then}\ 5\ \mathbf{else}\ 0\ \mathbf{in}\ x - y\\ &\mathbf{with}\ \{\,\mathbf{return}\ z \mapsto z\} \uplus \{\mathsf{decide}\ z\ k \mapsto \mathtt{max}\ (k\,\mathsf{true}, k\,\mathsf{false})\},\end{aligned}$$

where $\square$ denotes a hole. The functional form v_1 of this delimited continuation is bound to variable k in the operation clause of decide, and the program evaluates to $\mathtt{max}\ (v_1\,\mathsf{true}, v_1\,\mathsf{false})$. The function application $v_1\,\mathsf{true}$ fills the hole of the delimited continuation with argument true. Thus, it reduces

$$\begin{aligned}&\mathbf{handle}_{\mathsf{Choice}}\\ &\quad\mathbf{let}\ x = \mathbf{if}\ \boxed{\mathsf{true}}\ \mathbf{then}\ 20\ \mathbf{else}\ 10\ \mathbf{in}\ \mathbf{let}\ y = \mathbf{if}\ \mathsf{decide}\ ()\ \mathbf{then}\ 5\ \mathbf{else}\ 0\ \mathbf{in}\ x - y\\ &\mathbf{with}\ \{\,\mathbf{return}\ z \mapsto z\} \uplus \{\mathsf{decide}\ z\ k \mapsto \mathtt{max}\ (k\,\mathsf{true}, k\,\mathsf{false})\},\end{aligned}$$

where $\boxed{\mathsf{true}}$ comes from the argument. Then, it substitutes 20 for x, and then calls decide again. The operation clause invokes the delimited continuation v_2 captured by the second call with arguments true and false. The applications $v_2\,\mathsf{true}$ and $v_2\,\mathsf{false}$ choose 5 and 0 as y and return the results of $20-5$ and $20-0$ (that is, 15 and 20), respectively. Then, the operation clause return $\mathtt{max}\ (15, 20)$ as the result of $v_1\,\mathsf{true}$. Similarly, the function application $v_1\,\mathsf{false}$ results in $\mathtt{max}\ (5, 10)$. Thus, the entire program evaluates to $\mathtt{max}\ (\mathtt{max}\ (15, 20), \mathtt{max}\ (5, 10))$ and then to 20 finally.

While the operation clause in the above example uses captured continuations, effect handlers can also discard them. Using this ability, we can implement exception handling, as the following program that divides x by y if y is nonzero:

$$\begin{aligned}&\mathbf{effect}\ \mathsf{Exc} :: \{\mathsf{raise} : \mathsf{Unit} \Rightarrow \mathsf{Empty}\}\\ &\mathbf{let}\ g = \lambda x : \mathsf{Int}.\lambda y : \mathsf{Int}.\ \mathbf{handle}_{\mathsf{Exc}}\ (\mathbf{if}\ y = 0\ \mathbf{then}\ \mathsf{raise}\ ()\ \mathbf{else}\ x/y)\\ &\qquad\qquad\qquad\qquad\mathbf{with}\ \{\,\mathbf{return}\ z \mapsto \mathtt{int\_to\_string}\ z\} \uplus \{\mathsf{raise}\ p\ k \mapsto \mathtt{"divided\ by\ 0"}\ \}\end{aligned}$$

In this example, Exc is an effect label consisting of one operation raise with type $\mathsf{Unit} \Rightarrow \mathsf{Empty}$. Here, Empty is a type having no inhabitant, and we assume that an expression of this type can be regarded as that of any type. The return clause of the effect handler means that, when the handled expression evaluates to an integer, the handling construct returns its string version. Because the operation clause for raise discards the continuations, the handling construct returns the string `"divided by 0"` immediately once raise is called. Therefore, the operation call and effect handling in this example correspond to excepting raising and handling, respectively.

2.2 Effect Systems for Algebraic Effects and Handlers

This section briefly explains a role of effect systems for algebraic effect handlers and summarizes the existing systems.

2.2.1 A Role of Effect Systems. A property ensured by many effect systems in the literature is *effect safety*, which means that there is no unhandled operation. A simple example that breaks effect safety is op v, which just invokes operation op. Because no effect handler for op is given—thus, there is no way to interpret it—the program gets stuck. However, even if an operation call is enclosed by handling constructs, effect safety can be broken. For example, consider the following program:

effect Exc :: {raise : Unit ⇒ Empty}

effect State :: {set : Int ⇒ Unit, get : Unit ⇒ Int}

let $g = \lambda x : \mathsf{Int}.$ **handle**$_{\mathsf{Exc}}$ (**if** $x = 0$ **then** raise () **else** (**let** $y =$ get ()/x **in** set y; y))

with { **return** $z \mapsto$ `int_to_string` z} ⊎ {raise $p\,k \mapsto$ `"divided by 0"` }

g 42 2

The effect label State is for mutable state, providing two operations set and get to update and get the current values in the state. The function g divides the current value of the state (returned by get) by x, sets the result to the state, and returns it if x is nonzero. All the operation calls in the application g 42 2 at the last line are performed under the effect handler, but the call to get is not handled. Hence, this example is not effect safe.

In general, the effect systems enjoying effect safety need to track which effect each expression may invoke and which effect an effect handler targets. However, there are choices to represent the effects caused by expressions. Thus far, mainly two styles of formalization of effect systems have been studied: one is based on *sets* [Bauer and Pretnar 2013; Forster et al. 2017; Kammar et al. 2013; Kammar and Pretnar 2017; Saleh et al. 2018; Sekiyama et al. 2020], and the other is based on *rows* [Biernacki et al. 2019; Hillerström and Lindley 2016; Leijen 2017; Xie et al. 2022].

2.2.2 Set-Based Effect Systems. Set-based effect systems assign to an expression a set of effect labels that the expression may invoke. For example, they assign to an operation call a set that includes the effect label of the called operation. This is formalized as follows, where typing judgment $\Gamma \vdash e : A \mid s$ means that expression e is of type A under typing context Γ and may invoke effects in set s:

$$\frac{\text{Operation op} : A \Rightarrow B \text{ belongs to effect } l \quad \Gamma \vdash v : A \mid \{\}}{\Gamma \vdash \text{op}\ v : B \mid \{l\}}$$

Subeffecting, which is supported to unify the effects of multiple expressions (such as branches in conditional expressions), is implemented by allowing the expansion of sets:

$$\frac{\Gamma \vdash e : A \mid s \quad s \subseteq s'}{\Gamma \vdash e : A \mid s'}$$

In the presence of algebraic effect handlers, sets not only expand but also may shrink. Such manipulation is performed in handling constructs:

$$\frac{\Gamma \vdash e : A \mid s \qquad \{l\} \cup s' = s \qquad \cdots}{\Gamma \vdash \mathbf{handle}_l\, e\, \mathbf{with}\, h : B \mid s'}$$

where the omitted premise states that h is a handler for effect l, translating a computation of type A to type B. This inference rule is matched with the behavior of the handling constructs because they can make handled effects l "unobservable." The set-based effect systems defined in such a way can soundly overapproximate the observable effects of programs and guarantee the effect safety of expressions to which the empty set can be assigned.

For instance, consider the example in Section 2.2.1. A set-based effect system would assign the set {Exc, State} to the handled expression **if** $x = 0$ **then** raise () **else** (**let** $y =$ get ()/x **in** set y; y)

because it calls operation raise of Exc or get and set of State. Because this expression is only placed under the effect handler for Exc, the entire program $g\,42\,2$ could have set $\{\mathsf{State}\}$. As this set indicates that effect State may not be handled—and it *is not* actually—the effect system would conclude that the program may not be effect safe. If the program were wrapped by a handling construct with an effect handler for State, the empty set could be assigned to it; then, we could conclude that the program is effect safe.

2.2.3 Row-Based Effect Systems. Rows express collections of effect labels in a more structured way. In a monomorphic setting, they are just sequences of effect labels, as $\langle l_1, \ldots, l_n \rangle$, which is the row consisting only of labels $l_1, \ldots, l_n$. Rows are identified up to the reordering of labels. For example, $\langle l_1, l_2 \rangle$ equals $\langle l_2, l_1 \rangle$.[3]

Rows are often adapted in languages with effect polymorphism [Biernacki et al. 2019; Hillerström and Lindley 2016; Leijen 2017]. In such languages, rows are allowed to end with effect variables ρ, such as $\langle l_1, \ldots, l_n, \rho \rangle$, which means that an expression may invoke effects $l_1, \ldots, l_n$ as well as those in an instance of effect variable ρ. This extension enables abstraction over rows by universally quantifying effect variables. For example, consider function filtered_set, which, given an integer list and a function f from integers to Booleans, filters out the elements of the list using function f and then calls operation set of effect State on the remaining elements. Assume that the type of functions from type A to type B with effects in row r is described as $A \to_r B$. Then, filtered_set can be given type $\forall\rho.(\mathsf{Int\ List} \times (\mathsf{Int} \to_\rho \mathsf{Bool})) \to_{\langle \mathsf{State}, \rho \rangle} \mathsf{Unit}$. By instantiating ρ with $\langle l_1, \ldots, l_n \rangle$, this type can express that, when passed a function f that may cause effects $l_1, \cdots, l_n$, filtered_set may also cause them via the application of f.

Inference rules of the row-based effect systems are similar to those of set-based ones, except that subeffecting allows enlarging rows only when they do not end with effect variables (such rows are called *closed*, while rows ending with effect variables are *open* [Hillerström and Lindley 2016]):

$$\frac{\Gamma \vdash e : A \mid \langle l_1, \ldots, l_n \rangle}{\Gamma \vdash e : A \mid \langle l_1, \ldots, l_n, r \rangle}$$

Rows shrink in handling constructs where handled effects are removed:

$$\frac{\Gamma \vdash e : A \mid r \qquad \langle l, r' \rangle = r \qquad \cdots}{\Gamma \vdash \mathbf{handle}_l\, e\, \mathbf{with}\, h : B \mid r'}$$

Similar to set-based ones, the row-based effect systems also ensure the effect safety of expressions to which the empty row $\langle\rangle$ can be assigned. The reasoning about the example in Section 2.2.1 can be done similarly to the case with simple rows.

These are the common core of the row-based effect systems, but they can be further classified into two groups depending on the formalism of rows. One is simple rows [Hillerström and Lindley 2016], where each label can appear at most once in one row. In this formalism, any l_i in row $\langle l_1, \ldots, l_n \rangle$ must be different from l_j for any $j \neq i$. The other is scoped rows [Leijen 2017], where the same label can appear in one row multiple times. Therefore, given a scoped row $\langle l_1, \ldots, l_n \rangle$, any l_i is allowed to be equivalent to some l_j, unlike simple rows.

2.3 Our Work: Abstracting Effect Systems

All effect systems based on sets, simple rows, or scoped rows exploit the structures of the respective representations to augment and shrink the information about effects. However, it is not clear which part of these structures essentially contributes to type-and-effect safety. To reveal it, we provide an abstract model of effect collections and their manipulation and give an effect system relying only

[3]The label reordering might need to be restricted if effect labels are parameterized over, e.g., types, as discussed in Section 7.2.

on the abstract model. We also state sufficient conditions on the abstract model to guarantee the safety of our effect system. With the effect system depending only on the abstract nature of effect collections, we reveal the essence of safe effect systems for algebraic effect handlers.

We abstract the effect collections and manipulation in the effect systems for algebraic effect handlers by an *effect algebra*, which consists of an equivalence relation $\sim$ and a partial binary operation $\odot$[4], which mean the equivalence over effects and effect concatenation, respectively (these notations come from Morris and McKinna [2019]). For example, $\varepsilon_1 \odot \varepsilon_2 \sim \varepsilon_3$ intends to state that the concatenation of effects ε_1 and ε_2 is equal to ε_3. Our effect system is parameterized by effect algebras and manipulate effect collections only through the operation $\odot$ of a given effect algebra; hence, it does not suppose any concrete effect manipulation.

To abstract over the representations of effect collections, our effect system assumes two effect constructors. One is $\mathbb{0}$, which represents the empty collection and corresponds to the empty set and row in the set- and row-based effect systems, respectively. The other constructor is $(l)^\uparrow$, which constructs the effect collection composed only of effect label l.

With these abstractions, the inference rules that manipulate effect collections—i.e., those for operation calls, subeffecting, and handling constructs—are given as follows (here, we give only informal rules; the formal rule corresponding to each informal one T_[*RuleName*0] is found in Figure 5, named T_[*RuleName*] there.):

$$\frac{\text{Operation op} : A \Rightarrow B \text{ belongs to effect } l \quad \Gamma \vdash v : A \mid \mathbb{0}}{\Gamma \vdash \text{op}\ v : B \mid (l)^\uparrow}\ \text{T\_Op0}$$

$$\frac{\Gamma \vdash e : A \mid \varepsilon \qquad \varepsilon \odot \varepsilon_0 \sim \varepsilon'}{\Gamma \vdash e : A \mid \varepsilon'}\ \text{T\_Sub0} \qquad \frac{\Gamma \vdash e : A \mid \varepsilon \qquad (l)^\uparrow \odot \varepsilon' \sim \varepsilon \qquad \cdots}{\Gamma \vdash \mathbf{handle}_l\, e\, \mathbf{with}\, h : B \mid \varepsilon'}\ \text{T\_Handle0}$$

The rule T_Op0 for operation calls simply injects the corresponding effect label into the effect collection. The rule T_Sub0 allows subsumption with subeffecting, which expands the effect ε of an expression to ε' by appending some effects ε_0. The rule T_Handle0 for handling constructs means that, if a handled expression may invoke effects in ε, only the remaining ε' of excluding the handled effect l from ε is observable from the outer context.

It is noteworthy that the above usage of effect algebras pays attention to the order of effects appearing in effect collections. Specifically, the subsumption rule only allows appending extra effects ε_0 and does not allow prepending them, and the rule for handling constructs removes only the handled effect label that occurs first in ε. This mirrors the nature of the effect handling that an operation call is handled by the effect handler closest to the call. The importance of considering the order of effects is confirmed in, e.g., adopting a type-erasure semantics: as discussed in Section 7.2, our effect system becomes unsound under the type-erasure semantics if a given effect algebra is equipped with commutative $\odot$, which makes the effect system *in*sensitive to the order of effects.

While effect algebras are expressive enough to represent the manipulation of effect collections, some effect algebras make the effect system unsafe. For example, consider an effect algebra where $(l)^\uparrow \odot \varepsilon \sim \mathbb{0}$ holds. Given an operation op of the effect label l, the subsumption rule allows coercing the effect $(l)^\uparrow$ of an operation call op v to $\mathbb{0}$. It means that the effect system can state that op v invokes no unhandled operation, so the effect system with such an effect algebra is unsafe.

To prevent the use of such effect algebras, we establish conditions on effect algebras; we call them *safety conditions* and also call effect algebras meeting them *safe*. We prove that, given a safe effect algebra, our effect system satisfies type and effect safety. We also demonstrate the expressibility

[4]We pose certain requirements on $\sim$ and $\odot$ for effect safety in Section 3.

f, g, x, y, z, p, k (variables) $\quad \alpha, \beta, \gamma, \tau, \iota, \rho$ (typelike variables) $\quad$ op (operation names)
$l \in \mathrm{dom}(\Sigma_{\mathrm{lab}})$ (label names) $\quad \mathcal{F} \in \mathrm{dom}(\Sigma_{\mathrm{eff}})$ (effect constructors) $\quad C \in \mathrm{dom}(\Sigma_{\mathrm{lab}}) \cup \mathrm{dom}(\Sigma_{\mathrm{eff}})$

$$
\begin{array}{rcll}
K & ::= & \mathbf{Typ} \mid \mathbf{Lab} \mid \mathbf{Eff} & \text{(kinds)} \\
A, B, C & ::= & \tau \mid A \to_{\varepsilon} B \mid \forall \alpha : K.A^{\varepsilon} & \text{(types)} \\
\varepsilon & ::= & \rho \mid \mathcal{F}\, \boldsymbol{S}^{I} & \text{(effects)} \\
\sigma & ::= & \{\} \mid \sigma \uplus \{\mathsf{op} : \forall \boldsymbol{\beta}^{\mathcal{J}} : \boldsymbol{K}^{\mathcal{J}}.A \Rightarrow B\} & \text{(operation signatures)} \\
\Gamma & ::= & \emptyset \mid \Gamma, x : A \mid \Gamma, \alpha : K & \text{(typing contexts)}
\end{array}
\qquad
\begin{array}{rcll}
S, T & ::= & A \mid L \mid \varepsilon & \text{(typelikes)} \\
L & ::= & \iota \mid l\, \boldsymbol{S}^{I} & \text{(labels)} \\
\Xi & ::= & \emptyset \mid \Xi, l :: \forall \boldsymbol{\alpha}^{I} : \boldsymbol{K}^{I}.\sigma & \text{(effect contexts)}
\end{array}
$$

Fig. 1. Typelike syntax over an label signature Σ_{lab} and an effect signature Σ_{eff}.

of our framework by providing effect algebras for sets, simple rows, and scoped rows from the literature, as well as one for multisets, which are a new representation of effect collections.

3 Abstracting Effects

This section introduces the core notions of our effect system: effect algebras, an abstract model of effect collections and their manipulations. Because we aim at a formal effect system, we need to decide the syntactic representation of effect collections manipulated by the effect system. However, relying on specific representations prevents accommodating a variety of effect systems in the literature. To address this problem, we parameterize our effect system over the representations of effect collections and assume that the interface of their constructs is given by an *effect signature*.

Throughout this paper, we use the notation $\boldsymbol{\alpha}^{I}$ for a finite sequence $\alpha_0, \ldots, \alpha_n$ with an index set $I = \{0, \ldots, n\}$, where α is any metavariable. We also write $\{\boldsymbol{\alpha}^{I}\}$ for the set consisting of the elements of $\boldsymbol{\alpha}^{I}$. Index sets are designated by I, $\mathcal{J}$, and N. We omit index sets and write $\boldsymbol{\alpha}$ simply when they are not important (e.g., all the sequences of interest have the same length).

3.1 Syntax

We start by defining *label* and *effect signatures*, which specify available *label names* (the names of effects) and effect collection constructors as well as their kinds, respectively. We then introduce the syntax of types, effect labels, and effect collections using a given label and effect signature. Kinds, ranged by K, are **Typ** for types, **Lab** for effect labels, or **Eff** for effect collections.

Definition 3.1 (Label Signatures)**.** *Given a set S of label names, a label signature Σ_{lab} is a functional relation whose domain $\mathrm{dom}(\Sigma_{\mathrm{lab}})$ is S. The codomain of Σ_{lab} is the set of functional kinds of the form $\Pi_{i \in I} K_i \to \mathbf{Lab}$ for some I and $K_i^{i \in I}$ (if $I = \emptyset$, it means* **Lab** *simply).*

Definition 3.2 (Effect Signatures)**.** *Given a set S of effect constructors, an effect signature Σ_{eff} is a functional relation whose domain $\mathrm{dom}(\Sigma_{\mathrm{eff}})$ is S. The codomain of Σ_{eff} is the set of functional kinds of the form $\Pi_{i \in I} K_i \to \mathbf{Eff}$ for some I and $K_i^{i \in I}$. (if $I = \emptyset$, it means* **Eff** *simply).*

Definition 3.3 (Signatures)**.** *A signature Σ is the union of a label signature and an effect signature (note that they are disjoint).*

Hereinafter, the notation $\Pi \boldsymbol{K}^{I} \to K$ (or simply, $\Pi \boldsymbol{K} \to K$) denotes an abbreviation of $\Pi_{i \in I} K_i \to K$, and $C : \Pi \boldsymbol{K} \to K$ denotes the pair $\langle C, \Pi \boldsymbol{K} \to K \rangle$ for label name or effect constructor C.

Example 3.4 (Label Signatures of Exc and State)**.** The label signature for label names Exc and State used in Section 2.2.1 are given as $\{\mathsf{Exc} : \mathbf{Lab},\ \mathsf{State} : \mathbf{Lab}\}$. The label State in Section 2.2.1 assumes the values of state to be integers, but, if one wants to parameterize label State over the types of the state values, the signature of State changes to $\mathsf{State} : \mathbf{Typ} \to \mathbf{Lab}$. This signature

indicates that State can take a type argument A that represents the type of the state values. We call parameterized label names, as State of kind $\mathbf{Typ} \to \mathbf{Lab}$, *parametric effects*, which facilitate the reuse of program components as explained later.

The following is an effect signature for *effect sets*, effect collections implemented by sets.

Example 3.5 (Effect Signature of Effect Sets)**.** The effect signature $\Sigma_{\mathrm{eff}}^{\mathrm{Set}}$ of effect sets consists of the pairs $\{\} : \mathbf{Eff}$ (for the empty set), $\{-\} : \mathbf{Lab} \to \mathbf{Eff}$ (for singleton sets), and $- \underline{\cup} - : \mathbf{Eff} \times \mathbf{Eff} \to \mathbf{Eff}$ (for set unions).[5]

Given a signature $\Sigma = \Sigma_{\mathrm{lab}} \uplus \Sigma_{\mathrm{eff}}$, the syntax of types, ranged over by A, B, and C, effect labels (or labels for short), ranged over by L, and effect collections (or effects for short), ranged over by ε, is defined as in Figure 1. This work allows three kinds of polymorphism, that is, type, label, and effect polymorphism. To simplify their presentation, we introduce a syntactic category that unifies types, labels, and effects; we call its entities *typelikes* [Biernacki et al. 2019], which are ranged over by S and T. Typelikes are classified into types, labels, and effects using the kind system presented in Section 3.2. We use τ, ι, and ρ to designate type, label, and effect variables (i.e., typelike variables with kind **Typ**, **Eff**, and **Lab**), respectively, and α, β, and γ in a general context.

Types consist of: type variables; function types $A \to_{\varepsilon} B$, which represent functions from type A to B with effect ε; and polymorphic types $\forall \alpha : K.A^{\varepsilon}$, which represent (suspended) computation with effect ε abstracting over typelikes of kind K. We omit base types such as Int for simplification, but assume them and some operations on them (such as + for integers) in giving examples.

A label is a label variable or a label name, ranged over by l, possibly with type arguments. For example, consider State : $\mathbf{Typ} \to \mathbf{Lab}$ given in Example 3.4. A label State A represents mutable state possessing the values of the type A. We can implement State A using a state-passing effect handler, which abstracts over type arguments A [Leijen 2017]. Thus, the effect handler can be reused for different type arguments.

Effects are composed of effect variables and effect constructors, ranged over by $\mathcal{F}$, given by Σ_{eff}. As label names, effect constructors can take typelikes as arguments. For example, effect set $\{\mathsf{Exc}\}$ is represented by $\mathcal{F}$ Exc where $\mathcal{F}$ is the constructor $\{-\}$ for singleton sets.

Effect contexts, ranged over by Ξ, are finite sequences of declarations of effect label names. Each label name l is associated with a type scheme of the form $\forall \boldsymbol{\alpha} : \boldsymbol{K}.\sigma$, where σ is an operation signature parameterized over typelike variables $\boldsymbol{\alpha}$ of kinds $\boldsymbol{K}$. In general, the functional kind $\Pi \boldsymbol{K'} \to \mathbf{Lab}$ of l in Σ_{lab} needs to be consistent with the kind of the type scheme, that is, $\boldsymbol{K'} = \boldsymbol{K}$; we will formalize this requirement in Section 5.2. An *operation signature* is a set of pairs of an operation name op and its type $\forall \boldsymbol{\beta} : \boldsymbol{K}.A \Rightarrow B$. Here, A and B are the argument and return types of the operation, respectively, and they are parameterized over $\boldsymbol{\beta}$ of kinds $\boldsymbol{K}$. Namely, not only effect labels but also operations can be parametric. For example, the effect context for nonparametric effect labels Exc and State in Section 2.1 is given as

$$\mathsf{Exc} :: \{\mathsf{raise} : \mathsf{Unit} \Rightarrow \mathsf{Empty}\}, \mathsf{State} :: \{\mathsf{set} : \mathsf{Int} \Rightarrow \mathsf{Unit}, \mathsf{get} : \mathsf{Unit} \Rightarrow \mathsf{Int}\} .$$

If one wants to parameterize label State over the types of the state values, and operation raise of label Exc over return types (because it returns no value actually), the effect context can change to

$$\mathsf{Exc} :: \{\mathsf{raise} : \forall \alpha : \mathbf{Typ}.\mathsf{Unit} \Rightarrow \alpha\}, \mathsf{State} :: \forall \alpha : \mathbf{Typ}.\{\mathsf{set} : \alpha \Rightarrow \mathsf{Unit}, \mathsf{get} : \mathsf{Unit} \Rightarrow \alpha\} .$$

A difference between parametric effects and operations is that, while effect handlers for parametric effects can be typechecked depending on given type arguments, ones for parametric operations must abstract over type arguments. See Sekiyama and Igarashi [2019] for detail.

[5] We use "–" for unnamed arguments. Multiple occurrences of "–" are distinguished from each other; the i-th occurrence from the left represents the i-th argument.

Kinding $\boxed{\Gamma \vdash S : K}$ $\boxed{\Gamma \vdash \boldsymbol{S}^I : \boldsymbol{K}^I}$ $\iff \forall i \in I.(\Gamma \vdash S_i : K_i)$

$$\frac{\vdash \Gamma \quad \alpha : K \in \Gamma}{\Gamma \vdash \alpha : K}\ \text{K\_Var} \qquad \frac{\vdash \Gamma \quad C : \Pi \boldsymbol{K} \to K_0 \in \Sigma \quad \Gamma \vdash \boldsymbol{S} : \boldsymbol{K}}{\Gamma \vdash C\,\boldsymbol{S} : K_0}\ \text{K\_Cons}$$

$$\frac{\Gamma \vdash A : \mathbf{Typ} \quad \Gamma \vdash \varepsilon : \mathbf{Eff} \quad \Gamma \vdash B : \mathbf{Typ}}{\Gamma \vdash A \to_\varepsilon B : \mathbf{Typ}}\ \text{K\_Fun} \qquad \frac{\Gamma, \alpha : K \vdash A : \mathbf{Typ} \quad \Gamma, \alpha : K \vdash \varepsilon : \mathbf{Eff}}{\Gamma \vdash \forall \alpha : K.A^\varepsilon : \mathbf{Typ}}\ \text{K\_Poly}$$

Fig. 2. Kinding rules.

Typing contexts, ranged over by Γ, are finite sequences of bindings of the form $x : A$ or $\alpha : K$.

3.2 Kind System

We show our kind system in Figure 2. We omit the rules for well-formedness of typing contexts because they are defined as usual [Kawamata et al. 2024; Sekiyama et al. 2020]. The rules other than K_Cons are standard or straightforward. When signature Σ assigns $\Pi \boldsymbol{K} \to K_0$ to label name or effect constructor C, and typelike arguments $\boldsymbol{S}$ are of the kinds $\boldsymbol{K}$, respectively, the rule K_Cons assigns kind K_0 to the typelike $C\,\boldsymbol{S}$.

3.3 Effect Algebras

Now, we define effect algebras. In short, an effect algebra provides an effect signature Σ_{eff}, a partial monoid on effects defined over Σ_{eff}, and a function $(-)^\uparrow$ that injects labels to effects, but more formally, it also requires that each involved operation preserve well-formedness and kind-aware typelike substitution make a homomorphism. In what follows, we denote the sets of types, effect labels, and effect collections over a signature Σ by $\mathbf{Typ}(\Sigma)$, $\mathbf{Lab}(\Sigma)$, and $\mathbf{Eff}(\Sigma)$, respectively (we refer to the set of entities at kind K by $K(\Sigma)$).

Definition 3.6 (Well-Formedness-Preserving Functions)**.** *Given a signature Σ, a (possibly partial) function $f \in K_i(\Sigma)^{i \in \{1,\ldots,n\}} \rightharpoonup K(\Sigma)$ preserves well-formedness if*

$$\forall \Gamma, S_1, \ldots, S_n.\, \Gamma \vdash S_1 : K_1 \wedge \cdots \wedge \Gamma \vdash S_n : K_n \wedge f(S_1, \ldots, S_n) \in K(\Sigma) \implies \Gamma \vdash f(S_1, \ldots, S_n) : K\,.$$

Similarly, $f \in K(\Sigma)$ preserves well-formedness if $\Gamma \vdash f : K$ for any Γ.

In what follows, we write $\alpha \mapsto T \vdash \boldsymbol{S} : K_0$ for a quadruple $\langle \alpha, T, \boldsymbol{S}, K_0 \rangle$ such that $\exists \Gamma_1, K, \Gamma_2.\, (\forall S_0 \in \boldsymbol{S}.\, \Gamma_1, \alpha : K, \Gamma_2 \vdash S_0 : K_0) \wedge \Gamma_1 \vdash T : K$; it means that typelikes $\boldsymbol{S}$ are well formed at kind K_0 and substituting typelike T for typelike variable α in $\boldsymbol{S}$ preserves their well-formedness.

Definition 3.7 (Effect algebras)**.** *Given a label signature Σ_{lab}, an effect algebra is a quintuple $\langle \Sigma_{\text{eff}}, \odot, \mathbb{0}, (-)^\uparrow, \sim \rangle$ satisfying the following, where we let $\Sigma = \Sigma_{\text{lab}} \uplus \Sigma_{\text{eff}}$.*

- *$\odot \in \mathbf{Eff}(\Sigma) \times \mathbf{Eff}(\Sigma) \rightharpoonup \mathbf{Eff}(\Sigma)$, $\mathbb{0} \in \mathbf{Eff}(\Sigma)$, and $(-)^\uparrow \in \mathbf{Lab}(\Sigma) \to \mathbf{Eff}(\Sigma)$ preserve well-formedness. Furthermore, $\sim$ is an equivalence relation on $\mathbf{Eff}(\Sigma)$ and preserves well-formedness, that is, $\forall \varepsilon_1, \varepsilon_2.\, \varepsilon_1 \sim \varepsilon_2 \implies (\forall \Gamma.\, \Gamma \vdash \varepsilon_1 : \mathbf{Eff} \iff \Gamma \vdash \varepsilon_2 : \mathbf{Eff})$.*
- *$\langle \mathbf{Eff}(\Sigma), \odot, \mathbb{0} \rangle$ is a partial monoid under $\sim$, that is, the following holds:*
  - *$\forall \varepsilon \in \mathbf{Eff}(\Sigma).\, \varepsilon \odot \mathbb{0} \sim \varepsilon \wedge \mathbb{0} \odot \varepsilon \sim \varepsilon$; and*
  - *$\forall \varepsilon_1, \varepsilon_2, \varepsilon_3 \in \mathbf{Eff}(\Sigma)$.*
    $(\varepsilon_1 \odot \varepsilon_2) \odot \varepsilon_3 \in \mathbf{Eff}(\Sigma) \vee \varepsilon_1 \odot (\varepsilon_2 \odot \varepsilon_3) \in \mathbf{Eff}(\Sigma) \implies (\varepsilon_1 \odot \varepsilon_2) \odot \varepsilon_3 \sim \varepsilon_1 \odot (\varepsilon_2 \odot \varepsilon_3)$.
- *Typelike substitution respecting well-formedness is a homomorphism for $\odot$, $(-)^\uparrow$, and $\sim$, that is, the following holds:*
  - *$\forall \alpha, S, \varepsilon_1, \varepsilon_2.\, \alpha \mapsto S \vdash \varepsilon_1, \varepsilon_2 : \mathbf{Eff} \wedge \varepsilon_1 \odot \varepsilon_2 \in \mathbf{Eff}(\Sigma) \implies (\varepsilon_1 \odot \varepsilon_2)[S/\alpha] = \varepsilon_1[S/\alpha] \odot \varepsilon_2[S/\alpha]$;*

– $\forall \alpha, S, L.\ \alpha \mapsto S \vdash L : \mathbf{Lab} \implies (L)^{\uparrow}[S/\alpha] = (L[S/\alpha])^{\uparrow}$; *and*
– $\forall \alpha, S, \varepsilon_1, \varepsilon_2.\ \alpha \mapsto S \vdash \varepsilon_1, \varepsilon_2 : \mathbf{Eff} \wedge \varepsilon_1 \sim \varepsilon_2 \implies \varepsilon_1[S/\alpha] \sim \varepsilon_2[S/\alpha]$.

For example, an effect algebra for effect sets can be given as follows.

Example 3.8 (Effect Sets)**.** An effect algebra $\mathrm{EA}_{\mathrm{Set}}$ for effect sets is a tuple $\langle \Sigma_{\mathrm{eff}}^{\mathrm{Set}}, - \underline{\cup} -, \{\}, \{-\}, \sim_{\mathrm{Set}} \rangle$ where $\sim_{\mathrm{Set}}$ is the least equivalence relation satisfying the following rules:

$$\frac{}{\varepsilon \underline{\cup} \{\} \sim_{\mathrm{Set}} \varepsilon} \qquad \frac{}{\varepsilon_1 \underline{\cup} \varepsilon_2 \sim_{\mathrm{Set}} \varepsilon_2 \underline{\cup} \varepsilon_1} \qquad \frac{}{\varepsilon \underline{\cup} \varepsilon \sim_{\mathrm{Set}} \varepsilon}$$

$$\frac{}{(\varepsilon_1 \underline{\cup} \varepsilon_2) \underline{\cup} \varepsilon_3 \sim_{\mathrm{Set}} \varepsilon_1 \underline{\cup} (\varepsilon_2 \underline{\cup} \varepsilon_3)} \qquad \frac{\varepsilon_1 \sim_{\mathrm{Set}} \varepsilon_2 \quad \varepsilon_3 \sim_{\mathrm{Set}} \varepsilon_4}{\varepsilon_1 \underline{\cup} \varepsilon_3 \sim_{\mathrm{Set}} \varepsilon_2 \underline{\cup} \varepsilon_4}$$

These rules reflect that the union operator in sets has the identity element $\{\}$ and satisfies commutativity, idempotence, associativity, and compatibility.

We also show an instance for simple rows and scoped rows.

Example 3.9 (Simple Rows)**.** The effect signature $\Sigma_{\mathrm{eff}}^{\mathrm{Row}}$ for simple rows is the set of $\langle\rangle : \mathbf{Eff}$ and $\langle - \mid - \rangle : \mathbf{Lab} \times \mathbf{Eff} \to \mathbf{Eff}$. An effect algebra $\mathrm{EA}_{\mathrm{SimpR}}$ for them is $\langle \Sigma_{\mathrm{eff}}^{\mathrm{Row}}, \odot_{\mathrm{SimpR}}, \langle\rangle, \langle - \mid \langle\rangle \rangle, \sim_{\mathrm{SimpR}} \rangle$ where

$$\varepsilon_1 \odot_{\mathrm{SimpR}} \varepsilon_2 \stackrel{\mathrm{def}}{=} \begin{cases} \langle L_1 \mid \langle \cdots \langle L_n \mid \varepsilon_2 \rangle \cdots \rangle \rangle & (\text{if } \varepsilon_1 = \langle L_1 \mid \langle \cdots \langle L_n \mid \langle\rangle \rangle \cdots \rangle \rangle) \\ \varepsilon_1 & (\text{if } \varepsilon_1 = \langle L_1 \mid \langle \cdots \langle L_n \mid \rho \rangle \cdots \rangle \rangle \text{ and } \varepsilon_2 = \langle\rangle) \end{cases}$$

and $\sim_{\mathrm{SimpR}}$ is the least equivalence relation satisfying the following.

$$\frac{\varepsilon_1 \sim_{\mathrm{SimpR}} \varepsilon_2}{\langle L \mid \varepsilon_1 \rangle \sim_{\mathrm{SimpR}} \langle L \mid \varepsilon_2 \rangle} \qquad \frac{L_1 \neq L_2}{\langle L_1 \mid \langle L_2 \mid \varepsilon \rangle \rangle \sim_{\mathrm{SimpR}} \langle L_2 \mid \langle L_1 \mid \varepsilon \rangle \rangle} \qquad \frac{}{\langle L \mid \varepsilon \rangle \sim_{\mathrm{SimpR}} \langle L \mid \langle L \mid \varepsilon \rangle \rangle}$$

Note that the definition of $\varepsilon_1 \odot_{\mathrm{SimpR}} \varepsilon_2$ depends on whether effect ε_1 ends with an effect variable. If it does, ε_2 must be empty because simple rows ending with effect variables cannot be extended. Otherwise, $\varepsilon_1 \odot_{\mathrm{SimpR}} \varepsilon_2$ simply concatenates ε_1 and ε_2.

The first rule of $\sim_{\mathrm{SimpR}}$ means that the results of adding the same label to equivalent effects are also equivalent. The remaining two rules allow reordering different labels and collapsing multiple occurrences of the same label into one, respectively. The collapsing of multiple occurrences reflects the characteristic of simple rows that the same label appears at most once in a row because it means that two or more occurrences of a label cannot be distinguished from one occurrence of it.

Example 3.10 (Scoped Rows)**.** An effect algebra $\mathrm{EA}_{\mathrm{ScpR}}$ for scoped rows is defined in a way similar to that for simple rows. The only difference is in the definition of equivalence $\sim_{\mathrm{ScpR}}$. The equivalence $\sim_{\mathrm{ScpR}}$ for scoped rows is defined as the least equivalence relation satisfying the following rules:

$$\frac{\varepsilon_1 \sim_{\mathrm{ScpR}} \varepsilon_2}{\langle L \mid \varepsilon_1 \rangle \sim_{\mathrm{ScpR}} \langle L \mid \varepsilon_2 \rangle} \qquad \frac{L_1 \neq L_2}{\langle L_1 \mid \langle L_2 \mid \varepsilon \rangle \rangle \sim_{\mathrm{ScpR}} \langle L_2 \mid \langle L_1 \mid \varepsilon \rangle \rangle}$$

Unlike simple rows, scoped rows are distinguished if they have different numbers of occurrences of some label.

4 λ_{EA}: A Calculus with Abstract Effect System

This section shows the syntax, semantics, and type-and-effect system of our language λ_{EA}. It is similar to the call-by-value polymorphic λ-calculi with algebraic effect handlers in the literature [Biernacki et al. 2018; Leijen 2017; Sekiyama et al. 2020] except that it is parameterized over effect algebras. Throughout this and the next sections, we fix a label signature Σ_{lab}, effect algebra $\langle \Sigma_{\mathrm{eff}}, \odot, \mathbb{0}, (-)^{\uparrow}, \sim \rangle$ over Σ_{lab}, and effect context Ξ, which are given as parameters.

$$
\begin{array}{rcll}
e & ::= & v \mid v_1\ v_2 \mid v\ S \mid \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \mid \mathbf{handle}_{l\,S^I}\ e\ \mathbf{with}\ h & \text{(expressions)} \\
v & ::= & x \mid \mathbf{fun}\ (f, x, e) \mid \Lambda\alpha : K.e \mid \mathrm{op}_{l\,S^I}\ T^{\mathcal{J}} & \text{(values)} \\
h & ::= & \{\ \mathbf{return}\ x \mapsto e\} \mid h \uplus \{\mathrm{op}\ \boldsymbol{\beta}^{\mathcal{J}} : \boldsymbol{K}^{\mathcal{J}}\ p\ k \mapsto e\} & \text{(handlers)} \\
E & ::= & \square \mid \mathbf{let}\ x = E\ \mathbf{in}\ e \mid \mathbf{handle}_{l\,S^I}\ E\ \mathbf{with}\ h & \text{(evaluation contexts)}
\end{array}
$$

Fig. 3. Program syntax of λ_{EA}.

4.1 Syntax

We show the program syntax of λ_{EA} in Figure 3.

Expressions, ranged over e, are composed of: values; function applications $v_1\ v_2$; typelike applications $v\ S$; let-bindings $\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2$; and handling expressions $\mathbf{handle}_{l\,S^I}\ e\ \mathbf{with}\ h$. Values are: variables x; recursive functions $\mathbf{fun}\ (f, x, e)$; typelike abstractions $\Lambda\alpha : K.e$; or operations $\mathrm{op}_{l\,S^I}\ T^{\mathcal{J}}$. An operation $\mathrm{op}_{l\,S^I}\ T^{\mathcal{J}}$ accompanies two typelike sequences S^I and $T^{\mathcal{J}}$, which are parameters of effect label l and operation op, respectively. We write $\lambda x.e$ for $\mathbf{fun}\ (f, x, e)$ when variable f does not occur free in expression e.

An effect handler for label name l possesses one return clause and clauses for the operations of l. For a return clause $\{\ \mathbf{return}\ x \mapsto e\}$, the body e is executed once a handled expression evaluates to a value v; x is used to refer to the value v. For an operation clause $\{\mathrm{op}\ \boldsymbol{\beta} : \boldsymbol{K}\ p\ k \mapsto e\}$, the body e is executed once a handled expression calls operation op. Typelike variables $\boldsymbol{\beta}$, variable p, and variable k are replaced by typelike parameters attached to the operation call, the argument of the call, and the delimited continuation from the call up to the handling expression installing the effect handler, respectively.

Evaluation contexts, ranged over by E, are defined in a standard manner. They may wrap a hole $\square$ by let-constructs and handling constructs.

4.2 Operational Semantics

The operational semantics of λ_{EA} is defined in Figure 4. Following Biernacki et al. [2018], it uses the notion of *freeness*, which helps define the operational semantics of lift coercions in Section 7.1. Figure 4 defines 0-*freeness* of labels [Biernacki et al. 2018]. The judgment $0{-}\mathrm{free}(L, E)$, which is read as "an label L is 0-free in an evaluation context E," means that any operation of L called under E is not handled. For example, $0{-}\mathrm{free}(L, \square)$ and $0{-}\mathrm{free}(L, \mathbf{handle}_{L'}\ \mathbf{let}\ x = \square\ \mathbf{in}\ e\ \mathbf{with}\ h)$ hold (if $L \neq L'$). This is as expected because any operation of L called under the evaluation context $\square$ or $\mathbf{handle}_{L'}\ \mathbf{let}\ x = \square\ \mathbf{in}\ e\ \mathbf{with}\ h$ is never handled. By contrast, $0{-}\mathrm{free}(L, \mathbf{handle}_{L}\ \mathbf{let}\ x = \square\ \mathbf{in}\ e\ \mathbf{with}\ h)$ does not hold as any operation of L called under $\mathbf{handle}_{L}\ \mathbf{let}\ x = \square\ \mathbf{in}\ e\ \mathbf{with}\ h$ is handled by h. The operational semantics of λ_{EA} uses this notion to ensure that every call to an operation of effect label L is handled by the innermost L's effect handler enclosing the operation call. We generalize 0-freeness to n-freeness for an arbitrary natural number n in introducing lift coercions (Section 7.1).

We show the operational semantics of λ_{EA} in Figure 4. The semantics comprises two binary relations: the reduction relation $\longmapsto$ and the evaluation relation $\longrightarrow$. The reduction relation defines the basic computation; in contrast, the evaluation relation gives a way of reducing subexpressions.

The reduction relation is defined by five rules. Function applications, typelike applications, let-bindings are reduced as usual. The remaining are the standard rules to reduce handling expressions. Consider an expression $\mathbf{handle}_{l\,S^I}\ e\ \mathbf{with}\ h$. If the handled expression e is a value v, the rule R_Handle1 reduces the handling expression to the body e_r of the return clause $\{\ \mathbf{return}\ x \mapsto e_r\}$ of h by substituting v for x in e_r. The other rule R_Handle2 is used when e calls an operation op of label name l, that is, e takes the form $E[\mathrm{op}_{l\,S^I}\ T^{\mathcal{J}}\ v]$ for some E, $T^{\mathcal{J}}$, and v (it is guaranteed

Freeness of labels $\boxed{n\text{–free}(L, E)}$

$$\frac{}{0\text{–free}(L, \square)} \qquad \frac{n\text{–free}(L, E)}{n\text{–free}(L, \mathbf{let}\ x = E\ \mathbf{in}\ e)} \qquad \frac{n\text{–free}(L, E) \quad L \neq L'}{n\text{–free}(L, \mathbf{handle}_{L'}\ E\ \mathbf{with}\ h)}$$

Reduction $\boxed{e \longmapsto e'}$

$$\frac{}{\mathbf{fun}\ (f, x, e)\ v \longmapsto e[\mathbf{fun}\ (f, x, e)/f][v/x]}\ \text{R\_App} \qquad \frac{}{(\Lambda\alpha : K.e)\ S \longmapsto e[S/\alpha]}\ \text{R\_TApp}$$

$$\frac{}{\mathbf{let}\ x = v\ \mathbf{in}\ e \longmapsto e[v/x]}\ \text{R\_Let} \qquad \frac{\mathbf{return}\ x \mapsto e_r \in h}{\mathbf{handle}_{l\,S^I}\ v\ \mathbf{with}\ h \longmapsto e_r[v/x]}\ \text{R\_Handle1}$$

$$\frac{\mathrm{op}\ \boldsymbol{\beta}^{\mathcal{J}} : \boldsymbol{K}^{\mathcal{J}}\ p\ k \mapsto e \in h \quad v_{cont} = \lambda z.\mathbf{handle}_{l\,S^I}\ E[z]\ \mathbf{with}\ h \quad 0\text{–free}(l\,S^I, E)}{\mathbf{handle}_{l\,S^I}\ E[\mathrm{op}_{l\,S^I}\ \boldsymbol{T}^{\mathcal{J}}\ v]\ \mathbf{with}\ h \longmapsto e[\boldsymbol{T}^{\mathcal{J}}/\boldsymbol{\beta}^{\mathcal{J}}][v/p][v_{cont}/k]}\ \text{R\_Handle2}$$

Evaluation $\boxed{e \longrightarrow e'}$

$$\frac{e_1 \longmapsto e_2}{E[e_1] \longrightarrow E[e_2]}\ \text{E\_Eval}$$

Fig. 4. Operational semantics of λ_{EA}.

by the type-and-effect system that the typelike arguments to l in the operation call are S^I). The reduction rule R_Handle2 assumes $0\text{–free}(l\,S^I, E)$, which ensures that h is the effect handler closest to the operation call among the ones for $l\,S^I$. After substituting the argument typelikes $\boldsymbol{T}^{\mathcal{J}}$, the argument value v, and the captured delimited continuation v_{cont} (which installs the effect handler h on the captured evaluation context E because effect handlers in λ_{EA} are *deep*) for the corresponding variables of op's operation clause in h, the evaluation proceeds to reducing the clause's body.

The evaluation relation only has one rule E_Eval. It means that the evaluation of an entire program proceeds by decomposing it into a redex e and an evaluation context E, reducing e to an expression e', and then filling the hole of E with the reduction result e'.

4.3 Type-and-Effect System

We show the type-and-effect system of λ_{EA} in Figure 5. Typing judgments are of the form $\Gamma \vdash e : A \mid \varepsilon$, meaning that an expression e is typed at A under a typing context Γ and the evaluation of e may cause effect ε. The rules for variables, function abstractions, function applications, typelike abstractions, typelike applications, and let-bindings are standard.

The rule T_Sub allows subsumption by subtyping. We show the subtyping relation $\Gamma \vdash A <: B$ for values and the one $\Gamma \vdash A \mid \varepsilon_1 <: B \mid \varepsilon_2$ for computations at the bottom of Figure 5. The subtyping rules are standard except for the subeffecting $\Gamma \vdash \varepsilon_1 \circledcirc \varepsilon_2$, which is used in the rule ST_Comp for the second subtyping relation. The subeffecting is defined via the given effect algebra:

$$\Gamma \vdash \varepsilon_1 \circledcirc \varepsilon_2 \stackrel{\text{def}}{=} \exists \varepsilon.\, \varepsilon_1 \odot \varepsilon \sim \varepsilon_2 \wedge (\forall \varepsilon' \in \{\varepsilon_1, \varepsilon_2, \varepsilon\}.\, \Gamma \vdash \varepsilon' : \mathbf{Eff})\ .$$

The rule T_Op typechecks operation $\mathrm{op}_{l\,S^I}\ \boldsymbol{T}^{\mathcal{J}}$ if op belongs to effect label l, and if the kinds of typelike arguments S^I and $\boldsymbol{T}^{\mathcal{J}}$ are matched with those of parameters of l in the effect context Ξ. The operation is given a function type determined by the argument and return type of op in Ξ and typelike arguments S^I and $\boldsymbol{T}^{\mathcal{J}}$. Because every call to the operation only invokes effect label $l\,S^I$, the latent effect of the function type is given by injecting $l\,S^I$ via $(-)^{\uparrow}$.

Typing $\boxed{\Gamma \vdash e : A \mid \varepsilon}$

$$\frac{\vdash \Gamma \quad x : A \in \Gamma}{\Gamma \vdash x : A \mid \mathbb{0}}\ \text{T\_Var} \qquad \frac{\Gamma, f : A \to_\varepsilon B, x : A \vdash e : B \mid \varepsilon}{\Gamma \vdash \mathbf{fun}\,(f, x, e) : A \to_\varepsilon B \mid \mathbb{0}}\ \text{T\_Abs}$$

$$\frac{\Gamma \vdash v_1 : A \to_\varepsilon B \mid \mathbb{0} \quad \Gamma \vdash v_2 : A \mid \mathbb{0}}{\Gamma \vdash v_1\, v_2 : B \mid \varepsilon}\ \text{T\_App} \qquad \frac{\Gamma, \alpha : K \vdash e : A \mid \varepsilon}{\Gamma \vdash \Lambda\alpha : K.e : \forall \alpha : K.A^\varepsilon \mid \mathbb{0}}\ \text{T\_TAbs}$$

$$\frac{\Gamma \vdash v : \forall \alpha : K.A^\varepsilon \mid \mathbb{0} \quad \Gamma \vdash S : K}{\Gamma \vdash v\,S : A[S/\alpha] \mid \varepsilon[S/\alpha]}\ \text{T\_TApp} \qquad \frac{\Gamma \vdash e_1 : A \mid \varepsilon \quad \Gamma, x : A \vdash e_2 : B \mid \varepsilon}{\Gamma \vdash \mathbf{let}\, x = e_1\, \mathbf{in}\, e_2 : B \mid \varepsilon}\ \text{T\_Let}$$

$$\frac{\Gamma \vdash e : A \mid \varepsilon \quad \Gamma \vdash A \mid \varepsilon <: A' \mid \varepsilon'}{\Gamma \vdash e : A' \mid \varepsilon'}\ \text{T\_Sub}$$

$$\frac{\begin{array}{c} l :: \forall \boldsymbol{\alpha}^I : \boldsymbol{K}^I.\sigma \in \Xi \quad \mathrm{op} : \forall \boldsymbol{\beta}^J : \boldsymbol{K'}^J.A \Rightarrow B \in \sigma[\boldsymbol{S}^I/\boldsymbol{\alpha}^I] \\ \vdash \Gamma \quad \Gamma \vdash \boldsymbol{S}^I : \boldsymbol{K}^I \quad \Gamma \vdash \boldsymbol{T}^J : \boldsymbol{K'}^J \end{array}}{\Gamma \vdash \mathrm{op}_{l\boldsymbol{S}^I}\, \boldsymbol{T}^J : (A[\boldsymbol{T}^J/\boldsymbol{\beta}^J]) \to_{(l\boldsymbol{S}^I)\uparrow} (B[\boldsymbol{T}^I/\boldsymbol{\beta}^I]) \mid \mathbb{0}}\ \text{T\_Op}$$

$$\frac{\begin{array}{c} \Gamma \vdash e : A \mid \varepsilon' \qquad l :: \forall \boldsymbol{\alpha}^I : \boldsymbol{K}^I.\sigma \in \Xi \qquad \Gamma \vdash \boldsymbol{S}^I : \boldsymbol{K}^I \\ \Gamma \vdash_{\sigma[\boldsymbol{S}^I/\boldsymbol{\alpha}^I]} h : A \Rightarrow^\varepsilon B \quad (l\boldsymbol{S}^I)^\uparrow \odot \varepsilon \sim \varepsilon' \end{array}}{\Gamma \vdash \mathbf{handle}_{l\boldsymbol{S}^I}\, e\, \mathbf{with}\, h : B \mid \varepsilon}\ \text{T\_Handling}$$

Handler Typing $\boxed{\Gamma \vdash_\sigma h : A \Rightarrow^\varepsilon B}$

$$\frac{\Gamma, x : A \vdash e_r : B \mid \varepsilon}{\Gamma \vdash_{\{\}} \{\, \mathbf{return}\, x \mapsto e_r\} : A \Rightarrow^\varepsilon B}\ \text{H\_Return}$$

$$\frac{\begin{array}{c} \sigma = \sigma' \uplus \{\mathrm{op} : \forall \boldsymbol{\beta}^J : \boldsymbol{K}^J.A' \Rightarrow B'\} \\ \Gamma \vdash_{\sigma'} h : A \Rightarrow^\varepsilon B \qquad \Gamma, \boldsymbol{\beta}^J : \boldsymbol{K}^J, p : A', k : B' \to_\varepsilon B \vdash e : B \mid \varepsilon \end{array}}{\Gamma \vdash_\sigma h \uplus \{\mathrm{op}\, \boldsymbol{\beta}^J : \boldsymbol{K}^J\, p\, k \mapsto e\} : A \Rightarrow^\varepsilon B}\ \text{H\_Op}$$

Subtyping $\boxed{\Gamma \vdash A <: B}$ $\boxed{\Gamma \vdash A \mid \varepsilon_1 <: B \mid \varepsilon_2}$

$$\frac{\Gamma \vdash A : \mathbf{Typ}}{\Gamma \vdash A <: A}\ \text{ST\_Refl} \qquad \frac{\Gamma \vdash A_2 <: A_1 \quad \Gamma \vdash B_1 \mid \varepsilon_1 <: B_2 \mid \varepsilon_2}{\Gamma \vdash A_1 \to_{\varepsilon_1} B_1 <: A_2 \to_{\varepsilon_2} B_2}\ \text{ST\_Fun}$$

$$\frac{\Gamma, \alpha : K \vdash A_1 \mid \varepsilon_1 <: A_2 \mid \varepsilon_2}{\Gamma \vdash \forall \alpha : K.{A_1}^{\varepsilon_1} <: \forall \alpha : K.{A_2}^{\varepsilon_2}}\ \text{ST\_Poly} \qquad \frac{\Gamma \vdash A_1 <: B \quad \Gamma \vdash \varepsilon_1 \circledcirc \varepsilon_2}{\Gamma \vdash A \mid \varepsilon_1 <: B \mid \varepsilon_2}\ \text{ST\_Comp}$$

Fig. 5. Type-and-effect system of λ_{EA}.

The rule T_Handling is for handling expressions. Assume that a handled expression e is of type A and has effect ε'. If it is handled by an effect handler for effect label $l\boldsymbol{S}^I$, the operations of $l\boldsymbol{S}^I$ become unobservable from the outer context. Thus, the effect ε of the handling expression is the result of removing label $l\boldsymbol{S}^I$ from effect ε'. This "label-removing manipulation" is represented as $\Gamma \vdash (l\boldsymbol{S}^I)^\uparrow \odot \varepsilon \sim \varepsilon'$. Therefore, the result ε of the label-removing manipulation depends on the given effect algebra. For example, if the effect algebra $\mathrm{EA}_{\mathrm{SimpR}}$ for simple rows is given, the result of removing the label Exc from the effect $\langle \mathsf{Exc} \mid \langle \mathsf{Exc} \mid \langle \mathsf{Choice} \mid \langle\rangle \rangle \rangle \rangle$ can be $\langle \mathsf{Choice} \mid \langle\rangle \rangle$ because

$\langle \mathsf{Exc} \mid \langle\rangle \rangle \odot_{\mathrm{SimpR}} \langle \mathsf{Choice} \mid \langle\rangle \rangle \sim_{\mathrm{SimpR}} \langle \mathsf{Exc} \mid \langle \mathsf{Exc} \mid \langle \mathsf{Choice} \mid \langle\rangle \rangle \rangle \rangle$ holds (recall that simple rows can collapse multiple occurrences of the same label into one). On the contrary, the removing result in the algebra $\mathrm{EA}_{\mathrm{ScpR}}$ for scoped rows can be $\langle \mathsf{Exc} \mid \langle \mathsf{Choice} \mid \langle\rangle \rangle \rangle$ but cannot be $\langle \mathsf{Choice} \mid \langle\rangle \rangle$.

The type B of the handling expression is determined by handler h: typing judgments for handlers take the form $\Gamma \vdash_\sigma h : A \Rightarrow^\varepsilon B$, which means that handler h transforms computation of type A involving an effect label with operation signature σ to that of type B with effect ε. The rules H_Return and H_Op are for return and operation clauses and reflect the reduction rules R_Handle1 and R_Handle2, respectively. Note that the return type of a continuation variable k equals the type B of the handling expression as the effect handlers in λ_{EA} are deep [Kammar et al. 2013].

5 Safety Properties

This section shows the safety properties of λ_{EA}. The proofs rely on safety conditions, which are requirements on effect algebras. Under the assumption that a given effect algebra meets the safety conditions, we prove type-and-effect safety of λ_{EA}.

5.1 Safety Conditions

To prove type-and-effect safety, a given effect algebra must meet safety conditions shown in the following. We write $\varepsilon_1 \lessdot \varepsilon_2$ to state that $\varepsilon_1 \odot \varepsilon \sim \varepsilon_2$ for some ε.

Definition 5.1 (Safety Conditions)**.**
(1) For any L, $(L)^\uparrow \lessdot \mathbb{0}$ does not hold.
(2) If $(L)^\uparrow \lessdot \varepsilon$ and $(L')^\uparrow \odot \varepsilon' \sim \varepsilon$ and $L \neq L'$, then $(L)^\uparrow \lessdot \varepsilon'$.

Condition (1) disallows the subeffecting to hide an invoked effect label L as if it were not performed. Condition (2) means that, if an expression invoking a label L is given an effect ε, and an effect handler for a different label L' handles the expression, then the information of L still remains in the effect ε' assigned to the handling expression (that is, it is observable from the outer context).

To understand problems excluded by safety conditions (1) and (2), we consider effect algebras that violate one of the conditions, and then show unsafe programs being typeable under the algebras.

Example 5.2 (Unsafe Effect Algebras)**.**

Effect algebra violating safety condition (1) Consider an effect algebra such that $\emptyset \vdash (l)^\uparrow \lessdot \mathbb{0}$ holds for some l. Clearly, this effect algebra violates safety condition (1). In this case, $\emptyset \vdash \mathrm{op}_l\, v : A \mid \mathbb{0}$ can be derived for some A (if $\mathrm{op}_l\, v$ is well typed) because $\mathrm{op}_l\, v$ is given the effect $(l)^\uparrow$ and the subeffecting $\emptyset \vdash (l)^\uparrow \lessdot \mathbb{0}$ holds. However, the operation call is not handled.

Effect algebra violating safety condition (2) Consider an effect algebra such that safety condition (1), $(l)^\uparrow \lessdot (l')^\uparrow$, and $(l')^\uparrow \odot \mathbb{0} \sim (l')^\uparrow$ hold for some l and l' such that $l \neq l'$. This effect algebra must violate safety condition (2): if safety condition (2) were met, we would have $(l)^\uparrow \lessdot \mathbb{0}$, but it is contradictory with safety condition (1).
This effect algebra allows assigning the empty effect $\mathbb{0}$ to the expression $\mathbf{handle}_{l'}\, \mathrm{op}_l\, v\, \mathbf{with}\, h$ as illustrated by the following typing derivation, but the operation call in it is not handled.

$$\dfrac{\cdots \quad (l')^\uparrow \odot \mathbb{0} \sim (l')^\uparrow \qquad \dfrac{\emptyset \vdash \mathrm{op}_l\, v : A \mid (l)^\uparrow \quad \emptyset \vdash A \mid (l)^\uparrow <: A \mid (l')^\uparrow}{\emptyset \vdash \mathrm{op}_l\, v : A \mid (l')^\uparrow}\ \textsc{T\_Sub}}{\emptyset \vdash \mathbf{handle}_{l'}\, \mathrm{op}_l\, v\, \mathbf{with}\, h : B \mid \mathbb{0}}\ \textsc{T\_Handling}$$

Note that the effect algebras $\mathrm{EA}_{\mathrm{Set}}$, $\mathrm{EA}_{\mathrm{SimpR}}$, $\mathrm{EA}_{\mathrm{ScpR}}$, for which the effect safety has been shown in the literature, meet the safety conditions.

Theorem 5.3. *The effect algebras $\mathrm{EA}_{\mathrm{Set}}$, $\mathrm{EA}_{\mathrm{SimpR}}$, and $\mathrm{EA}_{\mathrm{ScpR}}$ meet safety conditions (1) and (2).*

5.2 Type-and-Effect Safety

This section shows type-and-effect safety. To prove it, we assume that an effect algebra meets the safety conditions and an effect context is proper, which means that it is consistent with a given label signature Σ_{lab} and the types of operations in it are well formed.

Definition 5.4 (Proper Effect Contexts). *An effect context Ξ is* proper *if, for any $l :: \forall \boldsymbol{\alpha}^I : \boldsymbol{K}^I.\sigma \in \Xi$, the following holds:*

- $l : \Pi \boldsymbol{K}^I \to \mathbf{Lab} \in \Sigma_{\mathrm{lab}}$*;*
- *the type schemes $\forall \boldsymbol{\alpha_0}^{I_0} : \boldsymbol{K_0}^{I_0}.\sigma_0$ associated with l by Ξ are uniquely determined; and*
- *for any* $\mathrm{op} : \forall \boldsymbol{\beta}^J : \boldsymbol{K_0}^J.A \Rightarrow B \in \sigma$ *and* $C \in \{A, B\}$, $\boldsymbol{\alpha}^I : \boldsymbol{K}^I, \boldsymbol{\beta}^J : \boldsymbol{K_0}^J \vdash C : \mathbf{Typ}$.

5.2.1 Type Safety. The statement of type safety is as follows. We write $\longrightarrow^*$ for the reflexive, transitive closure of $\longrightarrow$ and $e \not\longrightarrow$ to denote that there is no e' such that $e \longrightarrow e'$.

Lemma 5.5 (Type Safety). *If $\emptyset \vdash e : A \mid \varepsilon$ and $e \longrightarrow^* e' \not\longrightarrow$, then one of the following holds:*

- *$e' = v$ for some value v such that $\emptyset \vdash v : A \mid \varepsilon$; or*
- *$e' = E[\mathrm{op}_{l\,S^I}\, \boldsymbol{T}^J\, v]$ for some E, l, $\boldsymbol{S}^I$, op, $\boldsymbol{T}^J$, and v such that $0\text{–free}(l\,\boldsymbol{S}^I, E)$.*

While the type safety guarantees that the result of a program, if any, has the same type as the program, it does not ensure that all operations are handled even if the effect $\mathbb{0}$, which denotes that no unhandled operation remains, is assigned to the program: as shown shortly, the latter property is guaranteed by effect safety.

Type safety is proven via progress and preservation as usual [Wright and Felleisen 1994].

Lemma 5.6 (Progress). *If $\emptyset \vdash e : A \mid \varepsilon$, then one of the following holds: e is a value; $e \longrightarrow e'$ for some e'; or $e = E[\mathrm{op}_{l\,S^I}\, \boldsymbol{T}^J\, v]$ for some E, l, $\boldsymbol{S}^I$, op, $\boldsymbol{T}^J$, and v such that $0\text{–free}(l\,\boldsymbol{S}^I, E)$.*

Lemma 5.7 (Preservation). *If $\emptyset \vdash e : A \mid \varepsilon$ and $e \longrightarrow e'$, then $\emptyset \vdash e' : A \mid \varepsilon$.*

5.2.2 Effect Safety. Effect safety is stated as follows.

Lemma 5.8 (Effect Safety). *If $\Gamma \vdash e : A \mid \mathbb{0}$, then there exist no E, l, $\boldsymbol{S}^I$, op, $\boldsymbol{T}^J$, and v such that both $e = E[\mathrm{op}_{l\,S^I}\, \boldsymbol{T}^J\, v]$ and $0\text{–free}(l\,\boldsymbol{S}^I, E)$ hold.*

This lemma means, if an expression is assigned to $\mathbb{0}$, no unhandled operation call remains there.

5.2.3 Type-and-Effect Safety. We obtain type-and-effect safety—terminating programs with effect $\mathbb{0}$ always evaluates to values—as a corollary from type safety and effect safety.

Theorem 5.9 (Type-and-Effect Safety). *If $\emptyset \vdash e : A \mid \mathbb{0}$ and $e \longrightarrow^* e' \not\longrightarrow$, then $e' = v$ for some v.*

Corollary 5.10. *The effect system instantiated by the effect algebra $\mathrm{EA}_{\mathrm{Set}}$, $\mathrm{EA}_{\mathrm{SimpR}}$, or $\mathrm{EA}_{\mathrm{ScpR}}$ meets the type-and-effect safety (that is, any well-typed program terminates at a value unless it diverges).*

6 Formal Relationships between λ_{EA} and The Existing Systems

This section shows that λ_{EA} soundly models the key aspects of the existing effect systems. As targets, we select the effect systems of Pretnar [2015], Hillerström et al. [2017], and Leijen [2017], which employ sets, simple rows, and scoped rows, respectively, to represent effect collections. We call them Eff, Links, and Koka because they model the core part of the programming languages Eff [Bauer and Pretnar 2021], Links [Lindley et al. 2023], and Koka [Leijen 2024], respectively.[6]

[6]The core effect system of Links was first presented by Hillerström and Lindley [2016], but it seems to have a minor flaw in the typing of sequential composition. We thus refer to Hillerström et al. [2017] where the flaw is fixed.

Table 1. Comparison of the effectful aspects in λ_{EA} and the existing works. The mark ✗ means "not supported," and "explicit*" in the column "polymorphism" for Links indicates that Links supports not only explicit type and effect polymorphism, but also row polymorphism in the style of Rémy [1994] at the effect-level.

	effect collections	collected effects	effect contexts' assignment	polymorphism
λ_{EA}	effect algebras	label	global	explicit
Eff	sets	operation	global	✗
Links	simple rows	operation	local	explicit*
Koka	scoped rows	label	global	implicit

6.1 Differences between λ_{EA} and The Selected Systems

We aim to establish the formal connection between each of the existing systems and λ_{EA}, but there exist some gaps between them. First, the existing systems adopt their own syntax not only for effects but also for types and programs, which hinders the formal comparison. To address this problem, we define a syntactic translation $\mathbb{T}_{\mathcal{E}}$ from each $\mathcal{E}$ of the selected systems to the instance of λ_{EA} with the corresponding effect algebra. For example, operation calls in Eff take the form $\mathrm{op}(v, y.c)$, carrying continuations $y.c$. The translator $\mathbb{T}_{\mathrm{Eff}}$ converts it to the expression $\mathbf{let}\, y = \mathrm{op}_l\, \mathbb{T}_{\mathrm{Eff}}(v)\, \mathbf{in}\, \mathbb{T}_{\mathrm{Eff}}(c)$ in λ_{EA} using some appropriate label l. Readers interested in the complete definitions of the translations are referred to the supplementary material.

The remaining gaps between λ_{EA} and the existing systems are summarized in Table 1. Because addressing the gaps other than the representation of effect collections is beyond the scope of the present work, we impose certain assumptions on the existing systems for the comparison. In what follows, we detail the gaps and how we address them.

Collected effects. In λ_{EA}, effect collections gather effect labels, which are sets of operations of some specific effects. For example, the effect for state can be expressed by a label State equipped with operations get and set for getting and updating, respectively, the current state. In this style, which we call *label-based*, an operation call is given an effect collection including the effect label to which the called operation belongs, and a handler is required to handle all the operations of a specified label. λ_{EA} and Koka employ the label-based style. By contrast, Eff and Links adopt the *operation-based* style, where effect collections gather operations. In this style, an operation call is given an effect collection including the called operation (not labels), and effect handlers can implement any operation freely. To address this difference, when translating Eff and Links in the operation-based style to λ_{EA} in the label-based style, we assume that some labels are given and any effect collection appearing in Eff and Links can be decomposed into a subset of the given labels.

Effect contexts' assignment. Our language λ_{EA} supposes that an effect context Ξ is fixed during typechecking one program. We call this assignment of Ξ *global*. Eff and Koka employ the same assignment style for effect contexts. In contrast, in Links, effect contexts can change during the typechecking. For example, consider the following program.

$$\mathbf{handle}\,(\mathbf{if}\,\mathsf{ask}\,()\,\mathbf{then}\,0\,\mathbf{else}\,(\mathbf{handle}\,\mathsf{ask}\,()+1\,\mathbf{with}\,\{\,\mathbf{return}\,x \mapsto x\} \uplus \{\mathsf{ask}\,z\,k \mapsto k\,2\}))$$
$$\mathbf{with}\,\{\,\mathbf{return}\,x \mapsto x\} \uplus \{\mathsf{ask}\;z\,k \mapsto k\,\mathsf{true}\}$$

In this program, both ask operation calls take the unit value, but the first and second ones return Booleans and integers, respectively. This program cannot be typechecked if an effect context is globally fixed. Links can typecheck it because Links allows enclosing handlers to modify effect contexts; namely, effect contexts are assigned *locally*. To address the local assignment of effect

contexts, we assume that every operation has a unique, closed type in Links, which enables determining the types of operations globally.

Polymorphism. The languages λ_{EA}, Links, and Koka support type and effect polymorphism. Among them, only the polymorphism in Koka is *implicit*, that is, no term constructor for type abstraction and application is given. Unfortunately, it is not straightforward to translate a program (or its typing derivation) with implicit polymorphism in Koka to one with *explicit* polymorphism in λ_{EA} while preserving the meaning of the program because Koka does not adopt *value restriction* [Tofte 1990; Wright 1995]. Our approach to this difference in polymorphism is simply to forbid the use of implicit polymorphism in Koka and instead introduce explicit polymorphism by equipping Koka with term constructors for type abstraction and application as in λ_{EA} and Links. It is also noteworthy that Links supports more advanced polymorphism, inspired by row polymorphism proposed by Rémy [1994]. It introduces *presence types*, which can state that a specific label is present or absent in a row, *presence polymorphism*, and effect variables constrained by which labels are present or absent. This form of polymorphism facilitates solving unification problems in the composition of effect handlers [Hillerström and Lindley 2016]. Our translation from Links to λ_{EA} addresses these unique features in Links as follows: first, present labels remain in the translated row but labels with the absent flag do not; second, the constraints on effect variables are ignored; third, we assume that programs to be translated do not use presence polymorphism. We left the support for presence polymorphism as future work: it seems to be motivated by unification and type inference, which are beyond the scope of the present work.

6.2 Type-and-Effect Preservation of Translations

We show that the translations preserve well-typedness under the aforementioned assumptions.

Theorem 6.1. *Let* $(\mathcal{E}, \mathcal{A}) \in \{(\textsc{Eff}, \mathrm{EA}_{\mathrm{Set}}), (\textsc{Links}, \mathrm{EA}_{\mathrm{SimpR}}), (\textsc{Koka}, \mathrm{EA}_{\mathrm{ScpR}})\}$. *If a program* c *in the system* $\mathcal{E}$ *is well typed at an effect* ϵ, *then* $\mathbb{T}_{\mathcal{E}}(c)$ *is well typed at effect* $\mathbb{T}_{\mathcal{E}}(\epsilon)$ *in* λ_{EA} *with* $\mathcal{A}$.

This result guarantees that, for each $\mathcal{E}$ of the selected systems, the programs in $\mathcal{E}$ can be safely executed in the semantics of λ_{EA}. In other words, λ_{EA} can work as an intermediate language that ensures type-and-effect safety. Note that the equivalence relation on scoped rows in Koka is more restrictive than $\sim_{\mathrm{ScpR}}$ in $\mathrm{EA}_{\mathrm{ScpR}}$ because the row equivalence in Koka allows swapping effect labels $l_1\, \boldsymbol{S_1}$ and $l_2\, \boldsymbol{S_2}$ only if $l \neq l'$, whereas $\sim_{\mathrm{ScpR}}$ allows their swapping if the label names l_1 and l_2, *or* the type arguments $\boldsymbol{S_1}$ and $\boldsymbol{S_2}$ are different. This gap does not prevent proving Theorem 6.1 because it only means that λ_{EA} with $\mathrm{EA}_{\mathrm{ScpR}}$ may accept more programs than Koka. We will show an effect algebra with the row equivalence in Koka in Section 7.2.

7 Extensions of λ_{EA}

This section extends λ_{EA} and safety conditions to lift coercions and type-erasure semantics. We also introduce effect algebras safe for these extensions (including a new one based on multisets) and discuss how adaptable each effect representation addressed in this paper—sets, multisets, simple rows, and scoped rows—is for the extensions.

7.1 Lift Coercions

This section shows an extension to *lift coercions* [Biernacki et al. 2018, 2019] (also known as injection [Leijen 2018] or masking [Leijen 2024]). Given an effect label, a lift coercion forbids the innermost handler for the label to handle any operation of the label. They can prevent *accidental handling*, a situation that an effect handler handles an operation call against the programmer's intention. This paper focuses on how λ_{EA} is extended with lift coercions; see the prior work [Biernacki

$$e ::= \cdots \mid [e]_L \quad \text{(expressions)} \qquad E ::= \cdots \mid [E]_L \quad \text{(evaluation contexts)}$$

Freeness of labels $\boxed{n\text{–free}(L, E)}$

$$\frac{(n+1)\text{–free}(L, E)}{n\text{–free}(L, \mathbf{handle}_L\, E\, \mathbf{with}\, h)} \qquad \frac{n\text{–free}(L, E)}{n+1\text{–free}(L, [E]_L)} \qquad \frac{n\text{–free}(L, E) \quad L \neq L'}{n\text{–free}(L, [E]_{L'})}$$

Reduction $\boxed{e \longmapsto e'}$ **Typing** $\boxed{\Gamma \vdash e : A \mid \varepsilon}$

$$\frac{}{[v]_L \longmapsto v}\ \text{R\_Lift} \qquad \frac{\Gamma \vdash e : A \mid \varepsilon' \quad \Gamma \vdash L : \mathbf{Lab} \quad (L)^\uparrow \odot \varepsilon' \sim \varepsilon}{\Gamma \vdash [e]_L : A \mid \varepsilon}\ \text{T\_Lift}$$

Fig. 6. The extension for lift coercions.

et al. 2018, 2019; Leijen 2018] for the detail of the accidental handling and how lift coercions work to address it. We also show that the effect algebras $\mathrm{EA}_{\mathrm{Set}}$ and $\mathrm{EA}_{\mathrm{SimpR}}$ are unsafe in the extension and that $\mathrm{EA}_{\mathrm{ScpR}}$ and a new effect algebra for *multisets* are safe. Note that Biernacki et al. [2019] introduce coercions in other forms. We do not support them because they can be encoded with lift coercions (if label polymorphism is not used) [Biernacki et al. 2018, 2019].

7.1.1 Extending λ_{EA} to Lift Coercions. We show the extended part of λ_{EA} in Figure 6. Expressions and evaluation contexts are extended with lift coercions $[-]_L$. To define the semantics of lift coercions, we generalize 0-freeness to *n-freeness* for an arbitrary natural number n by following Biernacki et al. [2018]. The predicate $n\text{–free}(L, E)$ is defined by the rules in Figure 6 in addition to the ones given previously (Figure 4). Intuitively, $n\text{–free}(L, E)$ means that, for an operation op of L, the operation call in $E[\mathrm{op}_L\, T^{\vec{J}}\, v]$ will be handled by the $(n+1)$-th innermost enclosing handler for L. For example, $1\text{–free}(L, [\square]_L)$ and $0\text{–free}(L, \mathbf{handle}_L\, [\square]_L\, \mathbf{with}\, h_1)$ hold. Because the semantics of the effect handling (specifically, the reduction rule R_Handle2 in Figure 4) requires the label of the handled operation call to be 0-free in the evaluation context enclosing the operation call, the operation call in $\mathbf{handle}_L\, \mathbf{handle}_L\, [\mathrm{op}_L\, v]_L\, \mathbf{with}\, h_1\, \mathbf{with}\, h_2$ will be handled by h_2. If a lift coercion is given a value, it returns the value as it is (R_Lift). The type-and-effect system is extended with the rule T_Lift, which allows the information ε' of effects of an expression e to pass through the innermost effect handler for a label L by prepending L to ε'.

7.1.2 Safety Conditions and Type-and-Effect Safety. To ensure the safety of programs in the presence of lift coercions, we introduce a new safety condition in addition to the ones given in Section 5.

Definition 7.1 (Safety Condition for Lift Coercions)**.** *The safety condition added for lift coercions is: (3) If $(L)^\uparrow \odot \varepsilon_1 \sim (L_1)^\uparrow \odot \cdots \odot (L_n)^\uparrow \odot (L)^\uparrow \odot \varepsilon_2$ and $L \neq L_i$ for any i, then $\varepsilon_1 \sim (L_1)^\uparrow \odot \cdots \odot (L_n)^\uparrow \odot \varepsilon_2$.*

This new condition can be understood as follows. First, let ε_2 be an effect of an expression e. Then, the effect of the expression $[\cdots [[e]_L]_{L_n} \cdots]_{L_1}$ is given as $(L_1)^\uparrow \odot \cdots \odot (L_n)^\uparrow \odot (L)^\uparrow \odot \varepsilon_2$. Assume that the expression is handled by an effect handler for L and the remaining effect is ε_1. Then, ε_1 should retain the information that e is surrounded by lift coercions for $L_1, \cdots, L_n$ because the handling expression may be enclosed by effect handlers for $L_1, \cdots, L_n$. Such information is described by $(L_1)^\uparrow \odot \cdots \odot (L_n)^\uparrow \odot \varepsilon_2$. Thus, safety condition (3) requires $\varepsilon_1 \sim (L_1)^\uparrow \odot \cdots \odot (L_n)^\uparrow \odot \varepsilon_2$.

To see the importance of the new safety condition more concretely, we show that the effect algebras $\mathrm{EA}_{\mathrm{Set}}$ and $\mathrm{EA}_{\mathrm{SimpR}}$ violate this new condition and then present how they make some unsafe programs typeable.

Theorem 7.2 (Unsafe Effect Algebras with Lift Coercions)**.** *The effect algebras* $\mathrm{EA}_{\mathrm{Set}}$ *and* $\mathrm{EA}_{\mathrm{SimpR}}$ *do not meet safety condition (3). Furthermore, there exists an expression that is well typed under* $\mathrm{EA}_{\mathrm{Set}}$ *and* $\mathrm{EA}_{\mathrm{SimpR}}$ *and gets stuck.*

Proof. We consider only $\mathrm{EA}_{\mathrm{Set}}$ here; a similar discussion can be applied to $\mathrm{EA}_{\mathrm{SimpR}}$. Recall that the operation $\odot$ in $\mathrm{EA}_{\mathrm{Set}}$ is implemented by the set union, so it meets idempotence: $\{L\} \cup \{L\} \sim \{L\}$. Furthermore, we can use the empty set as the identity element, so $\{L\} \cup \{L\} \sim \{L\} \cup \{\}$. If safety condition (3) was met, $\{L\} \sim \{\}$ (where $\{L\}$, $\{\}$, and 0 are taken as ε_1, ε_2, and n, respectively, in Definition 7.1). However, the equivalence does not hold.

As a program that is typeable under $\mathrm{EA}_{\mathrm{Set}}$, consider $\mathbf{handle}_{\mathrm{Exc}}\ [\mathsf{raise}_{\mathrm{Exc}}\ \mathsf{Unit}\ ()]_{\mathrm{Exc}}\ \mathbf{with}\ h$ where $\mathrm{Exc} :: \{\mathsf{raise} : \forall\alpha : \mathbf{Typ}.\mathsf{Unit} \Rightarrow \alpha\}$. This program can be typechecked under an appropriate assumption as illustrated by the following typing derivation:

$$\dfrac{\cdots \quad \{\mathrm{Exc}\} \cup \{\} \sim \{\mathrm{Exc}\} \qquad \dfrac{\dfrac{\emptyset \vdash \mathsf{raise}_{\mathrm{Exc}}\ \mathsf{Unit}\ () : A \mid \{\mathrm{Exc}\} \quad \{\mathrm{Exc}\} \cup \{\mathrm{Exc}\} \sim \{\mathrm{Exc}\}}{\emptyset \vdash [\mathsf{raise}_{\mathrm{Exc}}\ \mathsf{Unit}\ ()]_{\mathrm{Exc}} : A \mid \{\mathrm{Exc}\}}\ \text{T\_Lift}}{\emptyset \vdash \mathbf{handle}_{\mathrm{Exc}}\ [\mathsf{raise}_{\mathrm{Exc}}\ \mathsf{Unit}\ ()]_{\mathrm{Exc}}\ \mathbf{with}\ h : B \mid \{\}}\ \text{T\_Handling}$$

However, the call to raise is not handled as it needs to be handled by the *second* closest handler. ∎

In contrast, the effect algebra $\mathrm{EA}_{\mathrm{ScpR}}$ for scoped rows satisfies safety condition (3). The point is that $\odot_{\mathrm{ScpR}}$ in $\mathrm{EA}_{\mathrm{ScpR}}$ is *not* idempotent. Therefore, they can represent as the information of effects how many lift coercions are used and how many effect handlers are necessary to handle expression. This observation gives us a new effect algebra with *multisets*. Multisets can have multiple instances of the same element and their sum operation is also nonidempotent. Thus, we can expect—and it is the case—that the algebra for multisets meets safety condition (3) as well as the other conditions.

Example 7.3 (Effect Multisets)**.** The effect signature $\Sigma_{\mathrm{eff}}^{\mathrm{MSet}}$ of effect multisets is given by $\{\} : \mathbf{Eff}$, $\{-\} : \mathbf{Lab} \to \mathbf{Eff}$, and $- \sqcup - : \mathbf{Eff} \times \mathbf{Eff} \to \mathbf{Eff}$ (which is the sum operation for multisets). An effect algebra $\mathrm{EA}_{\mathrm{MSet}}$ for multisets is defined by $\langle \Sigma_{\mathrm{eff}}^{\mathrm{MSet}}, - \sqcup -, \{\}, \{-\}, \sim_{\mathrm{MSet}} \rangle$ where $\sim_{\mathrm{MSet}}$ is the least equivalence relation satisfying the same rules as $\sim_{\mathrm{Set}}$ except for the idempotence rule.

Theorem 7.4. *The effect algebras* $\mathrm{EA}_{\mathrm{ScpR}}$ *and* $\mathrm{EA}_{\mathrm{MSet}}$ *meet safety conditions (1)–(3).*

The type-and-effect safety of λ_{EA} with lift coercions is proven similarly to Theorem 5.9 provided that an effect algebra meets safety conditions (1)–(3).

Theorem 7.5 (Type-and-Effect Safety)**.** *Assume that a given effect algebra meets safety conditions (1)–(3). If* $\emptyset \vdash e : A \mid \mathbb{0}$ *and* $e \longrightarrow^* e' \not\longrightarrow$*, then* $e' = v$ *for some* v*.*

7.2 Type-Erasure Semantics

This section shows an adaption of λ_{EA} to type-erasure semantics, which is different from those given in Sections 4 and 7.1 in that it does not rely on type arguments of label names in seeking effect handlers matching with called operations. Type erasure semantics is helpful to develop efficient implementations of effect handlers with parametric effects [Biernacki et al. 2019].

7.2.1 Formal Definition of Type-Erasure Semantics. The part modified to support the type-erasure semantics is shown in Figure 7. The label freeness in the type-erasure semantics refers only to label names, while the original definition in Figure 4 refers to entire labels. The only change in the semantics is that the reduction rule R_Handle2 is replaced by R_Handle2' presented in Figure 7. For instance, consider an expression $\mathbf{handle}_{\mathsf{State\ Int}}\ (\mathbf{handle}_{\mathsf{State\ Bool}}\ (\mathsf{set}_{\mathsf{State}\ A}\ v)\ \mathbf{with}\ h_1)\ \mathbf{with}\ h_2$. In the original semantics, it depends on the type argument A which of h_1 and h_2 handles the operation

Freeness of label names $\boxed{n\text{–free}(l, E)}$

$$\frac{}{0\text{–free}(l, \square)} \qquad \frac{n\text{–free}(l, E)}{n\text{–free}(l, \mathbf{let}\ x = E\ \mathbf{in}\ e)} \qquad \frac{n\text{–free}(l, E) \quad l \neq l'}{n\text{–free}(l, \mathbf{handle}_{l'\ S^{I}}\ E\ \mathbf{with}\ h)}$$

Reduction $\boxed{e \longmapsto e'}$

$$\frac{\mathrm{op}\ \boldsymbol{\beta}^{\mathcal{J}} : \boldsymbol{K}^{\mathcal{J}}\ p\ k \mapsto e \in h \quad v_{cont} = \lambda z.\mathbf{handle}_{l\,S^{I}}\ E[z]\ \mathbf{with}\ h \quad 0\text{–free}(l, E)}{\mathbf{handle}_{l\,S^{I}}\ E[\mathrm{op}_{l\,S'^{I}}\ \boldsymbol{T}^{\mathcal{J}}\ v]\ \mathbf{with}\ h \longmapsto e[\boldsymbol{T}^{\mathcal{J}}/\boldsymbol{\beta}^{\mathcal{J}}][v/p][v_{cont}/k]}\ \text{R\_HANDLE2'}$$

Fig. 7. Type-erasure semantics.

call. By contrast, in the type-erasure semantics, the handler h_1 will be chosen regardless of A. The type-and-effect system is not changed.

7.2.2 Safety Conditions and Type-and-Effect Safety. To ensure the safety in the type-erasure semantics, we need an additional safety condition.

Definition 7.6 (Safety Condition for Type-Erasure)**.** *The safety condition added for the type-erasure semantics is: (4) If* $(l\,\boldsymbol{S_1}^{I})^{\uparrow} \circledcirc \varepsilon$ *and* $(l\,\boldsymbol{S_2}^{I})^{\uparrow} \odot \varepsilon' \sim \varepsilon$*, then* $\boldsymbol{S_1}^{I} = \boldsymbol{S_2}^{I}$*.*

To understand this condition, assume that an operation of label name l is called with typelike parameters $\boldsymbol{S_1}^{I}$ and some effect ε_1 such that $(l\,\boldsymbol{S_1}^{I})^{\uparrow} \circledcirc \varepsilon$ is assigned to the operation call via subtyping. When the operation call is handled by an effect handler for effect label $l\,\boldsymbol{S_2}^{I}$, the typelike parameters $\boldsymbol{S_1}^{I}$ for the operation call and $\boldsymbol{S_2}^{I}$ for the handler must be matched. None of the effect algebras $\mathrm{EA_{Set}}$, $\mathrm{EA_{SimpR}}$, $\mathrm{EA_{ScpR}}$, and $\mathrm{EA_{MSet}}$ presented thus far meets this new condition, and, even worse, they can accept some programs unsafe in the type-erasure semantics.

Theorem 7.7 (Unsafe Effect Algebras in Type-Erasure Semantics)**.** *The effect algebras* $\mathrm{EA_{Set}}$, $\mathrm{EA_{MSet}}$, $\mathrm{EA_{SimpR}}$*, and* $\mathrm{EA_{ScpR}}$ *do not meet safety condition (4). Furthermore, there exists an expression that is well typed under these algebras and gets stuck.*

PROOF. Here we focus on the effect algebra $\mathrm{EA_{Set}}$, but a similar discussion can be applied to the other algebras. Recall that $\odot$ in $\mathrm{EA_{Set}}$ is implemented by the union operation for sets, and therefore it is commutative (i.e., it allows exchanging labels in a set no matter what label names and what type arguments are in the labels). Hence, for example, $\{l\,\mathsf{Int}\} \cup \{l\,\mathsf{Bool}\} \sim_{\mathrm{Set}} \{l\,\mathsf{Bool}\} \cup \{l\,\mathsf{Int}\}$ for a label name l taking one type parameter. It means that $\mathrm{EA_{Set}}$ violates safety condition (4).

To give a program that is typeable under $\mathrm{EA_{Set}}$ but unsafe in the type-erasure semantics, consider the following which uses an effect label $\mathsf{Writer} :: \forall\alpha : \mathbf{Typ}.\{\mathsf{tell} : \alpha \Rightarrow \mathsf{Unit}\}$:

$$\begin{aligned}&\mathbf{handle}_{\mathsf{Writer\,Int}}\ \mathbf{handle}_{\mathsf{Writer\,Bool}}\\ &\quad \mathsf{tell}_{\mathsf{Writer\,Int}}\ 42\\ &\ \mathbf{with}\ \{\,\mathbf{return}\ x \mapsto 0\} \uplus \{\mathsf{tell}\ p\ k \mapsto \mathbf{if}\ p\ \mathbf{then}\ 0\ \mathbf{else}\ 42\}\\ &\mathbf{with}\ \{\,\mathbf{return}\ x \mapsto x\} \uplus \{\mathsf{tell}\ p\ k \mapsto p\}\end{aligned}$$

This program is well typed because

- the operation call $\mathsf{tell}_{\mathsf{Writer\,Int}}\ 42$ can have effect $\{\mathsf{Writer\,Bool}\} \cup \{\mathsf{Writer\,Int}\}$ via subeffecting $\{\mathsf{Writer\,Int}\} \circledcirc \{\mathsf{Writer\,Bool}\} \cup \{\mathsf{Writer\,Int}\}$ (which holds because Writer Int and Writer Bool are exchangeable),
- the inner handling expression is well typed and its effect is $\{\mathsf{Writer\,Int}\}$, and
- the outer one is well typed and its effect is $\{\}$.

Table 2. Comparison of the effect algebras.

	Lift coercions	Adaptable to type-erasure	Multiple effect variables
$\mathrm{EA}_{\mathrm{Set}}$	✗	✓	✓
$\mathrm{EA}_{\mathrm{MSet}}$	✓	✓	✓
$\mathrm{EA}_{\mathrm{SimpR}}$	✗	✓	✗
$\mathrm{EA}_{\mathrm{ScpR}}$	✓	✓	✗

Note that this typing rests on the fact that the inner handler assumes that the argument variable p of its tell clause will be replaced by Boolean values as indicated by the type argument Bool to Writer. However, the variable p will be replaced by integer 42 and the program will get stuck. ■

The proof of Theorem 7.7 relies on the commutativity of $\odot$ in each effect algebra. This observation indicates that an effect algebra with *non*commutative $\odot$ can be safe even in the type-erasure semantics. In fact, the previous work [Biernacki et al. 2019; Leijen 2017, 2018] has given an instance of such an effect algebra. By following it, we can adapt the effect algebras defined thus far to be safe in the type-erasure semantics; we call the effect collections in such effect algebras *erasable*.

Example 7.8 (Erasable Effect Algebras)**.** An effect algebra $\mathrm{EA}_{\mathrm{ESet}}$ for erasable sets is defined similarly to $\mathrm{EA}_{\mathrm{Set}}$. The only difference is that the equivalence relation $\sim_{\mathrm{ESet}}$ of $\mathrm{EA}_{\mathrm{ESet}}$ is defined as $\sim_{\mathrm{Set}}$, but the commutativity rule used in the definition of $\sim_{\mathrm{Set}}$ is replaced with

$$\frac{l_1 \neq l_2}{\{l_1\ \boldsymbol{S_1}^{I_1}\} \uplus \{l_2\ \boldsymbol{S_2}^{I_2}\} \sim_{\mathrm{ESet}} \{l_2\ \boldsymbol{S_2}^{I_2}\} \uplus \{l_1\ \boldsymbol{S_1}^{I_1}\}}$$

which only allows exchanging labels with different names. Effect algebras $\mathrm{EA}_{\mathrm{ESet}}$, $\mathrm{EA}_{\mathrm{EMSet}}$, and $\mathrm{EA}_{\mathrm{EScpR}}$ for erasable sets, multisets, and scoped rows, respectively, are defined similarly.

Theorem 7.9. *The effect algebras* $\mathrm{EA}_{\mathrm{ESet}}$, $\mathrm{EA}_{\mathrm{EMSet}}$, $\mathrm{EA}_{\mathrm{ESimpR}}$, *and* $\mathrm{EA}_{\mathrm{EScpR}}$ *meet safety conditions (1), (2), and (4).*

Note that some equivalence properties holding on nonerasable effect collections do not hold on erasable ones. For instance, $\{\mathsf{Writer\ Int}\} \uplus \{\mathsf{Writer\ Bool}\} \sim \{\mathsf{Writer\ Bool}\} \uplus \{\mathsf{Writer\ Int}\}$ and $\rho_1 \uplus \rho_2 \sim \rho_2 \uplus \rho_1$ do not hold in erasable sets. The latter equivalence is not allowed because ρ_1 and ρ_2 may be replaced with, e.g., $\{\mathsf{Writer\ Int}\}$ and $\{\mathsf{Writer\ Bool}\}$, respectively. This limitation could be relaxed by supporting qualified types [Jones 1992].

Finally, we can prove the type-and-effect safety of λ_{EA} with the type-erasure semantics as Theorem 5.9 provided that an effect algebra meets safety conditions (1), (2), and (4).

Theorem 7.10 (Type-and-Effect Safety)**.** *Assume that a given effect algebra meets safety conditions (1), (2), and (4). If* $\emptyset \vdash e : A \mid \mathbb{0}$ *and* $e \longrightarrow^* e' \not\longrightarrow$, *then* $e' = v$ *for some* v.

7.3 Mixing Lift Coercions and Type-Erasure Semantics

It is easy to extend λ_{EA} with both lift coercions and type-erasure semantics and prove its type-and-effect safety if a given effect algebra is assumed to meet safety conditions (1)–(4). Among the effect algebras presented in the paper, only $\mathrm{EA}_{\mathrm{EScpR}}$ satisfies these conditions, and so λ_{EA} instantiated with it is type-and-effect safe. See the supplementary material for the detail of the combination.

8 Comparison of Effect Algebras

In this section, we discuss how different the effect algebras $\mathrm{EA}_{\mathrm{Set}}$, $\mathrm{EA}_{\mathrm{MSet}}$, $\mathrm{EA}_{\mathrm{SimpR}}$, and $\mathrm{EA}_{\mathrm{ScpR}}$ are; it is summarized in Table 2. The first column in Table 2 presents whether the effect algebras are safe

in the presence of lift coercions. As shown in Section 7.1, $\mathrm{EA}_{\mathrm{Set}}$ and $\mathrm{EA}_{\mathrm{SimpR}}$ are unsafe and $\mathrm{EA}_{\mathrm{MSet}}$ and $\mathrm{EA}_{\mathrm{ScpR}}$ are safe. The second column indicates whether the effect algebras can be adapted to the type-erasure semantics. As discussed in Section 7.2, none of the compared effect algebras is safe as it is, but all of them become safe if we can admit restricting the commutativity of the concatenation on effect collections. The third column shows whether each effect algebra allows multiple effect variables to appear in one effect collection. While $\mathrm{EA}_{\mathrm{SimpR}}$ and $\mathrm{EA}_{\mathrm{ScpR}}$ disallow it because effect variables can appear only at the end of rows, neither $\mathrm{EA}_{\mathrm{Set}}$ nor $\mathrm{EA}_{\mathrm{MSet}}$ has such a restriction.

Allowing multiple effect variables in one effect collection in $\mathrm{EA}_{\mathrm{Set}}$ and $\mathrm{EA}_{\mathrm{MSet}}$ leads to more powerful abstraction of effect collections. For example, consider a module interface IntSet for integer sets, which is given using $\mathrm{EA}_{\mathrm{Set}}$:

$$\begin{aligned}\exists\alpha : \mathbf{Typ}.\exists\rho : \mathbf{Eff}.\{\quad & \mathit{empty} : \alpha, \quad \mathit{add} : \mathsf{Int} \to_{\{\}} \alpha \to_{\{\}} \alpha, \quad \cdots, \quad \mathit{choose} : \alpha \to_{\rho} \mathsf{Int}, \\ & \mathit{accumulate} : \forall\beta : \mathbf{Typ}.\forall\rho' : \mathbf{Eff}.(\mathsf{Unit} \to_{\rho \cup \rho'} \beta) \to_{\rho'} \beta\,\mathsf{List} \quad \}\end{aligned}$$

In this type, type variable α is an abstract type representing integer sets, and the fields represent the operations on integer sets. The interface IntSet requires modules to implement, in addition to the basic operations on sets (e.g., the empty set *empty* and the addition of integers to sets *add*), two additional functions for nondeterministic computation. The function *choose* nondeterministically chooses one of the elements of a given integer set. Its type $\alpha \to_{\rho} \mathsf{Int}$ says that a call to *choose* causes the abstract effect ρ (and only the *choose* can cause ρ). Thus, the effect ρ expresses that nondeterministic choice has been performed. The other function *accumulate* collects all the results of the nondeterministic computations triggered by *choose*. It takes as an argument a function that may call *choose* to perform nondeterministic choice (it is indicated by the latent effect of the type $\mathsf{Unit} \to_{\rho \cup \rho'} \beta$) and use the chosen integer to compute the final result. The type of the final result is parameterized by type variable β, and *accumulate*'s return type $\beta\,\mathsf{List}$ means that all the final results are accumulated into a list. Even though the argument function may perform the effect ρ for nondeterministic choice, ρ does not appear in the result type of *accumulate*. This is because *accumulate* "handles" the effect ρ. Furthermore, the argument function may perform an additional effect ρ', which is propagated to the caller of *accumulate*. For example, consider the following program:

$$\begin{aligned}&\mathbf{let}\ x = \mathit{add}\,2\,(\mathit{add}\,1\,(\mathit{add}\,0\,(\mathit{add}\,(-1)\,\mathit{empty}))))\ \mathbf{in} \\ &\mathit{accumulate}\,\mathsf{Bool}\,\{\}\,(\lambda y : \mathsf{Unit}.0 < (\mathit{choose}\,x))\end{aligned}$$

This example constructs the set $\{2, 1, 0, -1\}$, computes whether each integer picked up by *choose* from the set is greater than 0, and accumulates the results into a list. Thus, it may return the list $[0 < 2; 0 < 2; 0 < 0; 0 < -1]$, that is, [true; true; false; false]. Note that the type variable β is instantiated with Bool as the argument function returns a Boolean value, and ρ' is instantiated with the empty effect set $\{\}$ as the argument function only performs nondeterministic choice. If an argument function of *accumulate* raises an exception, like:

$$\begin{aligned}&\mathbf{let}\ x = \mathit{add}\,2\,(\mathit{add}\,1\,(\mathit{add}\,0\,(\mathit{add}\,(-1)\,\mathit{empty})))\ \mathbf{in} \\ &\mathit{accumulate}\,\mathsf{Bool}\,\{\mathsf{Exc}\}\,(\lambda y : \mathsf{Unit}.\mathbf{let}\ z = \mathit{choose}\,x\ \mathbf{in}\ \mathbf{if}\ z < 0\ \mathbf{then}\ \mathsf{raise}_{\mathsf{Exc}}\,\mathsf{Bool}\,()\ \mathbf{else}\ 0 < z)\end{aligned}$$

then the effect variable ρ' is instantiated with $\{\mathsf{Exc}\}$ as the argument function calls the operation raise of Exc, and the evaluation of this program would result in carrying out raise because the set x contains -1.

The type interface IntSet exploits the benefit of $\mathrm{EA}_{\mathrm{Set}}$ that multiple effect variables can appear in one effect collection. First, it is noteworthy that abstracting the concrete effect for nondeterministic choice by the effect variable ρ enables abstracting module implementations over not only what effect labels are used in the implementations but also *how many labels are used there*; we provide certain implementations of IntSet with different numbers of effect labels in the supplementary

material. Furthermore, IntSet allows argument functions of *accumulate* to perform any effect ρ' besides ρ by unifying ρ and ρ' via $\cup$. Neither the effect algebra $\mathrm{EA}_{\mathrm{SimpR}}$ nor $\mathrm{EA}_{\mathrm{ScpR}}$ allows this unification because only one effect variable may appear in a row.

However, this is not the end of the story: some existing works have discussed benefits of using rows as effect collections. Hillerström and Lindley [2016] demonstrated that simple rows with row polymorphism in the style of Rémy [1994] are useful to solve the unification occurring in the composition of effect handlers. Leijen [2017] implemented a sound and complete type inference for the effect system with polymorphism by utilizing scoped rows. The current form of our theoretical framework, effect algebras, does not provide a means to discuss unification and type inference for algebraic effects and handlers, and it is left open how we can address it in an abstract manner.

9 Related Work

We have explained the existing effect systems for effect handlers in Section 2, compared some of them with the instances of our effect system in Section 6. We will also discuss what aspect of effect handlers our framework does not support in Section 10. In this section, we discuss the differences between our abstract effect system and the existing generic effect systems proposed to deal with various effects in one framework. We also compare effect algebras with the abstract theory of rows introduced by Morris and McKinna [2019].

Generic effect systems. Although, as far as we know, there is no prior work on abstracting effect systems for effect handlers with nor without algebraic structures, the research on generic effect systems that can reason about the use of a wide range of effects (such as file resource usage, memory usage and management, and exception checking) has been conducted. Marino and Millstein [2009] proposed a monomorphic type-and-effect system that tracks a set of capabilities (or privileges) to perform effectful operations such as memory manipulation and exception raising. Their effect system is generic in that it is parameterized over the forms of capabilities as well as the adjustments and checkings of capabilities per context. It assumes that capabilities are gathered into a set and its typing discipline relies on the set operations (e.g., the subeffecting is implemented by set inclusion). Rytz et al. [2012] generalized Marino and Millstein's effect system by allowing the use of a join semilattice to represent collections of capabilities and introducing effect polymorphism. Join-semilattices are underlying structures of effects in effect systems for *may analysis*. In such a system, the join operation $\sqcup$ and the ordering relation $\sqsubseteq$ in a join semilattice are used to merge multiple effects into one and to introduce effect overapproximation as subeffecting, respectively. As $\sqsubseteq$ can be induced by $\sqcup$ ($x \sqsubseteq y \iff x \sqcup y = y$), we define the subeffecting $\lessdot$ using $\odot$ in an effect algebra ($\varepsilon_1 \lessdot \varepsilon_2 \iff \exists \varepsilon.\, \varepsilon_1 \odot \varepsilon \sim \varepsilon_2$). Thus, the role of $\odot$ is similar to that of $\sqcup$, but $\odot$ is not required to be commutative nor idempotent, unlike $\sqcup$ (note that join operations are characterized by associativity, commutativity, and idempotence). In fact, $\odot$ in the effect algebra $\mathrm{EA}_{\mathrm{ScpR}}$ or $\mathrm{EA}_{\mathrm{MSet}}$ is nonidempotent, which is key to support lift coercions (Section 7.1.2), and $\odot$ in each of the effect algebras being safe in the type-erasure semantics is noncommutative (Section 7.2.2).

Recent developments of generic effect systems have focused on *sequential effect systems* [Atkey 2009; Gordon 2017, 2021; Ivaskovic et al. 2020; Katsumata 2014; Mycroft et al. 2016; Tate 2013], which aim to reason about the properties where the order of effects matters (e.g., whether no closed file will be read nor written). An approach common in the prior work on sequential effect systems is to introduce sequential composition $\triangleright$, an operation to compose effects happening sequentially. For example, given expressions e_1 with effect ε_1 and e_2 with ε_2, the effect of a let-expression $\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2$ is given by $\varepsilon_1 \triangleright \varepsilon_2$. The sequential composition can be characterized as a (partial) monoid. Thus, it might look similar to $\odot$ in an effect algebra, but their roles are significantly different: $\odot$ is used to expand (i.e., overapproximate) effects and remove specific labels from effects, whereas the sequential

composition $\triangleright$ is used to compose the effects of expressions executed sequentially. In fact, if we were to use $\odot$ to sequence effects, the safety of λ_{EA} in the type-erasure semantics would not hold even in the effect algebra $\mathrm{EA}_{\mathrm{EScpR}}$ for erasable scoped rows. For example, assume that an expression $\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2$ is given effect $\varepsilon_1 \odot \varepsilon_2$ if the effects ε_1 and ε_2 are assigned to e_1 and e_2. Then, the expression $e \stackrel{\text{def}}{=} \mathbf{let}\ x = \mathsf{tell}_{\mathsf{Writer\ Bool}}\ \mathsf{true}\ \mathbf{in}\ \mathsf{tell}_{\mathsf{Writer\ Int}}\ 1$ could have the effect $\{\mathsf{Writer\ Bool}\} \sqcup \{\mathsf{Writer\ Int}\}$ under $\mathrm{EA}_{\mathrm{EScpR}}$. Thus, the expression $\mathbf{handle}_{\mathsf{Writer\ Int}}\ (\mathbf{handle}_{\mathsf{Writer\ Bool}}\ e\ \mathbf{with}\ h_1)\ \mathbf{with}\ h_2$ would be well typed (for some appropriate effect handlers h_1 and h_2), although it may get stuck in the type-erasure semantics because the operation call $\mathsf{tell}_{\mathsf{Writer\ Int}}\ 1$ will be handled by the effect handler h_1 for Writer Bool. Readers might wonder why $\odot$ cannot work as a sequential composition despite the fact that join operations, which are also used to overapproximate effects, can. We think that this is because the assumptions on $\odot$ are weaker than those on join operations as discussed above. Making $\odot$ in effect algebras and $\triangleright$ in sequential effect systems coexist is a promising future direction, motivated by the recent study on sequential effect systems for control operators [Gordon 2020; Sekiyama and Unno 2023; Song et al. 2022].

Abstracting rows. Morris and McKinna [2019] proposed an algebraic theory of rows to abstract type systems for extensible data types including records and variants. Morris and McKinna separates the syntactic representations of rows from their models. The former is called a *row theory*, which is a triple of $\langle \mathcal{R}, \sim_{\mathcal{R}}, \Rightarrow \rangle$ where $\mathcal{R}$ is the set of syntactic representations of rows, $\sim_{\mathcal{R}}$ is an equivalence relation on $\mathcal{R}$, and $\Rightarrow$ is an entailment relation on row predicates (specifically, row equivalence and containment). It supposes a row concatenation operation $\odot_{\mathcal{R}}$ and defines row containment using the concatenation operation, as we defined subeffecting using the effect composition operation $\odot$. The latter, the models of rows, are formulated as a partial monoid $\langle \mathcal{M}, \cdot, \epsilon \rangle$, which works as a model of a row theory $\langle \mathcal{R}, \sim_{\mathcal{R}}, \Rightarrow \rangle$ if there is a mapping from $\langle \mathcal{R}, \sim_{\mathcal{R}}, \Rightarrow \rangle$ to the monoid such that the row concatenation operation $\odot_{\mathcal{R}}$ is interpreted by the monoid operation $\cdot$ and the entailment relation $\Rightarrow$ is by the logical implication on the equality on $\mathcal{M}$.

As easily found from the above description, our effect algebras are similar to Morris and McKinna's abstract theory of rows. Roughly speaking, effect algebras can be viewed as ones that unify a row theory with its model, but the unified form helps simplify the definition of effect algebras—we would have to pose some further safety conditions on a mapping from the syntax of effects to their model if we were to separate them. Furthermore, there are a few minor differences between effect algebras and Morris and McKinna's abstract theory of rows. First, effect algebras do not include the entailment of predicates because Morris and McKinna introduced it for implementing qualified types, but we do not support them. Second, effect algebras assume a label injection operation $(-)^{\uparrow}$, which is crucial to state certain safety conditions such as condition (1) in Definition 5.1. Third, effect algebras define an equivalence relation as a binary relation on effects, while Morris and McKinna restricts the use of an row equivalence relation $\sim_{\mathcal{R}}$ to the form $(- \odot_{\mathcal{R}} -) \sim_{\mathcal{R}} -$. The more flexible use of the equivalence relation in effect algebras simplifies the presentation of safety conditions. For example, if we were to adopt an equivalence relation in the ternary style, safety condition (3) introduced for lift coercions in Definition 7.1 would have to be written like:

$$\begin{aligned} & \exists\{\varepsilon'_0, \ldots, \varepsilon'_n\}.(\forall i \in \{0, \ldots, n-1\}.(L_i)^{\uparrow} \odot \varepsilon'_{i+1} \sim \varepsilon'_i) \wedge \\ & \qquad (L)^{\uparrow} \odot \varepsilon_1 \sim \varepsilon'_0 \wedge (L)^{\uparrow} \odot \varepsilon_2 \sim \varepsilon'_n \wedge L \notin \{L_1, \ldots, L_n\} \\ \Longrightarrow\ & \exists\{\varepsilon'_0, \ldots, \varepsilon'_n\}.(\forall i \in \{0, \ldots, n-1\}.(L_i)^{\uparrow} \odot \varepsilon'_{i+1} \sim \varepsilon'_i) \wedge \\ & \qquad \mathbb{0} \odot \varepsilon_1 \sim \varepsilon'_0 \wedge \mathbb{0} \odot \varepsilon_2 \sim \varepsilon'_n\,, \end{aligned}$$

which is more complicated than the one presented in Definition 7.1.

10 Conclusion and Future Work

In this paper, we give λ_{EA} equipped with the abstract effect system that can be instantiated to concrete effect systems, define safety conditions on effect algebras, and prove the type-and-effect safety of λ_{EA} by assuming that a given effect algebra satisfies the conditions. As far as we know, no research formalizes the differences among effect systems for effect handlers nor the requirements for the effect systems to prove safety properties. We reveal these essences via the abstraction of effect systems by effect algebras, and the formalization of the safety conditions. The safety conditions added for lift coercions or type-erasure semantics clarify the differences among effect algebras. In the rest of the paper, we discuss possible directions for future work.

Abstraction of handling mechanisms. Although the framework in the paper targets deep effect handlers, adapting it to shallow effect handlers is easy. In fact, we have provided this adaption and proved its safety under the same safety conditions as the ones given in the main paper; interested readers are referred to the supplementary material. In the literature, there are other proposals of the effect handling, especially for resolving the problem with *accidental handling* without relying on lift coercions. For instance, local effects [Biernacki et al. 2019], tunneling [Zhang and Myers 2019], and lexically scoped effect handlers [Biernacki et al. 2020; Brachthäuser et al. 2020] have been proposed. These approaches can be applied to address the accidental handling, but they employ significantly different styles. For example, lexically scoped effect handlers can enable a new notion of effect polymorphism, called *contextual polymorphism* [Brachthäuser et al. 2020]. Exploring abstraction to accommodate all of these mechanisms is a challenging but interesting direction.

Abstraction for unification and type inference. As mentioned in Section 6, our framework has not yet exposed the essential roles of rows in their main application—unification and type inference. One of our ambitious goals for future research is to give a theoretical framework that can discover differences among effect representations in unification and type-inference, which have been well explored with concrete effect representations, such as sets [Pretnar 2014], simple rows [Hillerström and Lindley 2016], and scoped rows [Leijen 2017], but not in an abstract manner.

Abstraction of constrained effect collections. Another interesting direction is to abstract *constrained* effect collections. For example, Hillerström and Lindley [2016] introduce Rémy's row polymorphism, which can state that some labels are present or absent in row variables, for effective unification and Tang et al. [2024] propose an effect system that allows type abstraction over subtyping constraints on row variables. Row constraints have been extensively studied for programming with records and variants [Cardelli and Mitchell 1989; Harper and Pierce 1991; Jones 1992; Rémy 1994]. Morris and McKinna [2019] proposed a type system which treats rows and constraints on them abstractly. Integrating the idea of their work with our framework is a promising approach.

Abstraction of implementation techniques. One approach to implementing effect handlers is to apply type-directed translation into an intermediate language [Hillerström et al. 2017; Leijen 2017; Schuster et al. 2022; Xie et al. 2020]. Exploring the type-directed translations and optimizations proposed thus far, such as a selective translation into continuation passing style (CPS) [Leijen 2017], in an abstract manner may lead to a common implementation infrastructure for languages with different effect systems or give an insight into the influence of effect representations on efficiency.

Acknowledgments

We thank Takeshi Tsukada for advice at the early stage of the research, and the anonymous reviewers for fruitful comments and suggestions. This work was supported in part by JSPS KAKENHI Grant Numbers JP19K20247, JP20H00582, JP20H05703, JP22K17875, and JP24H00699.

References

Robert Atkey. 2009. Parameterised notions of computation. *J. Funct. Program.* 19, 3-4 (2009), 335–376. https://doi.org/10.1017/S095679680900728X

Andrej Bauer and Matija Pretnar. 2013. An Effect System for Algebraic Effects and Handlers. In *Algebra and Coalgebra in Computer Science - 5th International Conference, CALCO 2013, Warsaw, Poland, September 3-6, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 8089)*, Reiko Heckel and Stefan Milius (Eds.). Springer, 1–16. https://doi.org/10.1007/978-3-642-40206-7_1

Andrej Bauer and Matija Pretnar. 2021. Eff, version 5.1. https://www.eff-lang.org/

Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. 2018. Handle with care: relational interpretation of algebraic effects and handlers. *Proc. ACM Program. Lang.* 2, POPL (2018), 8:1–8:30. https://doi.org/10.1145/3158096

Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. 2019. Abstracting algebraic effects. *Proc. ACM Program. Lang.* 3, POPL (2019), 6:1–6:28. https://doi.org/10.1145/3290319

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, 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

Luca Cardelli and John C. Mitchell. 1989. Operations on Records. In *Mathematical Foundations of Programming Semantics, 5th International Conference, Tulane University, New Orleans, Louisiana, USA, March 29 - April 1, 1989, Proceedings (Lecture Notes in Computer Science, Vol. 442)*, Michael G. Main, Austin Melton, Michael W. Mislove, and David A. Schmidt (Eds.). Springer, 22–52. https://doi.org/10.1007/BFB0040253

Yannick Forster, Ohad Kammar, Sam Lindley, and Matija Pretnar. 2017. On the expressive power of user-defined effects: effect handlers, monadic reflection, delimited control. *Proc. ACM Program. Lang.* 1, ICFP (2017), 13:1–13:29. https://doi.org/10.1145/3110257

D. J. Foulis and M. K. Bennett. 1994. Effect algebras and unsharp quantum logics. *Foundations of Physics* 24, 10 (01 Oct 1994), 1331–1352. https://doi.org/10.1007/BF02283036

Colin S. Gordon. 2017. A Generic Approach to Flow-Sensitive Polymorphic Effects. In *31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain (LIPIcs, Vol. 74)*, Peter Müller (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 13:1–13:31. https://doi.org/10.4230/LIPICS.ECOOP.2017.13

Colin S. Gordon. 2020. Lifting Sequential Effects to Control Operators. In *34th European Conference on Object-Oriented Programming, ECOOP 2020 (LIPIcs, Vol. 166)*, Robert Hirschfeld and Tobias Pape (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 23:1–23:30. https://doi.org/10.4230/LIPIcs.ECOOP.2020.23

Colin S. Gordon. 2021. Polymorphic Iterable Sequential Effect Systems. *ACM Trans. Program. Lang. Syst.* 43, 1 (2021), 4:1–4:79. https://doi.org/10.1145/3450272

Robert Harper and Benjamin C. Pierce. 1991. A Record Calculus Based on Symmetric Concatenation. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages, Orlando, Florida, USA, January 21-23, 1991*, David S. Wise (Ed.). ACM Press, 131–142. https://doi.org/10.1145/99583.99603

Daniel Hillerström and Sam Lindley. 2016. Liberating effects with rows and handlers. In *Proceedings of the 1st International Workshop on Type-Driven Development, TyDe@ICFP 2016, Nara, Japan, September 18, 2016*, James Chapman and Wouter Swierstra (Eds.). ACM, 15–27. https://doi.org/10.1145/2976022.2976033

Daniel Hillerström, Sam Lindley, Robert Atkey, and K. C. Sivaramakrishnan. 2017. Continuation Passing Style for Effect Handlers. In *2nd International Conference on Formal Structures for Computation and Deduction, FSCD 2017, September 3-9, 2017, Oxford, UK (LIPIcs, Vol. 84)*, Dale Miller (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 18:1–18:19. https://doi.org/10.4230/LIPIcs.FSCD.2017.18

Andrej Ivaskovic, Alan Mycroft, and Dominic Orchard. 2020. Data-Flow Analyses as Effects and Graded Monads. In *5th International Conference on Formal Structures for Computation and Deduction, FSCD 2020, June 29-July 6, 2020, Paris, France (Virtual Conference) (LIPIcs, Vol. 167)*, Zena M. Ariola (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 15:1–15:23. https://doi.org/10.4230/LIPICS.FSCD.2020.15

Mark P. Jones. 1992. A Theory of Qualified Types. In *ESOP '92, 4th European Symposium on Programming, Rennes, France, February 26-28, 1992, Proceedings (Lecture Notes in Computer Science, Vol. 582)*, Bernd Krieg-Brückner (Ed.). Springer, 287–306. https://doi.org/10.1007/3-540-55253-7_17

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

Ohad Kammar and Matija Pretnar. 2017. No value restriction is needed for algebraic effects and handlers. *J. Funct. Program.* 27 (2017), e7. https://doi.org/10.1017/S0956796816000320

Shin-ya Katsumata. 2014. Parametric effect monads and semantics of effect systems. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, Suresh Jagannathan and Peter Sewell (Eds.). ACM, 633–646. https://doi.org/10.1145/2535838.2535846

Fuga Kawamata, Hiroshi Unno, Taro Sekiyama, and Tachio Terauchi. 2024. Answer Refinement Modification: Refinement Type System for Algebraic Effects and Handlers. *Proc. ACM Program. Lang.* 8, POPL (2024), 115–147. https://doi.org/10.1145/3633280

Daan Leijen. 2017. Type directed compilation of row-typed algebraic effects. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 486–499. https://doi.org/10.1145/3009837.3009872

Daan Leijen. 2018. *Algebraic Effect Handlers with Resources and Deep Finalization.* Technical Report MSR-TR-2018-10. 35 pages. https://www.microsoft.com/en-us/research/publication/algebraic-effect-handlers-resources-deep-finalization/

Daan Leijen. 2024. Koka: a Functional Language with Effects, version 3.1.0. https://koka-lang.github.io/

Sam Lindley, Daniel Hillerström, Simon Fowler, James Cheney, Jan Stolarek, Frank Emrich, Rudi Horn, Vashti Galpin, Wilmer Ricciotti, and Philip and Wadler. 2023. Links: Linking Theory to Practice for the Web, version 0.9.8. https://links-lang.org/

Daniel Marino and Todd D. Millstein. 2009. A generic type-and-effect system. In *Proceedings of TLDI'09: 2009 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, Savannah, GA, USA, January 24, 2009*, Andrew Kennedy and Amal Ahmed (Eds.). ACM, 39–50. https://doi.org/10.1145/1481861.1481868

Eugenio Moggi. 1991. Notions of Computation and Monads. *Inf. Comput.* 93, 1 (1991), 55–92. https://doi.org/10.1016/0890-5401(91)90052-4

J. Garrett Morris and James McKinna. 2019. Abstracting extensible data types: or, rows by any other name. *Proc. ACM Program. Lang.* 3, POPL (2019), 12:1–12:28. https://doi.org/10.1145/3290325

Alan Mycroft, Dominic A. Orchard, and Tomas Petricek. 2016. Effect Systems Revisited - Control-Flow Algebra and Semantics. In *Semantics, Logics, and Calculi - Essays Dedicated to Hanne Riis Nielson and Flemming Nielson on the Occasion of Their 60th Birthdays (Lecture Notes in Computer Science, Vol. 9560)*, Christian W. Probst, Chris Hankin, and René Rydhof Hansen (Eds.). Springer, 1–32. https://doi.org/10.1007/978-3-319-27810-0_1

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. 2009. Handlers of Algebraic Effects. In *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5502)*, Giuseppe Castagna (Ed.). Springer, 80–94. https://doi.org/10.1007/978-3-642-00590-9_7

Gordon D. Plotkin and Matija Pretnar. 2013. Handling Algebraic Effects. *Log. Methods Comput. Sci.* 9, 4 (2013). https://doi.org/10.2168/LMCS-9(4:23)2013

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

Matija Pretnar. 2015. An Introduction to Algebraic Effects and Handlers. Invited tutorial paper. In *The 31st Conference on the Mathematical Foundations of Programming Semantics, MFPS 2015, Nijmegen, The Netherlands, June 22-25, 2015 (Electronic Notes in Theoretical Computer Science, Vol. 319)*, Dan R. Ghica (Ed.). Elsevier, 19–35. https://doi.org/10.1016/j.entcs.2015.12.003

Lukas Rytz, Martin Odersky, and Philipp Haller. 2012. Lightweight Polymorphic Effects. In *ECOOP 2012 - Object-Oriented Programming - 26th European Conference, Beijing, China, June 11-16, 2012. Proceedings (Lecture Notes in Computer Science, Vol. 7313)*, James Noble (Ed.). Springer, 258–282. https://doi.org/10.1007/978-3-642-31057-7_13

Didier Rémy. 1994. Type Inference for Records in a Natural Extension of ML. In *Theoretical Aspects Of Object-Oriented Programming. Types, Semantics and Language Design*, Carl A. Gunter and John C. Mitchell (Eds.). MIT Press.

Amr Hany Saleh, Georgios Karachalias, Matija Pretnar, and Tom Schrijvers. 2018. Explicit Effect Subtyping. In *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 10801)*, Amal Ahmed (Ed.). Springer, 327–354. https://doi.org/10.1007/978-3-319-89884-1_12

Philipp Schuster, Jonathan Immanuel Brachthäuser, Marius Müller, and Klaus Ostermann. 2022. A typed continuation-passing translation for lexical effect handlers. In *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*, Ranjit Jhala and Isil Dillig (Eds.). ACM, 566–579. https://doi.org/10.1145/3519939.3523710

Taro Sekiyama and Atsushi Igarashi. 2019. Handling Polymorphic Algebraic Effects. In *Programming Languages and Systems - 28th European Symposium on Programming, ESOP 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11423)*, Luís Caires (Ed.). Springer, 353–380. https://doi.org/10.1007/978-3-030-17184-1_13

Taro Sekiyama, Takeshi Tsukada, and Atsushi Igarashi. 2020. Signature restriction for polymorphic algebraic effects. *Proc. ACM Program. Lang.* 4, ICFP (2020), 117:1–117:30. https://doi.org/10.1145/3408999

Taro Sekiyama and Hiroshi Unno. 2023. Temporal Verification with Answer-Effect Modification: Dependent Temporal Type-and-Effect System with Delimited Continuations. *Proc. ACM Program. Lang.* 7, POPL, Article 71 (2023), 32 pages. https://doi.org/10.1145/3571264

Yahui Song, Darius Foo, and Wei-Ngan Chin. 2022. Automated Temporal Verification for Algebraic Effects. In *Programming Languages and Systems - 20th Asian Symposium, APLAS 2022, Auckland, New Zealand, December 5, 2022, Proceedings (Lecture Notes in Computer Science, Vol. 13658)*, Ilya Sergey (Ed.). Springer, 88–109. https://doi.org/10.1007/978-3-031-21037-2_5

Wenhao Tang, Daniel Hillerström, Sam Lindley, and J. Garrett Morris. 2024. Soundly Handling Linearity. *Proc. ACM Program. Lang.* 8, POPL (2024), 1600–1628. https://doi.org/10.1145/3632896

Ross Tate. 2013. The sequential semantics of producer effect systems. In *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, Roberto Giacobazzi and Radhia Cousot (Eds.). ACM, 15–26. https://doi.org/10.1145/2429069.2429074

Mads Tofte. 1990. Type Inference for Polymorphic References. *Inf. Comput.* 89, 1 (1990), 1–34. https://doi.org/10.1016/0890-5401(90)90018-D

Philip Wadler. 1998. The Marriage of Effects and Monads. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming (ICFP '98), Baltimore, Maryland, USA, September 27-29, 1998*, Matthias Felleisen, Paul Hudak, and Christian Queinnec (Eds.). ACM, 63–74. https://doi.org/10.1145/289423.289429

Andrew K. Wright. 1995. Simple Imperative Polymorphism. *Lisp and Symbolic Computation* 8, 4 (1995), 343–355.

Andrew K. Wright and Matthias Felleisen. 1994. A Syntactic Approach to Type Soundness. *Inf. Comput.* 115, 1 (1994), 38–94. https://doi.org/10.1006/inco.1994.1093

Ningning Xie, Jonathan Immanuel Brachthäuser, Daniel Hillerström, Philipp Schuster, and Daan Leijen. 2020. Effect handlers, evidently. *Proc. ACM Program. Lang.* 4, ICFP (2020), 99:1–99:29. https://doi.org/10.1145/3408981

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

Received 2024-02-28; accepted 2024-06-18