



# Sound and Complete Type Inference for Closed Effect Rows

Kazuki Ikemori<sup>1</sup>(✉), Youyou Cong<sup>1</sup>, Hidehiko Masuhara<sup>1</sup>, and Daan Leijen<sup>2</sup>

<sup>1</sup> Tokyo Institute of Technology, Tokyo, Japan

ikemori.k.aa@prg.is.titech.ac.jp, cong@c.titech.ac.jp, masuhara@acm.org

<sup>2</sup> Microsoft Research, Seattle, USA

daan@microsoft.com

**Abstract.** Koka is a functional programming language that has algebraic effect handlers and a row-based effect system. The row-based effect system infers types by naively applying the Hindley-Milner type inference. However, it infers effect-polymorphic types for many functions, which are hard to read by the programmers and have a negative runtime performance impact to the evidence-passing translation. In order to improve readability and runtime efficiency, we aim to infer *closed* effect rows when possible, and *open* those closed effect rows automatically at instantiation to avoid loss of typability. This paper gives a type inference algorithm with the *open* and *close* mechanisms. In this paper, we define a type inference algorithm with the open and close constructs.

**Keywords:** Algebraic effects and handlers · Type inference

## 1 Introduction

Koka [12,21] is a functional programming language that has algebraic effects and handlers [7,15] which are a recently introduced abstraction of computational effects. An important aspect<sup>1</sup> of Koka is that it tracks the (side) effects of functions in their type. For example, the following function:

```
fun sqr( x : int ) : ⟨⟩ int
  x * x
```

has an empty effect row type  $\langle \rangle$  as it has no side effect at all. In contrast, a function like:

```
fun head( xs : list⟨a⟩ ) : ⟨exn⟩ a
  match xs
  Cons(x,xx) → x
```

---

<sup>1</sup> Koka is the first practical language that tracks the (side) effects of functions in their type. .

gets the  $\langle \text{exn} \rangle$  effect row type as it may raise an exception at runtime (if we pass an empty list). To track effects, Koka uses row-based effect types [9]. These are quite suitable to combine with standard Hindley-Milner type inference as row equality has decidable unification (unlike subtyping for example).

When doing straightforward Hindley-Milner type inference with row-based effect types many functions will be polymorphic in the tail of the effect row (we call such effect rows *open*). For example, the naively inferred types of the previous two functions would be:

```
fun sqr : int → e int
fun head: list(a) → ⟨exn|e⟩ a
```

Observe that both types are polymorphic in the effect tail as effect variable  $e$ . These are in a way the natural types, signifying for example that we can use `sqr` in any effectful context. However, in practice, we prefer closed types instead, as these are easier to explain, and easier to read and write without needing to always consider the polymorphic tail.

Moreover, it is possible to generate more efficient code for functions with closed effect rows. When executing an effect operation (which is similar to raising an exception), there is generally a dynamic search at runtime for a corresponding handler of that effect. This can be expensive, and Koka uses *evidence passing* [20, 21] to pass handler information as a vector at runtime. When an effect row is closed, the runtime shape of the vector is statically determined, and instead of searching for a handler, we can select the right handler at a fixed offset in the vector. This can be much more efficient.

Therefore, the type inference algorithm in the current Koka compiler generally infers closed effect rows for function bindings, where it relies on two mechanisms, `open` and `close`, for converting between open and closed function types. However, the opening and closing features of the inference algorithm have never been formalized.

In this paper, we make the following contributions:

- We formalize type inference with the `open` and `close` mechanisms precisely (Sect. 4).
- We prove that the type inference algorithm is sound and complete (Sect. 5) and infers most general types.

First, we give an introduction to algebraic effects and handlers, and explain what kind of types we would like to infer (Sect. 2). Next, we present an implicitly typed calculus with algebraic effects and handlers, and give a set of declarative type inference rules (Sect. 3). Then, we define syntax-directed type inference rules and show they are sound and complete with respect to the declarative rules (Sect. 4). We also define a type inference algorithm and prove its soundness and completeness (Sect. 5). Lastly, we discuss related work (Sect. 6) and conclude with future directions (Sect. 7).

## 2 Motivation

### 2.1 Algebraic Effects and Handlers

Algebraic effects and handlers [7, 15] are a uniform mechanism for representing computational effects. When programming with algebraic effects and handlers, the user first declares an effect with a series of *operations*. They then define a handler that specifies the meaning of operations, using the continuation (`resume` in Koka) surrounding the operations. As an example, let us implement a reader effect in Koka.

```
effect read2
  ask-int() : int
  ask-bool(): bool
```

The reader effect `read2` has two operations `ask-int` and `ask-bool`, which take no argument and return an integer and a boolean, respectively. Below is a program that uses the two operations.

```
handler {
  ask-int() { resume(12) }
  ask-bool() { resume(True) }
} {
  if ask-bool() then ask-int() else 42
}
```

In the above program, the conditional expression is surrounded by a handler that specifies the meaning of `ask-int` and `ask-bool`. The evaluation goes as follows. First, `ask-bool()` is interpreted by the second clause of the handler, which says: continue (`resume`) evaluation with the value `True`. Second, the conditional expression reduces to the `then`-clause. Third, `ask-int()` is interpreted by the first clause of the handler, which says: continue evaluation with the value `12`. Thus, the program evaluates to `12`.

### 2.2 Naive Hindley-Milner Type Inference with Algebraic Effects and Handlers

Koka employs a row-based effect system similar to a record type system [9]. It is also equipped with polymorphic type inference, which is similar to the Hindley-Milner type inference but has an additional mechanism for manipulating effects.

It turns out that the types inferred by a natural extension of the Hindley-Milner inference are not suitable for evidence passing [20, 21]. As an example, consider the function `f` below.

```
fun f(x)
  ask-int()
  x
```

This function `f` is inferred to have the type `forall⟨a, e1⟩ a → ⟨read2 | e1⟩ a`. The type contains an effect variable `e1`, representing the effects of the function

body. Now, suppose we have the following user program (where `head` may raise an exception).

```
if True then f(1) else head([1])
```

As the two branches must have the same effect row,  $\langle \text{read}_2 | e_1 \rangle$  and  $\langle \text{exn} | e_2 \rangle$  are unified to  $\langle \text{exn}, \text{read}_2 | e_3 \rangle$  or  $\langle \text{read}_2, \text{exn} | e_3 \rangle$ , where  $e_3$  is a fresh effect variable. In the former case, the Koka compiler has to dynamically search for the `read2` handler because the effect row of `f` has been changed from  $\langle \text{read}_2 | e_2 \rangle$  to  $\langle \text{exn}, \text{read}_2 | e_3 \rangle$ . In the latter case, the Koka compiler would dynamically search for the `exn` handler because the effect row of `head` has been changed from  $\langle \text{exn} | e_2 \rangle$  to  $\langle \text{read}_2, \text{exn} | e_3 \rangle$ . To avoid dynamic search, we would like to infer the type `forall(a) a → ⟨read2⟩ a` for `f`. We call this function type *closed*, in the sense that we cannot grow the effect row by instantiating the effect variable. When `f` is given this closed function type, we know that `f` only performs the `read2` effect. Therefore, we can obtain a corresponding handler in constant time (see [20, 21] for more details). Intuitively, higher order functions such as `map` must have a effect variable and need to dynamically search for the corresponding handler because the Koka compiler cannot statically determine the effects of function body at runtime. The other functions such as `f` should not have a effect variable in order to generate the efficient code by evidence passing, because the set of effects of function body can be statically determined.

The Hindley-Milner type inference also occasionally yields type signatures that are more general than what the user may expect. Consider the identity function.

```
fun id( x )
  x
```

It is likely that the user defines `id` as a function of the following type.

```
fun id( x : a ): ⟨⟩ a
  x
```

Notice that the effect of function body is an empty row  $\langle \rangle$ . This is because the body of `id` is variable, which has no effect. However, based on the Hindley-Milner inference, `id` is given type `forall(a, e) a → e a`. Here, `e` is an effect variable representing *any* effects. When the user is shown this type, they might be surprised because they did not introduce the effect variable `e`. In contrast, the type `forall(a) a → ⟨⟩ a`, which has a total effect  $\langle \rangle$ , seems more natural.

We might take other approaches to show the precise type signatures, but we believe our approach is useful for users. For example, we could treat all effect rows as open ones by implicitly inserting an effect variable. This approach is adopted in the Frank language; we will provide more details in Sect. 6.

### 2.3 Type Inference with open and close

In order to optimize more functions and display precise types, the Koka compiler manipulates effect types using special mechanisms `open` and `close`. In this paper,

we formalize type inference with these mechanisms. The key principle is to close the effect row of all *named* functions (bound by the `let/val` expression), and open the effect row when we encounter variables of a closed function type. Let us illustrate how `open` and `close` work through an example.

```
val id = fn(x) x
id( ask-int() )
```

In the type system described in this paper, the function `fn(x) x` is given the most general type  $\text{forall}\langle a, e \rangle a \rightarrow e\ a$ . When the function is bound to `id`, the type is *closed* to  $\text{forall}\langle a \rangle a \rightarrow \langle \rangle a$ . This ensures that, when the type of a named function is displayed to the user, it must be of the closed form. When `id` is instantiated in the body, the type is *opened* again, first to  $\text{forall}\langle a, e \rangle a \rightarrow e\ a$  and then unified to  $\text{int} \rightarrow \langle \text{read}_2 \rangle \text{int}$  so that `id( ask-int() )` is well-typed in its context.

In general, `open` introduces a fresh effect variable  $e$  into a closed effect row  $\langle l_1, \dots, l_n \rangle$ , yielding  $\langle l_1, \dots, l_n \mid e \rangle$ . Dually, `close` removes an effect variable from an open effect row. Together these allow us to avoid dynamic handler search and display simplified type signatures.

The reader may find opening and closing similar to subtyping. We use these mechanisms to avoid complex constraints over the subtyping relation and undecidability of the unification algorithm. In the following sections, we will show simple type inference for `open` and `close`.

### 3 Implicitly Typed Calculus for Algebraic Effects and Handlers

In this section, we present `ImpKoka`, an implicitly typed surface language that has algebraic effects and handlers, as well as polymorphism and higher-order kinds à la System  $F^\omega$ . The structure of this section is as follows. First, we define the syntax of kinds, types, and effect rows (Sects. 3.1.1–3.1.3). Next, we define the syntax of expressions (Sect. 3.1.4). Lastly, we give a set of declarative type inference rules (Sect. 3.2).

Note that we do not define an operational semantics for `ImpKoka`, but instead define a translation from `ImpKoka` to an extension of System  $F^\epsilon$  [20]. The definition can be found in the appendix.

#### 3.1 Syntax

**3.1.1 Kinds and Types** We define the syntax of kinds and types of `ImpKoka` in Fig. 1. Similar to System  $F^\epsilon$  [20], we have kinds for value types ( $*$ ), type constructors ( $k \rightarrow k$ ), effect rows (`eff`), and effect labels (`lab`). Differently from System  $F^\epsilon$ , we distinguish between monotypes and type schemes. Monotypes consist of type variables  $\alpha^k$ , type constructors  $c^k\ \tau \dots \tau$ , and function types  $\tau \rightarrow \epsilon\ \tau$ . In particular,  $s^k$  is a special type constructors used in a unification function for operations, and the detail of  $s^k$  will be explained in Sect. 5.2. Note that type

Monotypes	MType	$\ni \tau$	$::= \alpha^k \mid \tau \rightarrow \epsilon \tau \mid c^k \tau \dots \tau$
Types	Type	$\ni \sigma$	$::= \tau \mid \forall \bar{\alpha}^k. \tau$
Kinds	Kind	$\ni k$	$::= * \mid k \rightarrow k \mid \text{eff} \mid \text{lab}$
Type Constructors	Con	$\ni c^k$	$::= \langle \rangle \mid \langle \_ \mid \_ \rangle \mid c^{\text{lab}} \mid s^k$
Effect Rows	Eff	$\ni \epsilon$	$::= \langle \rangle \mid \alpha^{\text{eff}} \mid \langle l \mid \epsilon \rangle$
Kind Context	KCtx	$\ni \Xi$	$::= \emptyset \mid \Xi, \alpha^k$
Type Context	TCtx	$\ni \Gamma, \Delta$	$::= \emptyset \mid \Gamma, x : \sigma$
Substitution	Subst	$\ni \theta$	$::= \emptyset \mid \theta[\alpha^k := \tau]$
Effect signature		$sig$	$::= \{ op_i : \forall \bar{\alpha}_i^k. \tau_1^i \rightarrow \epsilon \tau_2^i \}$
Effect signatures		$\Sigma$	$::= \{ l_i : sig_i \}$
Syntax Convention		$\langle l_1, \dots, l_n \rangle$	$\doteq \langle l_1 \mid \dots \mid l_n \mid \langle \rangle \dots \rangle$
		$\langle l_1, \dots, l_n \mid \mu \rangle$	$\doteq \langle l_1 \mid \dots \mid l_n \mid \mu \dots \rangle$
		$\mu ::= \alpha^{\text{eff}}, \quad l ::= c^{\text{lab}}$	

**Fig. 1.** Types, kinds, effect signatures, and effect rows of ImpKoka

variables and constructors have a kind annotation  $k$ . The effect  $\epsilon$  in a function type  $\tau \rightarrow \epsilon \tau$  represents the effect performed by the function body. We will use  $\alpha$  and  $\beta$  for value type variables and  $\mu$  for effect type variables, often without kind annotations.

**3.1.2 Signatures** We define operations  $op$ , effect signatures  $sig$ , and sets of effect signatures  $\Sigma$  again as in System  $F^\epsilon$  (Fig. 1). An effect signature is a set of pairs of an operation name and a type. A set of effect signatures associates each effect label  $l_i$  with a corresponding effect signature  $sig_i$ . We assume that operation names and effect labels are all unique, and that  $\Sigma$  is defined at the top level.

**3.1.3 Effect Rows** In ImpKoka, we use effect rows [5, 11] to represent a collection of effects to be performed by an expression. As in System  $F^\epsilon$ , an effect row is either an empty row  $\langle \rangle$ , or an effect variable  $\mu$ , or an extension  $\langle l \mid \epsilon \rangle$  of an effect row  $\epsilon$  with an effect label  $l$ , respectively. For example, assuming  $\text{exn}$  is an effect label representing exceptions,  $\langle \text{exn}, \text{read}_2 \rangle$  is an effect row representing a collection of exception and reader effects. The kinds of  $\langle \rangle$  and  $l$  are  $\text{eff}$  (representing an effect type) and  $\text{lab}$  (representing an effect label), respectively. The kind of  $\langle \_ \mid \_ \rangle$  is  $\text{lab} \rightarrow \text{eff} \rightarrow \text{eff}$ . We will use  $\langle \rangle$  and  $\langle \_ \mid \_ \rangle$  without kind annotations.

The equivalence relation for effect rows is also defined in the same way as in System  $F^\epsilon$  (Fig. 2). The  $[\text{REFL}]$  and  $[\text{EQ-TRANS}]$  rules are the reflexivity and transitivity rules. The  $[\text{EQ-SWAP}]$  rule says that two effect labels  $l_1$  and  $l_2$  can be swapped if they are distinct. The  $[\text{EQ-HEAD}]$  rule tells us that two effect rows are equivalent when their heads and tails are equivalent.

Note that effect rows can have multiple occurrences of the same effect label. For example, we may have  $\langle \text{exn} \rangle$  and  $\langle \text{exn}, \text{exn} \rangle$ , and they are treated as different effect rows. The advantage of this design is that we can define type inference rules in a simple manner, by using only type equivalence.

$$\begin{array}{c}
\frac{}{\epsilon \equiv \epsilon} \text{ [REFL]} \qquad \frac{\epsilon_1 \equiv \epsilon_2 \quad \epsilon_2 \equiv \epsilon_3}{\epsilon_1 \equiv \epsilon_3} \text{ [EQ-TRANS]} \\
\frac{l_1 \neq l_2 \quad \epsilon_1 \equiv \epsilon_2}{\langle l_1, l_2 \mid \epsilon_1 \rangle \equiv \langle l_2, l_1 \mid \epsilon_2 \rangle} \text{ [EQ-SWAP]} \qquad \frac{\epsilon_1 \equiv \epsilon_2}{\langle l \mid \epsilon_1 \rangle \equiv \langle l \mid \epsilon_2 \rangle} \text{ [EQ-HEAD]}
\end{array}$$


---

**Fig. 2.** Equivalence of row types

Expressions	Exp	⊃	$e ::= v$	(value)
			$ee$	(application)
			$\text{let } x = v \text{ in } e$	(bind)
Values	Val	⊃	$v ::= x$	(variable)
			$\lambda x. e$	(function)
			$\text{handler } h$	(effect handler)
			$\text{perform } op$	(operation)
Handlers	Hnd	⊃	$h ::= \{\overline{op_i \rightarrow v_i}\}$	(operation clauses)

---

**Fig. 3.** Expressions of ImpKoka

**3.1.4 Expressions** We define the syntax of expressions of ImpKoka in Fig. 3. In addition to the standard lambda terms, we have let-binding  $\text{let } x = v \text{ in } e$ , handlers  $\text{handler } h$  and operation performing  $\text{perform } op$ . Note that we treat both handler  $h$  and  $\text{perform } op$  as a value. If we want to handle an expression  $e$  with handler  $h$ , we write  $\text{handler } h (\lambda_. e)$  via application. Similarly, if we want to perform an operation  $op$  with argument  $e$ , we write  $\text{perform } op e$  via application.

## 3.2 Declarative Type Inference Rules

We now turn to the declarative type inference rules with **open** and **close** (Fig. 4). Here, we use a typing judgment of the form  $\Xi \mid \Gamma \mid \Delta \vdash e : \sigma \mid \epsilon$ . The judgment states that, under type variable context  $\Xi$  and typing context  $\Gamma$  and  $\Delta$ , expression  $e$  has type  $\sigma$  and performs effect  $\epsilon$ . Among the two contexts,  $\Delta$  is a type assignment for named (i.e., **let**-bound) functions, which inhabit a function type with a closed effect row (we will call such types *closed function types*). The other context  $\Gamma$  is a type assignment for all other variables.

The  $[\text{VAR}]$  and  $[\text{VAROPEN}]$  rules in Fig. 4 are interesting. The  $[\text{VAR}]$  rule concludes with a type  $\sigma$  that comes from either  $\Delta$  or  $\Gamma$ , and an arbitrary effect  $\epsilon$ . Note that, although a variable does not perform any effects, we cannot replace  $\epsilon$  by  $\langle \rangle$ , because we do not have subeffecting rules in ImpKoka. Note also that the rule applies only to variables whose type is not a closed function type. The  $[\text{VAROPEN}]$  rule derives an open function type for a variable of a closed function type. The variable must reside in  $\Delta$ , because we can only open the type of named functions. As an example, we can use  $[\text{VAROPEN}]$  to make  $\text{id}$  ( $\text{perform } ask\text{-int } ()$ ) well-typed, because we can derive  $\Xi \mid \Gamma \mid \Delta \vdash \text{id} : \text{int} \rightarrow \langle \text{read}_2 \rangle \text{int}$  by  $[\text{VAROPEN}]$  and  $\Xi \mid \Gamma \mid \Delta \vdash \text{perform } ask\text{-int } () : \text{int} \mid \langle \text{read}_2 \rangle$  by  $[\text{PERFORM}]$  and  $[\text{APP}]$ .

$\Xi \mid \Gamma \mid \Delta \vdash e : \sigma \mid \epsilon \quad \Xi \mid \Gamma \mid \Delta \vDash h : \tau \mid l \mid \epsilon$
--

  

$$\frac{x : \sigma \in \Gamma, \Delta \quad \Xi \vdash_{\text{wf}} \epsilon : \text{eff} \quad \sigma \text{ not a closed function type}}{\Xi \mid \Gamma \mid \Delta \vdash x : \sigma \mid \epsilon} \quad [\text{VAR}]$$

$$\frac{f : \forall \bar{\alpha}^k. \tau_1 \rightarrow \langle l_1, \dots, l_n \rangle \tau_2 \in \Delta \quad \Xi \vdash_{\text{wf}} \epsilon : \text{eff} \quad \Xi \vdash_{\text{wf}} \epsilon' : \text{eff}}{\Xi \mid \Gamma \mid \Delta \vdash f : \forall \bar{\alpha}^k. \tau_1 \rightarrow \langle l_1, \dots, l_n \mid \epsilon \rangle \tau_2 \mid \epsilon'} \quad [\text{VAROPEN}]$$

$$\frac{\Xi \mid \Gamma, x : \tau_1 \mid \Delta \vdash e : \tau_2 \mid \epsilon \quad \Xi \vdash_{\text{wf}} \tau_1 : * \quad \Xi \vdash_{\text{wf}} \epsilon' : \text{eff}}{\Xi \mid \Gamma \mid \Delta \vdash \lambda x. e : \tau_1 \rightarrow \epsilon \tau_2 \mid \epsilon'} \quad [\text{LAM}] \quad \frac{\Xi \mid \Gamma \mid \Delta \vdash e_1 : \tau_2 \rightarrow \epsilon \tau \mid \epsilon \quad \Xi \mid \Gamma \mid \Delta \vdash e_2 : \tau_2 \mid \epsilon}{\Xi \mid \Gamma \mid \Delta \vdash e_1 e_2 : \tau \mid \epsilon} \quad [\text{APP}]$$

$$\frac{\Xi, \alpha^k \mid \Gamma \mid \Delta \vdash v : \sigma \mid \langle \rangle \quad k \neq \text{lab} \quad \alpha^k \notin \text{ftv}(\Gamma, \Delta)}{\Xi \mid \Gamma \mid \Delta \vdash v : \forall \alpha^k. \sigma \mid \langle \rangle} \quad [\text{GEN}] \quad \frac{\Xi \mid \Gamma \mid \Delta \vdash e : \forall \alpha^k. \sigma \mid \epsilon \quad \Xi \vdash_{\text{wf}} \tau : k}{\Xi \mid \Gamma \mid \Delta \vdash e : \sigma[\alpha^k := \tau] \mid \epsilon} \quad [\text{INST}]$$

$$\frac{\Xi \mid \Gamma \mid \Delta \vdash v_1 : \sigma_1 \mid \langle \rangle \quad \Xi \mid \Gamma \mid \Delta, x : \sigma_1 \vdash e_2 : \tau_2 \mid \epsilon}{\Xi \mid \Gamma \mid \Delta \vdash \text{let } x = v_1 \text{ in } e_2 : \tau_2 \mid \epsilon} \quad [\text{LET}]$$

$$\frac{op : \forall \bar{\alpha}^k. \tau_1 \rightarrow \tau_2 \in \Sigma(l) \quad \bar{\alpha}^k \notin \Xi \quad \Xi \vdash_{\text{wf}} \bar{\tau} : \bar{k} \quad \Xi \vdash_{\text{wf}} \epsilon : \text{eff} \quad \Xi \vdash_{\text{ef}} \epsilon' : \text{eff}}{\Xi \mid \Gamma \mid \Delta \vdash \text{perform } op : [\bar{\alpha}^k := \bar{\tau}](\tau_1 \rightarrow \langle l \mid \epsilon \rangle \tau_2) \mid \epsilon'} \quad [\text{PERFORM}]$$

$$\frac{\Xi \mid \Gamma \mid \Delta \vDash h : \tau \mid l \mid \epsilon \quad \Xi \vdash_{\text{wf}} \epsilon' : \text{eff}}{\Xi \mid \Gamma \mid \Delta \vdash \text{handler } h : ((\langle \rangle \rightarrow \langle l \mid \epsilon \rangle \tau) \rightarrow \epsilon \tau) \mid \epsilon'} \quad [\text{HANDLER}]$$

$$\frac{\bar{\alpha}_i^k \notin \text{ftv}(\epsilon, \tau) \quad op_i : \forall \bar{\alpha}_i^k. \tau_1^i \rightarrow \tau_2^i \in \Sigma(l) \quad \Xi \mid \Gamma \mid \Delta \vdash v_i : \forall \bar{\alpha}_i^k. \tau_1^i \rightarrow \epsilon((\tau_2^i \rightarrow \epsilon \tau) \rightarrow \epsilon \tau) \mid \langle \rangle}{\Xi \mid \Gamma \mid \Delta \vDash \{op_1 \rightarrow v_1, \dots, op_n \rightarrow v_n\} : \tau \mid l \mid \epsilon} \quad [\text{OPS}]$$

**Fig. 4.** Declarative type inference rules

Other type rules in Fig. 4 are standard. The  $[\text{LAM}]$  rule derives a function type for a lambda abstraction. Observe that the effect  $\epsilon$  of the body  $e$  is integrated into the type  $\tau_1 \rightarrow \epsilon \tau_2$  in the conclusion. Note that the lambda-bound variable  $x$  is added to  $\Gamma$ , not  $\Delta$ . Therefore, in the derivation of the body  $e$ , the type of  $x$  cannot be opened using  $[\text{VAROPEN}]$ . The  $[\text{APP}]$  rule requires that the function  $e_1$ , the argument  $e_2$ , and the body of the function have the same effect  $\epsilon$ . The  $[\text{GEN}]$  rule derives a polymorphic type. Similar to the value restriction in ML we only allow values  $v$  [19], which is necessary for soundness of the translation from  $\text{ImpKoka}$  to  $\text{System F}^\epsilon + \text{restrict}$ . The  $[\text{INST}]$  rule is completely standard. The  $[\text{LET}]$  rule is used to bind values with polymorphic types. As in the  $[\text{GEN}]$  rule, the expression being bound is a value, and must have a total effect  $\langle \rangle$ . Notice that bound variable  $x$  is added to the context  $\Delta$ , not  $\Gamma$ . Therefore, in the derivation of the body  $e_2$ , the type of  $x$  can be opened via  $[\text{VAROPEN}]$ .



$\vdash \theta : \Xi \Rightarrow \Xi' \quad \Xi \vdash \sigma_1 \sqsubseteq \sigma_2$	
$\frac{}{\vdash \emptyset : \cdot \Rightarrow \Xi} \text{ [EMPTY]}$	$\frac{\vdash \theta : \Xi_1 \Rightarrow \Xi_2 \quad \Xi_2 \vdash \tau : k}{\vdash \theta[\alpha^k := \tau] : (\Xi_1, \alpha^k) \Rightarrow \Xi_2} \text{ [EXTEND]}$
$\frac{\Xi \vdash_{\text{wf}} \bar{\tau} : \bar{k} \quad \tau_2 = [\bar{\alpha}^{\bar{k}} := \bar{\tau}] \tau_1 \quad \bar{\beta}^{\bar{k}'} \notin \text{ftv}(\forall \bar{\alpha}^{\bar{k}}. \tau_1)}{\Xi \vdash \forall \bar{\alpha}^{\bar{k}}. \tau_1 \sqsubseteq \forall \bar{\beta}^{\bar{k}'}. \tau_2} \text{ [TYPEORDERING]}$	

**Fig. 5.** Type substitution and type ordering

The [PERFORM], [HANDLER] and [OPS] rules are also standard and used for algebraic effects and handlers. The [PERFORM] rule is used to type an operation call. The type in the conclusion is a monotype, which is instantiated by using a sequence of types  $\bar{\tau}$ . The notation  $op : \forall \bar{\alpha}^{\bar{k}}. \tau_1 \rightarrow \tau_2 \in \Sigma(l)$  of the premise means that the operation  $op : \forall \bar{\alpha}^{\bar{k}}. \tau_1 \rightarrow \tau_2$  belongs to the effect signature corresponding to the effect label  $l$ . The [HANDLER] rule is used to type a handler. It takes a thunked computation (*action*) of type  $() \rightarrow \langle l \mid \epsilon \rangle \tau$  and handles the effect  $l$ . The [OPS] rule takes care of operation clauses of a handler. The type of each operation clause is a nested function type of the form  $\forall \bar{\alpha}_i^{\bar{k}_i}. \tau_1^i \rightarrow \epsilon ((\tau_2^i \rightarrow \epsilon \tau) \rightarrow \epsilon \tau)$ , where  $\tau_1^i$  is the input type of the operation, and  $\tau_2^i \rightarrow \epsilon \tau$  is the type of the continuation. The condition  $\bar{\alpha}_i^{\bar{k}_i} \notin \text{ftv}(\epsilon, \tau)$  is necessary for type preservation of the translation from ImpKoka to System  $F^\epsilon + \text{restrict}$ .

## 4 Syntax-Directed Type Inference Rules

In this section, we formalize the syntax-directed type inference rules with `open` and `close`, following [6, 10]. These rules allow us to determine which typing rule to apply to an expression from the syntax. In what follows, we first define type substitution and type ordering, and then elaborate the key cases of the inference rules.

### 4.1 Type Substitution

Figure 5 shows the definition of type substitution, which is inspired by [4]. The judgment  $\vdash \theta : \Xi_1 \Rightarrow \Xi_2$  means substitution  $\theta$  replaces type variables in context  $\Xi_1$  with types well-formed under context  $\Xi_2$ . There are two formation rules for substitutions: [EMPTY] forms an empty substitution, and [EXTEND] extends a substitution  $\theta$  with  $[\alpha^k := \tau]$ . As a convention, we write *id* to mean the identity substitution.

### 4.2 Type Ordering

Figure 5 shows the definition of type ordering, which is similar to that of System F. The judgment  $\Xi \vdash \sigma_1 \sqsubseteq \sigma_2$  means type  $\sigma_2$  is more specific than type  $\sigma_1$

$\Xi \mid \Gamma \mid \Delta \Vdash_{\mathfrak{s}} e : \tau \mid \epsilon \quad \Xi \mid \Gamma \mid \Delta \Vdash_{\mathfrak{s}} h : \tau \mid l \mid \epsilon$
--

  

$$\frac{x : \sigma \in \Gamma, \Delta \quad \Xi \vdash \sigma \sqsubseteq \tau \quad \Xi \Vdash_{\text{wf}} \epsilon : \text{eff}}{\Xi \mid \Gamma \mid \Delta \Vdash_{\mathfrak{s}} x : \tau \mid \epsilon} \quad [\text{VAR}]$$
  

$$\frac{f : \forall \bar{\alpha}^k. \tau_1 \rightarrow \langle l_1, \dots, l_n \rangle \tau_2 \in \Delta \quad \Xi \vdash \forall \bar{\alpha}^k. \tau_1 \rightarrow \langle l_1, \dots, l_n \rangle \tau_2 \sqsubseteq \tau'_1 \rightarrow \langle l_1, \dots, l_n \rangle \tau'_2 \quad \Xi \Vdash_{\text{wf}} \epsilon : \text{eff} \quad \Xi \vdash_{\text{wf}} \epsilon' : \text{eff}}{\Xi \mid \Gamma \mid \Delta \Vdash_{\mathfrak{s}} f : \tau'_1 \rightarrow \langle l_1, \dots, l_n \rangle \epsilon \mid \tau'_2 \mid \epsilon'} \quad [\text{VAROPEN}]$$
  

$$\frac{\Xi \mid \Gamma \mid \Delta \Vdash_{\mathfrak{s}} v_1 : \tau_1 \mid \langle \rangle \quad \Xi \setminus \bar{\alpha}^k \mid \Gamma \mid \Delta, x : \text{Close}(\sigma_1) \Vdash_{\mathfrak{s}} e_2 : \tau_2 \mid \epsilon \quad \sigma_1 = \text{Gen}(\Gamma, \Delta, \tau_1) = \forall \bar{\alpha}^k. \tau_1}{\Xi \mid \Gamma \mid \Delta \Vdash_{\mathfrak{s}} \text{let } x = v_1 \text{ in } e_2 : \tau_2 \mid \epsilon} \quad [\text{LET}]$$
  

$$\frac{op_i : \forall \bar{\alpha}^k_i. \tau_1^i \rightarrow \tau_2^i \in \Sigma(l) \quad \bar{\alpha}^k_i \notin \text{ftv}(\epsilon, \tau) \quad \Xi, \bar{\alpha}^k_i \mid \Gamma \mid \Delta \Vdash_{\mathfrak{s}} v_i : \tau_1^i \rightarrow \epsilon \ ((\tau_2^i \rightarrow \epsilon \tau) \rightarrow \epsilon \tau) \mid \langle \rangle \quad \bar{\alpha}^k_i \notin \text{ftv}(\Gamma, \Delta)}{\Xi \mid \Gamma \mid \Delta \Vdash_{\mathfrak{s}} \{op_1 \rightarrow v_1, \dots, op_n \rightarrow v_n\} : \tau \mid l \mid \epsilon} \quad [\text{OPS}]$$

**Fig. 6.** Syntax-directed type inference rules (excerpt)

under context  $\Xi$ . The context  $\Xi$  is used to inspect the kinds of the types  $\bar{\tau}$  that replace the type variables  $\bar{\alpha}^k$ . For example, the following relationship holds.

$$\begin{aligned} \Xi \vdash \forall \alpha. \alpha \rightarrow \langle \rangle \alpha \sqsubseteq \text{int} \rightarrow \langle \rangle \text{int} \\ \Xi \vdash \forall \alpha \mu. \alpha \rightarrow \mu \alpha \sqsubseteq \forall \beta. (\beta \rightarrow \langle \text{exn} \rangle \beta) \rightarrow \langle \text{exn} \rangle (\beta \rightarrow \langle \text{exn} \rangle \beta) \end{aligned}$$

Using type ordering, we define type equivalence as follows.

$$\sigma_1 = \sigma_2 \Leftrightarrow \sigma_1 \sqsubseteq \sigma_2 \wedge \sigma_2 \sqsubseteq \sigma_1$$

### 4.3 Inference Rules with open and close

Figure 6 is an excerpt of the syntax-directed type inference rules, consisting of those rules that are changed from the declarative rules. Compared to the declarative rules we saw in Sect. 3.2, there are no rules corresponding to  $[\text{GEN}]$  and  $[\text{INST}]$ , because these rules are not syntax directed. All other rules are identical to the declarative rules. Another difference is that we use two auxiliary functions  $\text{Gen}(\cdot, \cdot, \cdot)$  and  $\text{Close}(\cdot)$ . The  $\text{Gen}$  function is the standard generalization function, with a standard definition of free type variables. The  $\text{Close}$  function closes the effect row of a function type. For example,  $\text{Close}(\forall \mu \alpha. \alpha \rightarrow \mu \alpha) = \forall \alpha. \alpha \rightarrow \langle \rangle \alpha$ .

$$\text{Gen}(\Gamma, \Delta, \tau) = \forall (\text{ftv}(\tau) - \text{ftv}(\Gamma, \Delta)). \tau$$

$$\begin{aligned} \text{Close}(\forall \mu \bar{\alpha}^k. \tau_1 \rightarrow \langle l_1, \dots, l_n \mid \mu \rangle \tau_2) &= \forall \bar{\alpha}^k. \tau_1 \rightarrow \langle l_1, \dots, l_n \rangle \tau_2 \quad \text{iff } \mu \notin \text{ftv}(\tau_1, \tau_2) \\ \text{Close}(\sigma) &= \sigma \quad \text{otherwise} \end{aligned}$$

Among the key rules,  $[\text{VAR}]$  is standard and derives the type instantiated by  $\bar{\tau}$ . The  $[\text{VAROPEN}]$  rule derives an open function type from a closed one by inserting an arbitrary effect row. The  $[\text{LET}]$  rule generalizes the type of the bound expression using  $\text{Gen}$ , and derives the type of the body under an extended type context  $\Delta$ ,  $x : \text{Close}(\sigma)$ . In the  $[\text{OPS}]$  rule, we derive a monomorphic type without the type variables bound by the quantifier.

The syntax-directed type inference rules are sound and complete with respect to the declarative type inference rules.

**Theorem 1.** (*syntax-directed inference rules is sound*)

If  $\Xi \mid \Gamma \mid \Delta \Vdash_{\mathfrak{s}} e : \tau \mid \epsilon$  then  $\Xi \mid \Gamma \mid \Delta \vdash e : \tau \mid \epsilon$ .

**Theorem 2.** (*syntax-directed inference rules is complete*)

If  $\Xi \mid \Gamma \mid \Delta \vdash e : \sigma \mid \epsilon$  then  $\Xi' \mid \Gamma \mid \Delta \Vdash_{\mathfrak{s}} e : \tau \mid \epsilon$ ,

where  $\Xi \subseteq \Xi'$  and  $\Xi' \vdash \text{Gen}(\Gamma, \Delta, \tau) \sqsubseteq \sigma$ .

#### 4.4 Principles on the Use of $[\text{LET}]$ Rule

It is important that the  $\text{Close}$  function is applied only in the  $[\text{LET}]$  rule. The reason is that, if  $\text{Close}$  is used to close a function type that is not universally quantified, the type system cannot track the effects to be handled. Let us illustrate the problem using a variation of  $\text{Close}$  and  $[\text{LAM}]$  rule. We first define  $\text{Close}_1$  as a function that closes the effect variable of a monomorphic function type. For example,  $\text{Close}_1(\alpha \rightarrow \mu \alpha) = \alpha \rightarrow \langle \rangle \alpha$ , where  $\Gamma = \emptyset$  and  $\Delta = \emptyset$ .

$\text{Close}_1(\tau_1 \rightarrow \langle l_1, \dots, l_n \mid \mu \rangle \tau_2) = \tau_1 \rightarrow \langle l_1, \dots, l_n \rangle \tau_2$  iff  $\mu \notin \text{ftv}(\tau_1, \tau_2)$   
 $\text{Close}_1(\tau) = \tau$  otherwise

We next define  $[\text{LAM1}]$  as a typing rule that closes the monomorphic function type of a lambda-bound variable using  $\text{Close}_1$  and derives the type of the body under an extended type context  $\Delta$ ,  $x : \tau'_1$ . This allows more functions to have a closed function type as their domain.

$$\frac{\Xi \mid \Gamma \mid \Delta, x : \tau'_1 \Vdash_{\mathfrak{s}} e : \tau_2 \mid \epsilon \quad \tau'_1 = \text{Close}_1(\tau_1)}{\Xi \vdash_{\text{wf}} \tau_1 : * \quad \Xi \vdash_{\text{wf}} \epsilon' : \text{eff}} \quad [\text{LAM1}]$$

$$\Xi \mid \Gamma \mid \Delta \Vdash_{\mathfrak{s}} \lambda x. e : \tau_1 \rightarrow \epsilon \tau_2 \mid \epsilon'$$

With  $[\text{LAM1}]$ , it is possible to derive wrong effects. Consider the following expression.

let  $f = \lambda g. g ()$  in  $f (\lambda \_ . \text{perform } \text{ask-int } ())$

First, we derive  $\emptyset \mid \emptyset \mid \emptyset \Vdash_{\mathfrak{s}} \lambda g. g () : ((\rightarrow \mu \alpha) \rightarrow \langle \rangle \alpha)$  by the  $[\text{LAM1}]$  and  $[\text{VAROPEN}]$  rules. Next, we extend the type context  $\Delta$  with  $f : \forall \mu \alpha. ((\rightarrow \mu \alpha) \rightarrow \langle \rangle \alpha)$  by the  $[\text{LET}]$  rule. Then, we obtain the following derivation by the  $[\text{APP}]$  rule.

$\emptyset \mid \emptyset \mid f : \forall \mu \alpha. ((\rightarrow \mu \alpha) \rightarrow \langle \rangle \alpha) \Vdash_{\mathfrak{s}} f (\lambda \_ . \text{perform } \text{ask-int } ()) : \text{int} \mid \langle \rangle$

This typing judgment is clearly wrong, because it does not track the  $\text{read}_2$  effect. The problem can be avoided if we use `Close` only in the  $[\text{LET}]$  rule: in that case,  $f$  will be given a correct type  $\forall\mu. \alpha \langle () \rightarrow \mu \alpha \rangle \rightarrow \mu \alpha$ .

#### 4.5 Fragility of $[\text{LET}]$ Rule

An unfortunate aspect of our current rules is that the  $[\text{LET}]$  rule is fragile in the sense that insertion of a let-binding may change the typability of programs. Let us consider the following functions:

$$\begin{aligned} \text{remote} &= \lambda f. \text{ perform } \text{ask-bool } () \\ \text{foo} &= \lambda f. \text{ remote } f; f () \end{aligned}$$

In our type system, the type of the function  $\text{remote}$  is inferred as  $\alpha \rightarrow \langle \text{read}_2 \mid \mu \rangle \text{ bool}$ . The function  $\text{foo}$  is also well-typed. First, the lambda-bound variable  $f$  is added to the context  $\Gamma$ , second, the type of  $\text{remote } f$  is inferred as  $\Xi \mid \Gamma \mid \Delta \Vdash_{\mathfrak{s}} \text{remote } f : \text{ bool } \mid \langle \text{read}_2 \mid \mu \rangle$  by the  $[\text{APP}]$  rule, and finally, the type of  $f$  is inferred as  $\Xi \mid \Gamma \mid \Delta \Vdash_{\mathfrak{s}} f : () \rightarrow \langle \text{read}_2 \mid \mu \rangle \beta \mid \langle \text{read}_2 \mid \mu \rangle$  by the  $[\text{APP}]$  rule, where  $e_1; e_2$  is a syntax sugar of  $(\lambda_. e_2) e_1$ . Therefore, the type of function  $\text{foo}$  is inferred as  $(() \rightarrow \langle \text{read}_2 \mid \mu \rangle \beta) \rightarrow \langle \text{read}_2 \mid \mu \rangle \beta$  and  $\text{foo}$  is judged well-typed.

Suppose though that we have a function  $\text{remote}$  that is explicitly annotated with type  $(() \rightarrow \langle \rangle ()) \rightarrow \langle \text{read}_2 \rangle \text{ bool}$ .

$$\begin{aligned} \text{remote} &: (() \rightarrow \langle \rangle ()) \rightarrow \langle \text{read}_2 \rangle \text{ bool} \\ \text{remote} &= \lambda f. \text{ perform } \text{ask-bool } () \end{aligned}$$

$$\begin{aligned} \text{foo} &= \lambda f. \text{ remote } f; f () \\ \text{bar} &= \lambda f. \text{ remote } f; \text{ let } g = f \text{ in } g () \end{aligned}$$

Here  $\text{foo}$  and  $\text{bar}$  only differ in the explicit let-binding for  $f$ . The naive Hindley-Milner type inference without open and close mechanisms rejects both  $\text{foo}$  and  $\text{bar}$ , because a closed function type cannot be opened. In contrast, our inference rules reject  $\text{foo}$  but accept  $\text{bar}$ . The type of  $\text{remote } f$  is inferred as  $\Xi \mid \Gamma \mid \Delta \Vdash_{\mathfrak{s}} \text{remote } f : \text{ bool } \mid \langle \text{read}_2 \rangle$ , and the type of  $f$  is unified to a closed type  $() \rightarrow \langle \rangle ()$  by  $\text{remote}$ . Its type cannot be opened to  $() \rightarrow \langle \text{read}_2 \rangle ()$ , because the lambda-variable  $f$  is included in  $\Gamma$  and cannot be applied the  $[\text{VAROPEN}]$  rule. Hence, we cannot apply the  $[\text{APP}]$  rule to  $f ()$ . On the other hand, the  $\text{bar}$  definition is well-typed. The function  $f$  is now a `let`-bound variable, thus its type can be opened to  $() \rightarrow \langle \text{read}_2 \rangle ()$  by the  $[\text{VAROPEN}]$  rule. Hence, we can apply the  $[\text{APP}]$  rule to  $g ()$ .

This is clearly not desirable and we would like to address this in future work. On the other hand, it is not uncommon to find this form of fragility in practical type systems (like the monomorphism restriction in Haskell, inference for GADT's [14], etc.) and it may work out fine in practice. Experiments on all the standard libraries of Koka (~15000 lines) showed only 2 instances where a let binding was required.

$\begin{aligned} \text{dom} &: \text{Subst} \rightarrow \text{TypeVars} \\ \text{dom}(\emptyset) &= \emptyset \\ \text{dom}(\theta[\alpha^k := \tau]) &= \{\alpha^k\} \cup \text{dom}(\theta) \\ \\ \text{codom} &: \text{Subst} \rightarrow \text{MTypes} \\ \text{codom}(\emptyset) &= \emptyset \\ \text{codom}(\theta[\alpha^k := \tau]) &= \{\tau\} \cup \text{codom}(\theta) \\ \\ (-) &: (\text{Subst}, \text{TypeVars}) \rightarrow \text{Subst} \\ \emptyset - \bar{\alpha}^k &= \emptyset \\ \theta[\beta^k := \tau] - \bar{\alpha}^k &= \theta - \bar{\alpha}^k \quad (\beta^k \in \bar{\alpha}^k) \\ \theta[\beta^k := \tau] - \bar{\alpha}^k &= (\theta - \bar{\alpha}^k)[\beta^k := \tau] \quad (\beta^k \notin \bar{\alpha}^k) \end{aligned}$	$\begin{aligned} \text{const} &: \text{MTypes} \rightarrow \text{Con} \\ \text{const}(\emptyset) &= \emptyset \\ \text{const}(\{s^k\} \cup \bar{\tau}) &= \{s^k\} \cup \text{const}(\bar{\tau}) \\ \text{const}(\{\tau\} \cup \bar{\tau}) &= \text{const}(\bar{\tau}) \\ \\ \text{tail} &: \text{Eff} \rightarrow \text{Eff} \\ \text{tail}(\langle \rangle) &= \langle \rangle \\ \text{tail}(\mu) &= \mu \\ \text{tail}(\langle l \mid \epsilon \rangle) &= \text{tail}(\epsilon) \end{aligned}$
--	--

---

Fig. 7. Auxiliary functions

## 5 Type Inference Algorithm

In this section, we formalize the type inference algorithm with open and close as an extension of Algorithm W [3]. We first define auxiliary functions (Sect. 5.1), and then discuss the unification algorithm (Sect. 5.2) and the type inference algorithm (Sect. 5.3).

### 5.1 Auxiliary Functions

In Fig. 7, we define auxiliary functions. The functions  $\text{dom}(\cdot)$ ,  $\text{codom}(\cdot)$ , and  $\text{tail}(\cdot)$  are completely standard. The  $\text{dom}$  function takes a substitution  $\theta$  and returns a set of type variables that are included in the domain of  $\theta$ . The  $\text{codom}$  function also takes a substitution  $\theta$  and returns a set of monomorphic types that are included in the codomain of  $\theta$ . The  $\text{tail}$  function takes an effect row and returns the tail of the effect row. Note that the function is defined only on non-empty effect rows. The  $(-)$  and  $\text{const}$  functions are used in the  $\text{unifyOp}$  function (Sect. 5.2). The former takes a substitution  $\theta$  and set of type variables  $\bar{\alpha}^k$ , and returns a new substitution with  $\bar{\alpha}^k$  removed from the domain of  $\theta$ . The latter takes a set of monomorphic types, and returns a set of skolem constants.

### 5.2 Unification Algorithm

In Fig. 8, we define the unification algorithm. The algorithm is a natural extension of the standard Robinson unification algorithm [18]. It consists of three functions  $\text{unify}(\cdot, \cdot, \cdot)$ ,  $\text{unifyEffect}(\cdot, \cdot, \cdot)$ , and  $\text{unifyOp}(\cdot, \cdot, \cdot)$ , which take care of value types, effect types, and the type of operation clauses, respectively. Among these functions,  $\text{unify}$  and  $\text{unifyEffect}$  are standard. The  $\text{unifyOp}$  function originates from the special treatment of Koka's operation clauses.

Let us look at the  $\text{unify}$  and  $\text{unifyEffect}$  functions. Given a triple  $(\Xi, \epsilon, l)$  of a type context, an effect row type, and an effect label,  $\text{unifyEffect}$  returns a

$$\begin{aligned}
& \text{unify} : (\text{KCtx} \times \text{MType} \times \text{MType}) \rightarrow (\text{KCtx} \times \text{Subst}) \\
& \text{unify}(\Xi, \alpha^k, \alpha^k) = \text{return}(\Xi, \text{id}) \\
& \text{unify}(\langle \Xi, \alpha^k \rangle, \alpha^k, \tau) = \\
& \quad \text{assert } \Xi \vdash_{\text{wf}} \tau : k \\
& \quad \text{return}(\Xi, \text{id}[\alpha^k := \tau]) \\
& \text{unify}(\langle \Xi, \alpha^k \rangle, \tau, \alpha^k) = \\
& \quad \text{assert } \Xi \vdash_{\text{wf}} \tau : k \\
& \quad \text{return}(\Xi, \text{id}[\alpha^k := \tau]) \\
& \text{unify}(\Xi, \langle \rangle, \langle \rangle) = \text{return}(\Xi, \text{id}) \\
& \text{unify}(\Xi, \langle l \mid \epsilon_1 \rangle, \langle l' \mid \epsilon_2 \rangle) = \\
& \quad \text{let } (\Xi_1, \theta_1, \epsilon_3) = \text{unifyEffect}(\Xi, \langle l' \mid \epsilon_2 \rangle, l) \\
& \quad \text{assert } \text{tail}(\epsilon_1) \notin \text{dom}(\theta_1) \\
& \quad \text{let } (\Xi_2, \theta_2) = \text{unify}(\Xi_1, \theta_1(\epsilon_1), \epsilon_3) \\
& \quad \text{return}(\Xi_2, \theta_2 \circ \theta_1) \\
& \text{unify}(\Xi, \tau_1 \rightarrow \epsilon \tau_2, \tau'_1 \rightarrow \epsilon' \tau'_2) = \\
& \quad \text{let } (\Xi_1, \theta_1) = \text{unify}(\Xi, \tau_1, \tau'_1) \\
& \quad \text{let } (\Xi_2, \theta_2) = \text{unify}(\Xi_1, \theta_1(\epsilon), \theta_1(\epsilon')) \\
& \quad \text{let } (\Xi_3, \theta_3) = \text{unify}(\Xi_2, (\theta_2 \circ \theta_1)(\tau'_2), (\theta_2 \circ \theta_1)(\tau'_2)) \\
& \quad \text{return}(\Xi_3, \theta_3 \circ \theta_2 \circ \theta_1) \\
& \\
& \text{unifyEffect} : (\text{KCtx} \times \text{Eff} \times \text{Lab}) \rightarrow (\text{KCtx} \times \text{Subst} \times \text{Eff}) \\
& \text{unifyEffect}(\Xi, \langle l' \mid \epsilon \rangle, l) = \\
& \quad | l' \equiv l \Rightarrow \text{return}(\Xi, \text{id}, \epsilon) \\
& \quad | l' \neq l \Rightarrow \text{let } (\Xi_1, \theta_1, \epsilon_1) = \text{unifyEffect}(\Xi, \epsilon, l) \\
& \quad \quad \text{return}(\Xi_1, \theta_1, \langle l' \mid \epsilon_1 \rangle) \\
& \text{unifyEffect}(\langle \Xi, \mu \rangle, \mu, l) = \\
& \quad \text{assume } \mu_1 \text{ is fresh} \\
& \quad \text{return}(\langle \Xi, \mu_1 \rangle, \text{id}[\mu := \langle l \mid \mu_1 \rangle], \mu_1) \\
& \\
& \text{unifyOp} : (\text{KCtx} \times \text{Type} \times \text{MType}) \rightarrow (\text{KCtx} \times \text{Subst}) \\
& \text{unifyOp}(\Xi, \forall \bar{\alpha}^k. \tau_1, \tau_2) = \\
& \quad \text{assume } \bar{s}^k \text{ are fresh (skolem constants)} \\
& \quad \text{let } (\Xi_1, \theta_1) = \text{unify}(\Xi, \text{id}[\bar{\alpha}^k := \bar{s}^k]_{\tau_1}, \tau_2) \\
& \quad \text{assert } \bar{s}^k \notin \text{const}(\text{codom}(\theta_1 - \text{ftv}(\tau_2))) \\
& \quad \text{return}(\langle \Xi_1, \bar{\alpha}^k \rangle, [\bar{s}^k \mapsto \bar{\alpha}^k] \circ \theta_1)
\end{aligned}$$


---

**Fig. 8.** Unification algorithm

triple  $(\Xi_1, \theta_1, \epsilon_1)$  of a new type context, a substitution, and an effect row. We can transform  $\epsilon$  into an effect row whose head effect label is  $l$ . As an example, let us consider the following unification problem.

$$\text{unifyEffect}(\Xi, \langle \text{read}_2 \mid \mu \rangle, \text{exn})$$

This succeeds and returns the following effect row and substitution:

$$\epsilon_1 = \langle \text{read}_2 \mid \mu_1 \rangle \quad \theta_1 = \text{id}[\mu := \langle \text{exn} \mid \mu_1 \rangle]$$

The *unifyEffect* function is sound. That is, if  $\text{unifyEffect}(\Xi, \epsilon, l)$  succeeds, it returns a substitution  $\theta_1$  and an effect row  $\epsilon_1$  that satisfy  $\theta_1(\epsilon) \equiv \langle l \mid \theta_1(\epsilon_1) \rangle$ .

**Theorem 3.** (*unifyEffect is sound*)

If  $\Xi \vdash_{\text{wf}} \epsilon : \text{eff}$ ,  $\Xi \vdash_{\text{wf}} l : \text{lab}$  and  $\text{unifyEffect}(\Xi, \epsilon, l) = (\Xi_1, \theta_1, \epsilon_1)$ , then  $\vdash \theta_1 : \Xi \Rightarrow \Xi_1$  and  $\theta_1(\epsilon) \equiv \langle l \mid \theta_1(\epsilon_1) \rangle$ .

The *unifyEffect* function is also complete. That is, if  $\epsilon$  can be rewritten to an effect row of the form  $\langle l \mid \theta(\epsilon') \rangle$  by the substitution  $\theta$ ,  $\text{unifyEffect}(\Xi, \epsilon, l)$  succeeds and returns the most general substitution  $\theta_1$ .

**Theorem 4.** (*unifyEffect is complete*)

If  $\Xi \vdash_{\text{wf}} \epsilon : \text{eff}$ ,  $\Xi \vdash_{\text{wf}} l : \text{lab}$ ,  $\vdash \theta : \Xi \Rightarrow \Xi_2$  and  $\theta(\epsilon) \equiv \langle l \mid \theta(\epsilon') \rangle$ , then  $\text{unifyEffect}(\Xi, \epsilon, l) = (\Xi_1, \theta_1, \epsilon_1)$  and there exists  $\vdash \theta_2 : \Xi_1 \Rightarrow \Xi_2$  such that  $\theta = \theta_2 \circ \theta_1$ .

The *unify* function is similar to *unifyEffect*. Given a triple  $(\Xi, \tau_1, \tau_2)$  of a type context and two monomorphic types, *unify* returns a pair  $(\Xi_1, \theta_1)$  of a new type context and a substitution. The most interesting case of the unification algorithm is  $\text{unify}(\Xi, \langle l \mid \epsilon_1 \rangle, \langle l' \mid \epsilon_2 \rangle)$  that corresponds to [UNI-ROW] case in the [9]. First,  $\text{unifyEffect}(\Xi, \langle l' \mid \epsilon_2 \rangle, l)$  in the first line returns a substitution  $\theta_1$  and an effect row  $\epsilon_3$ , where  $\langle l \mid \epsilon_3 \rangle \equiv \theta_1(\langle l' \mid \epsilon_2 \rangle)$ . Next,  $\text{tail}(\epsilon_1) \notin \text{dom}(\theta_1)$  in the second line confirms that effect variable of  $\epsilon_1$  is not included in the domain of  $\theta_1$ . This is needed to ensure the termination of the unification algorithm. Then,  $\text{unify}(\Xi_1, \theta_1(\epsilon_1), \epsilon_3)$  in the third line returns a substitution  $\theta_2$ , where  $\theta_2(\epsilon_3) \equiv (\theta_2 \circ \theta_1)(\epsilon_1)$ . Thus, we obtain the following equation.

$$(\theta_2 \circ \theta_1)(\langle l \mid \epsilon_1 \rangle) \equiv (\theta_2)(\langle l \mid \epsilon_3 \rangle) \equiv (\theta_2 \circ \theta_1)(\langle l' \mid \epsilon_2 \rangle)$$

The unification algorithm is sound: if  $\text{unify}(\Xi, \tau_1, \tau_2)$  succeeds, it returns the substitution  $\theta_1$  that unifies  $\tau_1$  and  $\tau_2$ .

**Theorem 5.** (*unify is sound*)

If  $\Xi \vdash_{\text{wf}} \tau_1 : k$ ,  $\Xi \vdash_{\text{wf}} \tau_2 : k$ , and  $\text{unify}(\Xi, \tau_1, \tau_2) = (\Xi_1, \theta_1)$ , then  $\vdash \theta_1 : \Xi \Rightarrow \Xi_1$  and  $\theta_1(\tau_1) = \theta_1(\tau_2)$ .

The unification algorithm is also complete: if two types  $\tau_1$  and  $\tau_2$  are unifiable,  $\text{unify}(\Xi, \tau_1, \tau_2)$  succeeds and it returns the most general substitution  $\theta_1$ .

**Theorem 6.** (*unify is complete*)

If  $\Xi \vdash_{\text{wf}} \tau_1 : k$ ,  $\Xi \vdash_{\text{wf}} \tau_2 : k$ ,  $\vdash \theta : \Xi \Rightarrow \Xi_2$  and  $\theta(\tau_1) = \theta(\tau_2)$ , then  $\text{unify}(\Xi, \tau_1, \tau_2) = (\Xi_1, \theta_1)$  and there exists  $\vdash \theta_2 : \Xi_1 \Rightarrow \Xi_2$  such that  $\theta = \theta_2 \circ \theta_1$ .

We now look at the *unifyOp* function. The function takes a triple  $(\Xi, \forall \bar{\alpha}^k. \tau_1, \tau_2)$  of a type variable context, a type scheme, and a monomorphic type  $\tau$ , and returns a pair  $(\Xi_1, \theta_1)$  of a new type context and a substitution. Following [10], we use *skolem constants* to ensure that bound type variables  $\bar{\alpha}^k$  do not escape or occur free in the substitution  $[\overline{s^k} \mapsto \alpha^k] \circ \theta_1$ . The algorithm reads as follows.

We first replace the free type variables  $\bar{\alpha}^k$  of  $\tau_1$  with fresh skolem constants  $\bar{s}^k$ , and unify the resulting type with  $\tau_2$ . When we obtain a substitution  $\theta_1$  as the result, we check the codomain of  $\theta_1 - \text{ftv}(\tau_2)$  does not contain skolem constants  $\bar{s}^k$ , using the `const` function. If this checking succeeds, we construct a new substitution by replacing the skolem constants  $\bar{s}^k$  in  $\theta_1$  back to type variables  $\bar{\alpha}^k$ , and return the resulting substitution. As an example, consider the following unification problem:

$$\text{unifyOp}(\Xi, \forall \alpha. \alpha \rightarrow \text{exn} ((\alpha \rightarrow \langle \text{exn} \rangle \text{int}) \rightarrow \langle \text{exn} \rangle \text{int}), \beta_1 \rightarrow \mu \beta_2)$$

This succeeds and returns the following substitution:

$$[\text{s} \mapsto \alpha] \circ \theta_1 = \text{id}[\beta_1 := \alpha, \beta_2 := ((\alpha \rightarrow \langle \text{exn} \rangle \text{int}) \rightarrow \langle \text{exn} \rangle \text{int}), \mu := \langle \text{exn} \rangle]$$

where  $\theta_1 = \text{id}[\beta_1 := \text{s}, \beta_2 := ((\text{s} \rightarrow \langle \text{exn} \rangle \text{int}) \rightarrow \langle \text{exn} \rangle \text{int}), \mu := \langle \text{exn} \rangle]$ .

The soundness and completeness of `unifyOp` can be proven in a similar way to that of `unify` and `unifyEffect`.

**Theorem 7.** (*unifyOp is sound*)

If  $\Xi \vdash_{\text{wf}} \forall \bar{\alpha}^k. \tau_1 : k$ ,  $\Xi \vdash_{\text{wf}} \tau_2 : k$  and  $\text{unifyOp}(\Xi, \forall \bar{\alpha}^k. \tau_1, \tau_2) = (\Xi_1, \theta_1)$ , then  $\vdash \theta_1 : \Xi \Rightarrow \Xi_1$  and  $\theta_1(\forall \bar{\alpha}^k. \tau_1) = \forall \bar{\alpha}^k. \theta_1(\tau_2)$ .

**Theorem 8.** (*unifyOp is complete*)

If  $\Xi \vdash_{\text{wf}} \forall \bar{\alpha}^k. \tau_1 : k$ ,  $\Xi \vdash_{\text{wf}} \tau_2 : k$ ,  $\vdash \theta : \Xi \Rightarrow \Xi_2$  and  $\theta(\forall \bar{\alpha}^k. \tau_1) = \forall \bar{\alpha}^k. \theta(\tau_2)$ , then  $\text{unifyOp}(\Xi, \forall \bar{\alpha}^k. \tau_1, \tau_2) = (\Xi_1, \theta_1)$  and there exists  $\vdash \theta_2 : \Xi_1 \Rightarrow \Xi_2$  such that  $\theta = \theta_2 \circ \theta_1$ .

### 5.3 Type Inference Algorithm

In Figs.9 and 10, we define the type inference algorithm. The algorithm is an extension of Algorithm W [3] with kinding and a row-based effect system. The functions `infer`( $\cdot, \cdot, \cdot, \cdot$ ) and `inferHandler`( $\cdot, \cdot, \cdot, \cdot$ ) are defined by mutual induction. Given a quadruple  $(\Xi, \Gamma, \Delta, e)$  of a type variable context, two typing contexts, and an expression, `infer` returns a quadruple  $(\Xi, \theta, \tau, \epsilon)$  of a new type variable context, a substitution, a monomorphic type, and an effect row. The effect row  $\epsilon$  represents the effect performed by the expression  $e$ .

Let us go through individual cases. In the variable case, if  $x$  has a closed function type and resides in  $\Delta$ , `infer` yields the most general type by opening the closed effect row to  $\mu_1$ . The `infer` function also yields an arbitrary effect  $\mu_2$  as  $x$  is a value. If  $x$  does not have a closed function type, `infer` generates a new type variable  $\bar{\beta}^k$  as in a standard type inference algorithm.

The abstraction and application cases are standard, except that they involve inference of effect rows.

In the case of a let-expression, the bound expression is generalized by `Gen` and the effect row of the function type is closed by `Close`. Then, the type context



$$\begin{aligned}
& infer : (\text{KCtx} \times \text{TCtx} \times \text{TCtx} \times \text{Exp}) \rightarrow (\text{KCtx} \times \text{Subst} \times \text{MType} \times \text{Eff}) \\
& infer(\Xi, \Gamma, \Delta, x) = \\
& \quad | x : \forall \bar{\alpha}^k. \tau_1 \rightarrow \langle l_1, \dots, l_n \rangle \tau_2 \in \Delta \\
& \quad \quad \Rightarrow \text{assume } \bar{\beta}^k, \mu_1 \text{ and } \mu_2 \text{ are fresh.} \\
& \quad \quad \text{return } ((\Xi, \bar{\beta}^k, \mu_1, \mu_2), id, id[\bar{\alpha}^k := \bar{\beta}^k](\tau_1 \rightarrow \langle l_1, \dots, l_n \mid \mu_1 \rangle \tau_2), \mu_2) \\
& \quad | x : \forall \bar{\alpha}^k. \tau \in \Gamma, \Delta \\
& \quad \quad \Rightarrow \text{assume } \bar{\beta}^k \text{ and } \mu \text{ are fresh.} \\
& \quad \quad \text{return } ((\Xi, \bar{\beta}^k, \mu), id, id[\bar{\alpha}^k := \bar{\beta}^k] \tau, \mu) \\
& infer(\Xi, \Gamma, \Delta, \lambda x. e) = \\
& \quad \text{assume } \alpha^* \text{ and } \mu \text{ are fresh.} \\
& \quad \text{let } (\Xi_1, \theta_1, \tau_1, \epsilon_1) = infer((\Xi, \alpha^*), (\Gamma, x : \alpha^*), \Delta, e) \\
& \quad \text{return } (\Xi_1, \theta_1, \theta_1(\alpha^*) \rightarrow \epsilon_1 \tau_1, \mu) \\
& infer(\Xi, \Gamma, \Delta, e_1 e_2) = \\
& \quad \text{assume } \alpha^* \text{ is fresh.} \\
& \quad \text{let } (\Xi_1, \theta_1, \tau_1, \epsilon_1) = infer(\Xi, \Gamma, \Delta, e_1) \\
& \quad \text{let } (\Xi_2, \theta_2, \tau_2, \epsilon_2) = infer(\Xi_1, \theta_1(\Gamma), \theta_1(\Delta), e_2) \\
& \quad \text{let } (\Xi_3, \theta_3) = unify(\Xi_2, \theta_2(\tau_1), \tau_2 \rightarrow \epsilon_2 \alpha^*) \\
& \quad \text{let } (\Xi_4, \theta_4) = unify(\Xi_3, (\theta_3 \circ \theta_2)(\epsilon_1), \theta_3(\epsilon_2)) \\
& \quad \text{return } (\Xi_4, \theta_4 \circ \theta_3 \circ \theta_2 \circ \theta_1, (\theta_4 \circ \theta_3)(\alpha^*), (\theta_4 \circ \theta_3)(\epsilon_2)) \\
& infer(\Xi, \Gamma, \Delta, \text{let } x = v_1 \text{ in } e_2) = \\
& \quad \text{let } (\Xi_1, \theta_1, \tau_1, \epsilon_1) = infer(\Xi, \Gamma, \Delta, v_1) \\
& \quad \text{let } (\Xi_2, \theta_2) = unify(\Xi_1, \epsilon_1, \langle \rangle) \\
& \quad \text{let } \forall \bar{\alpha}^k. \theta_2(\tau_1) = \text{Gen}((\theta_2 \circ \theta_1)(\Gamma), (\theta_2 \circ \theta_1)(\Delta), \theta_2(\tau_1)) \\
& \quad \text{let } \sigma = \text{Close}(\forall \bar{\alpha}^k. \theta_2(\tau_1)) \\
& \quad \text{let } (\Xi_3, \theta_3, \tau_2, \epsilon_2) = infer(\Xi_2 \setminus \bar{\alpha}^k, (\theta_2 \circ \theta_1)(\Gamma), ((\theta_2 \circ \theta_1)(\Delta), x : \sigma), e_2) \\
& \quad \text{return } (\Xi_3, (\theta_3 \circ \theta_2 \circ \theta_1), \tau_2, \epsilon_2) \\
& infer(\Xi, \Gamma, \Delta, \text{perform } op) = \\
& \quad \text{assume } \bar{\beta}^k, \mu_1 \text{ and } \mu_2 \text{ are fresh.} \\
& \quad \text{let } \forall \bar{\alpha}^k. \tau_1 \rightarrow \tau_2 = \text{Op}(\Sigma, op) \\
& \quad \text{return } ((\Xi, \bar{\beta}^k, \mu_1, \mu_2), id, [\bar{\alpha}^k := \bar{\beta}^k](\tau_1 \rightarrow \mu_1 \tau_2), \mu_2) \\
& infer(\Xi, \Gamma, \Delta, \text{handler } h) = \\
& \quad \text{assume } \mu \text{ is fresh.} \\
& \quad \text{let } (\Xi_1, \theta_1, \tau_1, l_1, \epsilon_1) = inferHandler(\Xi, \Gamma, \Delta, h) \\
& \quad \text{return } ((\Xi, \mu), \theta_1, ((\rightarrow \langle l_1 \mid \epsilon_1 \rangle \tau_1) \rightarrow \epsilon_1 \tau_1, \mu)
\end{aligned}$$


---

**Fig. 9.** Type inference algorithm

$\Delta$  is extended with the closed effect row and used for the inference of the body of the let-expression. Consider the inference of the following let-expression.

$$\text{let } f = \lambda x. x \text{ in } f \ 1$$

```

inferHandler : (KCtx × TCtx × TCtx × Hnd) → (KCtx × Subst × MType × Lab × Eff)
inferHandler(Ξ, Γ, Δ, {op1 → v1, ..., opn → vn}) =
    assume β* and μ are fresh.
    let l = Label(Σ, op1)
    assert Σ(l) = {op1, ..., opn}
    let (Ξ0, θ0) = (Ξ, id)
    for i ∈ {1, ..., n}
        assume  $\overline{\alpha}_i^k$  are fresh.
        let  $\forall \overline{\alpha}_i^k. \tau_1^i \rightarrow \tau_2^i = \text{Op}(\Sigma, op_i)$ 
        let (Ξi1, θi1, τi, εi) = infer(Ξi-13, θi-1(Γ), θi-1(Δ), vi)
        let (Ξi2, θi2) = unify(Ξi1, εi, ⟨⟩)
        let (Ξi3, θi3)
            = unifyOp(Ξi2, (θi2 ∘ θi1 ∘ θi-1)( $\forall \overline{\alpha}_i^k. \tau_1^i \rightarrow \mu((\tau_2^i \rightarrow \mu \beta^*) \rightarrow \mu \beta^*)$ ), θi2(τi))
        let θi = θi3 ∘ θi2 ∘ θi1 ∘ θi-1
        assert  $\overline{\alpha}_i^k \notin \text{ftv}(\theta_i(\Gamma), \theta_i(\Delta))$ 
    return (Ξn3, θn, θn(β*), l, θn(μ))
    
```

---

**Fig. 10.** Type inference algorithm for handlers

The type of  $f$  is inferred to be  $\forall \alpha \mu. \alpha \rightarrow \mu \alpha$ , where  $\mu$  is an effect variable. Since the effect row of the function is closed immediately after generalization, the inference of the body  $f \ 1$  is done by  $\textit{infer}(\Xi, \Gamma, (\Delta, f : \forall \alpha. \alpha \rightarrow \langle \rangle \alpha), f \ 1)$ .

In the case of an operation call,  $\textit{infer}$  simply returns the type instantiated with a new effect variable. Here,  $\text{Op}(\cdot, \cdot)$  is an auxiliary function that selects from  $\Sigma$  the signature of the operation  $op$ .

In the handler case, we use two auxiliary functions  $\textit{inferHandler}$  and  $\text{Label}$ . The  $\textit{inferHandler}$  function infers the type of a handler. It receives a quadruple  $(\Xi, \Gamma, \Delta, h)$  of a type variable context, two typing contexts, and a handler, and returns a quintuple  $(\Xi, \theta, \tau, l, \epsilon)$  of a new type variable context, a substitution, a monomorphic type, an effect label, and an effect row. Here,  $\tau$  is the return type of the continuation captured by the handler  $h$ ,  $l$  is the effect label handled by  $h$ , and  $\epsilon$  is the rest of the effect row. The  $\text{Label}$  function returns the effect label corresponding to the given operation.

It is important to use  $\textit{unifyOp}$  instead of  $\textit{unify}$  in the type inference of handlers. For example, consider the following effect signature and handler:

$$\begin{aligned} \Sigma &= \{ l_1 : \{ op : \forall \alpha. \alpha \rightarrow \alpha \} \} \\ \text{handler } &\{ op \rightarrow \lambda x \ k. k(x + 1) \} \end{aligned}$$

This handler should be rejected for the following reason. First, the operation  $op$  is defined as having type  $\forall \alpha. \alpha \rightarrow \alpha$ . Therefore, the operation clause of  $op$  must have a type of the form  $\forall \alpha. \alpha \rightarrow \mu((\alpha \rightarrow \mu \beta) \rightarrow \mu \beta)$ , where the input and output types of the operation are universally quantified. Second, the operation clause  $\lambda x \ k. k(x + 1)$  is inferred to have type  $\text{int} \rightarrow \mu((\text{int} \rightarrow \mu \beta) \rightarrow \mu \beta)$ , where the input and output types are a concrete type  $\text{int}$ . If we use  $\textit{unifyOp}$ , unification of  $\forall \alpha. \alpha \rightarrow \mu((\alpha \rightarrow \mu \beta) \rightarrow \mu \beta)$  and  $\text{int} \rightarrow \mu((\text{int} \rightarrow \mu \beta) \rightarrow \mu \beta)$

fails, because the bound variable  $\alpha$  and  $\text{int}$  cannot be unified. On the other hand, if we use *unify*, unification of the two types succeeds, because we would pass a monomorphic type  $\alpha \rightarrow \mu ((\alpha \rightarrow \mu \beta) \rightarrow \mu \beta)$  to *unify*, which can be unified with  $\text{int} \rightarrow \mu ((\text{int} \rightarrow \mu \beta) \rightarrow \mu \beta)$ .

The soundness and completeness with respect to the syntax-directed inference rules of *infer* can be proven by induction on the structure of  $e$ .

**Theorem 9.** (*infer is sound with respect to syntax-directed inference rules*)

If  $\text{infer}(\Xi, \Gamma, \Delta, e) = (\Xi_1, \theta, \tau, \epsilon)$ , then  $\vdash \theta : \Xi \Rightarrow \Xi_1$  and  $\Xi_1 \mid \theta(\Gamma) \mid \theta(\Delta) \Vdash_{\mathfrak{s}} e : \tau \mid \epsilon$ .

**Theorem 10.** (*infer is complete with respect to syntax-directed inference rules*)

If  $\vdash \theta : \Xi \Rightarrow \Xi_2$  and  $\Xi_2 \mid \theta(\Gamma) \mid \theta(\Delta) \Vdash_{\mathfrak{s}} e : \tau \mid \epsilon$ , then  $\text{infer}(\Xi, \Gamma, \Delta, e) = (\Xi_1, \theta_1, \tau, \epsilon)$ , and there exists  $\vdash \theta_2 : \Xi_1 \Rightarrow \Xi_2$  such that  $\theta = \theta_2 \circ \theta_1$ .

Using the results so far, we can prove the main theorems: the type inference algorithm for the declarative inference rules is sound and complete.

**Theorem 11.** (*infer is sound*)

If  $\text{infer}(\Xi, \Gamma, \Delta, e) = (\Xi_1, \theta, \tau, \epsilon)$ , then  $\vdash \theta : \Xi \Rightarrow \Xi_1$  and  $\Xi_1 \mid \theta(\Gamma) \mid \theta(\Delta) \vdash e : \tau \mid \epsilon$ .

**Proof.** By Theorem 1 and Theorem 9.

**Theorem 12.** (*infer is complete*)

If  $\vdash \theta : \Xi \Rightarrow \Xi_2$  and  $\Xi_2 \mid \theta(\Gamma) \mid \theta(\Delta) \vdash e : \tau \mid \epsilon$ , then  $\text{infer}(\Xi, \Gamma, \Delta, e) = (\Xi_1, \theta_1, \tau, \epsilon)$ , and there exists  $\vdash \theta_2 : \Xi_1 \Rightarrow \Xi_2$  such that  $\theta = \theta_2 \circ \theta_1$ .

**Proof.** By Theorem 2 and Theorem 10.

## 6 Related Work

There are a variety of languages supporting effect handlers in the literature. Eff [1] is an ML-like language that employs the Hindley-Milner type inference [8, 16]. Differently from Koka, Eff has an effect system based on subtyping. As a result, the type inference algorithm [8] of Eff is more complex than the one presented in this paper.

Frank [2, 13] is a language that has effect rows and effect polymorphism similar to Koka. The difference is that Frank treats all effect rows as open ones by implicitly inserting effect variables. This means the user does not need to write type variables to express effect polymorphism, but it also means error messages may contain effect variables that the user did not write. Moreover, this approach to treating effect rows is not suitable for evidence passing because it gives rise to unnecessary search for handlers.

Links [5] is another language with a row-based effect system and effect polymorphism. What is different from Koka is that effect rows in Links are based on Remy’s record types [17], where each effect label is annotated with a presence type. Presence types increase the expressiveness of the language, but they also complicate the inference algorithm.

There is a type inference algorithm for an older version of Koka [11], which solely supports built-in effects such as exceptions and references. Similar to the current Koka, it has effect rows [9] and effect polymorphism. The type inference algorithm is an extension of the Hindley-Milner algorithm, but it infers an open effect row for all functions due to the lack of `open` and `close`.

## 7 Conclusion and Future Work

In this paper, we formalized a type inference algorithm with `open` and `close` and proved its soundness and completeness. The inference algorithm helps the Koka compiler statically determines handlers, and thus improves performance. Moreover, it allows the compiler to display precise signatures.

In future work, we plan to improve the current typing rules in order to make the typability of programs robust against small syntactic rewrites. One possible approach is to add the `open` keyword to avoid implicit opening of closed function types. This construct is similar to  $[\cdot]$  in [4] that avoids implicitly instantiating variables. However, this choice gives a gap between the formalization and implementation of Koka, because the current Koka implicitly opens closed functions types.

**Acknowledgement.** We would like to thank the anonymous reviewers for their valuable comments and the members of Masuhara laboratory for helpful advice. We are particularly grateful to Naoya Furudono for useful suggestions and discussions. This work was supported in part by JSPS KAKENHI under Grant No. JP19K24339.

## Appendix

In this appendix, we present an extension of System  $F^\epsilon$ , which we call System  $F^\epsilon + \text{restrict}$ , and a type-directed translation from `ImpKoka` to System  $F^\epsilon + \text{restrict}$ . The translation allows us to prove the type soundness of `ImpKoka` without directly defining an operational semantics for `ImpKoka`. This is a well-known technique, and is used in [10], for instance. The target calculus of the translation has a new construct `restrict`, which is necessary for establishing soundness of the translation.

### A System $F^\epsilon + \text{restrict}$

In Figs. 12, 13 and 14, we define the syntax, operational semantics, and typing rules of System  $F^\epsilon + \text{restrict}$ . The typing rule  $[\text{RESTRICT}]$  extends the closed effect row  $\langle l_1, \dots, l_n \rangle$  of the expression  $e$  to  $\langle l_1, \dots, l_n \mid \epsilon \rangle$ , where  $\epsilon$  is an arbitrary effects.

$$\boxed{\frac{\Xi \mid \Gamma \mid \Delta \vdash e : \sigma \mid \epsilon \rightsquigarrow e' \quad \Xi \mid \Gamma \mid \Delta \vdash h : \tau \mid l \mid \epsilon \rightsquigarrow h'}{\Xi \mid \Gamma \mid \Delta \vdash x : \sigma \mid \epsilon \rightsquigarrow x} \quad \Xi \mid \Gamma \mid \Delta \vdash h : \tau \mid l \mid \epsilon \rightsquigarrow h'}$$

$$\frac{x : \sigma \in \Gamma, \Delta \quad \Xi \vdash_{\text{wf}} \epsilon : \text{eff} \quad \sigma \text{ not a closed function type} \quad \Xi \mid \Gamma \mid \Delta \vdash x : \sigma \mid \epsilon \rightsquigarrow x \quad [\text{VAR}]}{f : \forall \bar{\alpha}^k. \tau_1 \rightarrow \langle l_1, \dots, l_n \rangle \tau_2 \in \Delta \quad \Xi \vdash_{\text{wf}} \epsilon : \text{eff} \quad \Xi \vdash_{\text{wf}} \epsilon' : \text{eff}} \quad [\text{VAROPEN}]$$

$$\frac{\Xi \mid \Gamma \mid \Delta \vdash f : \forall \bar{\alpha}^k. \tau_1 \rightarrow \langle l_1, \dots, l_n \mid \epsilon \rangle \tau_2 \mid \epsilon' \rightsquigarrow \Lambda \bar{\alpha}^k. \lambda \langle l_1, \dots, l_n \mid \epsilon \rangle x : \tau_1. \text{restrict}^{\langle l_1, \dots, l_n \rangle} f [\bar{\alpha}^k] x}{\Xi \mid \Gamma \mid \Delta \vdash x : \tau_1 \mid \epsilon \rightsquigarrow x}$$

$$\frac{\Xi \mid \Gamma, x : \tau_1 \mid \Delta \vdash e : \tau_2 \mid \epsilon \rightsquigarrow e' \quad \Xi \vdash_{\text{wf}} \tau_1 : * \quad \Xi \vdash_{\text{wf}} \epsilon' : \text{eff}}{\Xi \mid \Gamma \mid \Delta \vdash \lambda x. e : \tau_1 \rightarrow \epsilon \tau_2 \mid \epsilon' \rightsquigarrow \lambda^x x : \tau_1. e'} \quad [\text{LAM}]$$

$$\frac{\Xi \mid \Gamma \mid \Delta \vdash e_1 : \tau_2 \rightarrow \epsilon \tau \mid \epsilon \rightsquigarrow e'_1 \quad \Xi \mid \Gamma \mid \Delta \vdash e_2 : \tau_2 \mid \epsilon \rightsquigarrow e'_2}{\Xi \mid \Gamma \mid \Delta \vdash e_1 e_2 : \tau \mid \epsilon \rightsquigarrow e'_1 e'_2} \quad [\text{APP}]$$

$$\frac{\Xi, \alpha^k \mid \Gamma \mid \Delta \vdash v : \sigma \mid \langle \rangle \rightsquigarrow v' \quad k \neq \text{lab}}{\Xi \mid \Gamma \mid \Delta \vdash v : \forall \alpha^k. \sigma \mid \langle \rangle \rightsquigarrow \Lambda \alpha^k. v'} \quad [\text{GEN}]$$

$$\frac{\Xi \mid \Gamma \mid \Delta \vdash e : \forall \alpha^k. \sigma \mid \epsilon \rightsquigarrow e' \quad \Xi \vdash_{\text{wf}} \tau : k}{\Xi \mid \Gamma \mid \Delta \vdash e : \sigma[\alpha^k := \tau] \mid \epsilon \rightsquigarrow e'[\tau]} \quad [\text{INST}]$$

$$\frac{\Xi \mid \Gamma \mid \Delta \vdash v_1 : \sigma_1 \mid \langle \rangle \rightsquigarrow v'_1 \quad \Xi \mid \Gamma \mid \Delta, x : \sigma_1 \vdash e_2 : \tau_2 \mid \epsilon \rightsquigarrow e'_2}{\Xi \mid \Gamma \mid \Delta \vdash \text{let } x = v_1 \text{ in } e_2 : \sigma_2 \mid \epsilon \rightsquigarrow (\lambda^x x : \sigma_1. e'_2) v'_1} \quad [\text{LET}]$$

$$\frac{op : \forall \bar{\alpha}^k. \tau_1 \rightarrow \tau_2 \in \Sigma(l) \quad \bar{\alpha}^k \notin \Xi \quad \Xi \vdash_{\text{wf}} \bar{\tau} : \bar{k} \quad \Xi \vdash_{\text{wf}} \epsilon' : \text{eff}}{\Xi \mid \Gamma \mid \Delta \vdash \text{perform } op : (\tau_1 \rightarrow \langle l \mid \epsilon \rangle \tau_2) [\bar{\alpha}^k := \bar{\tau}] \mid \epsilon' \rightsquigarrow \text{perform}^\epsilon op \bar{\tau}} \quad [\text{PERFORM}]$$

$$\frac{\Xi \mid \Gamma \mid \Delta \vdash h : \tau \mid l \mid \epsilon \rightsquigarrow h' \quad \Xi \vdash_{\text{wf}} \epsilon' : \text{eff}}{\Xi \mid \Gamma \mid \Delta \vdash \text{handler } h : ((l \rightarrow \langle l \mid \epsilon \rangle \tau) \rightarrow \epsilon \tau) \mid \epsilon' \rightsquigarrow \text{handler}^\epsilon h'} \quad [\text{HANDLER}]$$

$$\frac{op_i : \forall \bar{\alpha}_i^k. \tau_1^i \rightarrow \tau_2^i \in \Sigma(l) \quad \bar{\alpha}_i^k \notin \text{ftv}(\epsilon, \tau) \quad \Xi \mid \Gamma \mid \Delta \vdash v_i : \forall \bar{\alpha}_i^k. \tau_1^i \rightarrow \epsilon((\tau_2^i \rightarrow \epsilon \tau) \rightarrow \epsilon \tau) \mid \langle \rangle \rightsquigarrow v'_i}{\Xi \mid \Gamma \mid \Delta \vdash \{op_1 \rightarrow v_1, \dots, op_n \rightarrow v_n\} : \tau \mid l \mid \epsilon \rightsquigarrow \{op_1 \rightarrow v'_1, \dots, op_n \rightarrow v'_n\}} \quad [\text{OPS}]$$

**Fig. 11.** Translation to system  $F^\epsilon + \text{restrict}$ 

We can prove the type soundness of System  $F^\epsilon + \text{restrict}$  by showing the following theorems.

**Theorem 13.** (*progress*)

If  $\emptyset \vdash e_1 : \sigma \mid \langle \rangle$  then either  $e_1$  is a value or  $e_1 \mapsto e_2$ .

**Theorem 14.** (*preservation*)

If  $\emptyset \vdash e_1 : \sigma \mid \langle \rangle$  and  $e_1 \mapsto e_2$  then  $\emptyset \vdash e_2 : \sigma \mid \langle \rangle$ .

Types		Kinds	
$\sigma ::= \alpha^k$	(type variables of kind $k$ )	$k ::= *$	(value type)
$c^k \tau \dots \tau$	(type constructor of kind $k$ )	$k \rightarrow k$	(type constructors)
$\sigma \rightarrow \epsilon \sigma$	(function type)	<b>eff</b>	(effect type $(\mu, \epsilon)$ )
$\forall \alpha^k. \sigma$	(quantified type)	<b>lab</b>	(basic effect $(l)$ )
Effect signature	$sig$	$::=$	$\{op_1 : \forall \bar{\alpha}_1. \sigma_1 \rightarrow \sigma'_1, \dots, op_n : \forall \bar{\alpha}_n. \sigma_n \rightarrow \sigma'_n\}$
Effect signatures	$\Sigma$	$::=$	$\{l_1 : sig_1, \dots, l_n : sig_n\}$
Type Constructors	$\langle \rangle$	:	<b>eff</b> empty effect row (total)
	$\langle \_   \_ \rangle$	:	<b>lab</b> $\rightarrow$ <b>eff</b> $\rightarrow$ <b>eff</b> effect row extension
Syntax	$\langle l_1, \dots, l_n \rangle$	$\doteq$	$\langle l_1   \dots   \langle l_n   \langle \rangle \rangle \dots \rangle$
	$\langle l_1, \dots, l_n   \mu \rangle$	$\doteq$	$\langle l_1   \dots   \langle l_n   \mu \rangle \dots \rangle$
	$\epsilon ::= \sigma^{\text{eff}}, \quad \mu ::= \alpha^{\text{eff}}, \quad l ::= c^{\text{lab}}$		

---

**Fig. 12.** Types of system  $F^\epsilon + \text{restrict}$ 

Expressions		Values	
$e ::= v$	(value)	$v ::= x$	(variables)
$e e$	(application)	$\lambda^\epsilon x : \sigma. e$	(abstraction)
$e[\sigma]$	(type application)	$\Lambda \alpha^k. e$	(type abstraction)
<b>handle</b> $h e$	(handler instance)	<b>handler</b> $^\epsilon h$	(effect handler)
<b>restrict</b> $^{\langle l_1, \dots, l_n \rangle} v$	(restrict)	<b>perform</b> $^\epsilon op \bar{\sigma}$	(operation)
Handlers	$h ::= \{op_1 \rightarrow f_1, \dots, op_n \rightarrow f_n\}$		
Evaluation Context	$F ::= \square \mid F e \mid v F \mid F \sigma$		
	<b>restrict</b> $^{\langle l_1, \dots, l_n \rangle} F$		
	$E ::= \square \mid E e \mid v E \mid E \sigma$		
	<b>handle</b> $h E \mid \text{restrict}^{\langle l_1, \dots, l_n \rangle} E$		
( <i>app</i> )	$(\lambda^\epsilon x : \sigma. e) v$	$\longrightarrow$	$e[x := v]$
( <i>handler</i> )	$(\text{handler}^\epsilon h) v$	$\longrightarrow$	<b>handle</b> $h (v ())$
( <i>return</i> )	<b>handle</b> $h v$	$\longrightarrow$	$v$
( <i>perform</i> )	<b>handle</b> $h E[\text{perform } op \bar{\sigma} v]$	$\longrightarrow$	$f[\bar{\sigma}] v k$ iff $op \notin \text{bop}(E) \wedge (op \rightarrow f) \in h$ where $op : \forall \bar{\alpha}. \sigma_1 \rightarrow \sigma_2 \in \Sigma(l)$
( <i>restrict</i> )	<b>restrict</b> $^{\langle l_1, \dots, l_n \rangle} v$	$\longrightarrow$	$v$
		$\text{bop}(\square)$	$= \emptyset$
		$\text{bop}(E e)$	$= \text{bop}(E)$
		$\text{bop}(v E)$	$= \text{bop}(E)$
		$\text{bop}(\text{handle } h E)$	$= \text{bop}(E) \cup \{op \mid (op \rightarrow f) \in h\}$
		$\text{bop}(\text{restrict}^{\langle l_1, \dots, l_n \rangle} E)$	$= \text{bop}(E)$
	$\frac{e \longrightarrow e'}{E[e] \mapsto E[e']} \text{ [STEP]}$		

---

**Fig. 13.** Expressions and operational semantics of  $F^\epsilon + \text{restrict}$

$\frac{\Gamma \vdash e : \sigma \mid \epsilon}{\uparrow} \quad \frac{\Gamma \vdash_{\text{val}} v : \sigma}{\uparrow} \quad \frac{\Gamma \vdash_{\text{ops}} h : \sigma \mid l \mid \epsilon}{\uparrow}$
--

  

$$\frac{\Gamma \vdash e : \sigma \mid \langle l_1, \dots, l_n \rangle \quad \Gamma \vdash_{\text{wf}} \epsilon : \text{eff}}{\Gamma \vdash \text{restrict}^{\langle l_1, \dots, l_n \rangle} e : \sigma \mid \langle l_1, \dots, l_n \mid \epsilon \rangle} \text{ [RESTRICT]} \quad \frac{x : \sigma \in \Gamma}{\Gamma \vdash_{\text{val}} x : \sigma} \text{ [VAR]}$$
  

$$\frac{\Gamma, x : \sigma_1 \vdash e : \sigma_2 \mid \epsilon \quad \Gamma \vdash_{\text{wf}} \sigma_1 : *}{\Gamma \vdash_{\text{val}} \lambda^e x : \sigma_1. e : \sigma_1 \rightarrow \epsilon \sigma_2} \text{ [LAM]} \quad \frac{\Gamma \vdash_{\text{val}} v : \sigma \quad \Gamma \vdash_{\text{wf}} \epsilon : \text{eff}}{\Gamma \vdash v : \sigma \mid \epsilon} \text{ [VAL]}$$
  

$$\frac{\Gamma \vdash e_1 : \sigma_2 \rightarrow \epsilon \sigma \mid \epsilon \quad \Gamma \vdash e_2 : \sigma_2 \mid \epsilon}{\Gamma \vdash e_1 e_2 : \sigma \mid \epsilon} \text{ [APP]}$$
  

$$\frac{\Gamma \vdash e : \sigma \mid \langle \rangle \quad \alpha^k \notin \text{ftv}(\Gamma) \quad k \neq \text{lab}}{\Gamma \vdash_{\text{val}} \Lambda \alpha^k. e : \forall \alpha. \sigma} \text{ [TABS]}$$
  

$$\frac{\Gamma \vdash e : \forall \alpha^k. \sigma_1 \mid \epsilon \quad \Gamma \vdash_{\text{wf}} \sigma' : k}{\Gamma \vdash e[\sigma'] : \sigma[\alpha := \sigma'] \mid \epsilon} \text{ [TAPP]}$$
  

$$\frac{op : \forall \bar{\alpha}^k. \sigma_1 \rightarrow \sigma_2 \in \Sigma(l) \quad \bar{\alpha}^k \notin \text{ftv}(\Gamma) \quad \Gamma \vdash_{\text{wf}} \epsilon : \text{eff}}{\Gamma \vdash_{\text{val}} \text{perform}^\epsilon op \bar{\sigma} : \sigma_1[\bar{\alpha}^k := \bar{\sigma}] \rightarrow \langle l \mid \epsilon \rangle \sigma_2[\bar{\alpha}^k := \bar{\sigma}]} \text{ [PERFORM]}$$
  

$$\frac{\Gamma \vdash_{\text{ops}} h : \sigma \mid l \mid \epsilon \quad \Gamma \vdash e : \sigma \mid \langle l \mid \epsilon \rangle}{\Gamma \vdash \text{handle}^\epsilon h e : \sigma \mid \epsilon} \text{ [HANDLE]}$$
  

$$\frac{\Gamma \vdash_{\text{ops}} h : \sigma \mid l \mid \epsilon}{\Gamma \vdash_{\text{val}} \text{handler}^\epsilon h : ((\rightarrow \langle l \mid \epsilon \rangle \sigma) \rightarrow \epsilon \sigma)} \text{ [HANDLER]}$$
  

$$\frac{op_i : \forall \bar{\alpha}^k. \sigma_1 \rightarrow \sigma_2 \in \Sigma(l) \quad \bar{\alpha}^k \notin \text{ftv}(\Gamma) \quad \Gamma \vdash_{\text{val}} v_i : \forall \bar{\alpha}^k. \sigma_1 \rightarrow \epsilon((\sigma_2 \rightarrow \epsilon \sigma) \rightarrow \sigma)}{\Gamma \vdash_{\text{ops}} \{op_1 \rightarrow v_1, \dots, op_n \rightarrow v_n\} : \sigma \mid l \mid \epsilon} \text{ [OPS]}$$

Fig. 14. Typing rules of  $F^\epsilon + \text{restrict}$ 

## B Type-Directed Translation to System $F^\epsilon + \text{restrict}$

In Fig. 11, we next define the type-directed translation from  $\text{ImpKoka}$  to System  $F^\epsilon + \text{restrict}$ . The judgment  $\Xi \mid \Gamma \mid \Delta \vdash e : \sigma \mid \epsilon \rightsquigarrow e'$  states that an expression  $e$  has type  $\sigma$  and effect  $\epsilon$  under the type variable context  $\Xi$  and typing contexts  $\Gamma$  and  $\Delta$ , and translates to an expression  $e'$  of System  $F^\epsilon + \text{restrict}$ . We can easily prove the soundness of the type-directed translation.

**Theorem 15.** (*soundness of type-directed translation*)

If  $\Xi \mid \Gamma \mid \Delta \vdash e : \sigma \mid \epsilon \rightsquigarrow e'$  then  $\Gamma, \Delta \vdash e' : \sigma \mid \epsilon$ .

**Proof.** By straightforward induction on the typing derivation.

## References

1. Bauer, A., Pretnar, M.: Programming with algebraic effects and handlers. *J. Logic. Algebraic Methods Program.* **84**(1), 108–123 (2015)
2. Convent, L., Lindley, S., McBride, C., McLaughlin, C.: Doo bee doo bee doo. *J. Funct. Program.* **30**, e9 (2020). To appear in the special issue on algebraic effects and handlers
3. Damas, L., Milner, R.: Principal type-schemes for functional programs. In: DeMillo, R.A. (ed.) *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages*, Albuquerque, New Mexico, USA, January 1982, pp. 207–212. ACM Press (1982). <https://doi.org/10.1145/582153.582176>
4. Emrich, F., Lindley, S., Stolarek, J., Cheney, J., Coates, J.: FreezeML: complete and easy type inference for first-class polymorphism. In: Donaldson, A.F., Torlak, E. (eds.) *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020*, London, UK, 15–20 June 2020, pp. 423–437. ACM (2020). <https://doi.org/10.1145/3385412.3386003>
5. Hillerström, D., Lindley, S.: Liberating effects with rows and handlers. In: Chapman, J., Swierstra, W. (eds.) *Proceedings of the 1st International Workshop on Type-Driven Development, TyDe@ICFP 2016*, Nara, Japan, 18 September 2016, pp. 15–27. ACM (2016). <https://doi.org/10.1145/2976022.2976033>
6. Jones, M.P.: A theory of qualified types. In: Krieg-Brückner, B. (ed.) *ESOP 1992*. LNCS, vol. 582, pp. 287–306. Springer, Heidelberg (1992). [https://doi.org/10.1007/3-540-55253-7\\_17](https://doi.org/10.1007/3-540-55253-7_17)
7. Kammar, O., Plotkin, G.D.: Algebraic foundations for effect-dependent optimisations. In: *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012*, New York, NY, USA, pp. 349–360. ACM (2012). <https://doi.org/10.1145/2103656.2103698>
8. Karachalias, G., Pretnar, M., Saleh, A.H., Vanderhallen, S., Schrijvers, T.: Explicit effect subtyping. *J. Funct. Program.* **30**, e15 (2020). <https://doi.org/10.1017/S0956796820000131>
9. Leijen, D.: Extensible records with scoped labels. In: *Proceedings of the 2005 Symposium on Trends in Functional Programming*, pp. 297–312 (2005)
10. Leijen, D.: HMF: simple type inference for first-class polymorphism. In: *Proceedings of the 13th ACM Symposium of the International Conference on Functional Programming, ICFP 2008*, Victoria, Canada, September 2008. <https://doi.org/10.1145/1411204.1411245>
11. Leijen, D.: Koka: programming with row polymorphic effect types. In: *5th Workshop on Mathematically Structured Functional Programming, MSFP 2014* (2014). doi: 10.4204/EPTCS.153.8
12. Leijen, D.: 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, pp. 486–499, January 2017. <https://doi.org/10.1145/3009837.3009872>
13. Lindley, S., McBride, C., McLaughlin, C.: Do be do be do. In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*, Paris, France, pp. 500–514, January 2017. <https://doi.org/10.1145/3009837.3009897>
14. Jones, S.P., Washburn, G., Weirich, S.: Wobbly types: type inference for generalised algebraic data types. MS-CIS-05-26. Microsoft Research, July 2004



15. Plotkin, G.D., Power, J.: Algebraic operations and generic effects. *Appl. Categ. Struct.* **11**(1), 69–94 (2003). <https://doi.org/10.1023/A:1023064908962>
16. Pretnar, M.: Inferring algebraic effects. *Log. Methods Comput. Sci.* **10**(3) (2014). [https://doi.org/10.2168/LMCS-10\(3:21\)2014](https://doi.org/10.2168/LMCS-10(3:21)2014)
17. Rémy, D.: Type inference for records in natural extension of ML. In: *Theoretical Aspects of Object-Oriented Programming*, pp. 67–95 (1994)
18. Robinson, J.A.: A machine-oriented logic based on the resolution principle. *J. ACM* **12**(1), 23–41 (1965). <https://doi.org/10.1145/321250.321253>
19. Wright, A.K.: Simple imperative polymorphism. *LISP Symb. Comput.* **8**(4), 343–355 (1995)
20. Xie, N., Brachthausen, J., Hillerstrom, D., Schuster, P., Leijen, D.: Effect handlers, evidently. In: *The 25th ACM SIGPLAN International Conference on Functional Programming (ICFP)*. ACM SIGPLAN, August 2020. <https://www.microsoft.com/en-us/research/publication/effect-handlers-evidently/>
21. Xie, N., Leijen, D.: Generalized evidence passing for effect handlers - efficient compilation of effect handlers to C. In: *Proceedings of the 26th ACM SIGPLAN International Conference on Functional Programming (ICFP 2021)*. Virtual, August 2021