Sound and Complete Type Inference for Closed Effect Rows

Kazuki Ikemori[1], Youyou Cong[2], Hidehiko Masuhara[3], and Daan Leijen[4]

[1] Tokyo Institute of Technology, Tokyo, Japan
ikemori.k.aa@prg.is.titech.ac.jp
[2] Tokyo Institute of Technology, Tokyo, Japan
cong@c.titech.ac.jp
[3] Tokyo Institute of Technology, Tokyo, Japan
masuhara@acm.org
[4] 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. Along with the soundness and the completeness, we proved that the algorithm infers most general types.

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[1] 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
```

[1] Koka is the first practical language that tracks the (side) effects of functions in their type.

gets the ⟨exn⟩ 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 (Section 4).
- We prove that the type inference algorithm is sound and complete (Section 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 (Section 2). Next, we present an implicitly typed calculus with algebraic effects and handlers, and give a set of declarative type inference rules (Section 3). Then, we define syntax-directed type inference rules and show they are sound and complete with respect to the declarative rules (Section 4). We also define a type inference algorithm and prove its soundness and completeness (Section 5). Lastly, we discuss related work (Section 6) and conclude with future directions (Section 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 `read`$_2$ 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` belonw.

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

This function `f` is inferred to have the type `forall`$\langle$`a,e`$_1\rangle$ `a` $\rightarrow$ $\langle$`read`$_2$`|e`$_1\rangle$ `a`. The type contains an effect variable `e`$_1$, 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([]) else head([1])
```

As the two branches must have the same effect row, $\langle$`read`$_2$`|e`$_1\rangle$ and $\langle$`exn|e`$_2\rangle$ are unified to $\langle$`exn,read`$_2$`|e`$_3\rangle$ or $\langle$`read`$_2$`,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 `read`$_2$ handler because the effect row of `f` has been changed from $\langle$`read`$_2$`|e`$_2\rangle$ to $\langle$`exn,read`$_2$`|e`$_3\rangle$. In the latter case, the Koka compiler would dynamically search for the `exn` handler because the effect row of `g` has been changed from $\langle$`exn|e`$_2\rangle$ to $\langle$`read`$_2$`,exn|e`$_3\rangle$. To avoid dynamic search, we would like to infer the type `forall`$\langle$`a`$\rangle$ `a` $\to$ $\langle$`read`$_2\rangle$ `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 `read`$_2$ effect. Therefore, we can obtain a corresponding handler in constant time (see [20, 21] for more details). In tuitively, higer 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`$\langle$`a,e`$\rangle$ `a` $\to$ `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`$\langle$`a`$\rangle$ `a` $\to$ $\langle\rangle$ `a`, which has a total effect $\langle\rangle$, seems more natural.

We might take other approachs 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 Section 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 `forall⟨a,e⟩ a → e a`. When the function is bound to `id`, the type is *closed* to `forall⟨a⟩ a → ⟨⟩ 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 `forall⟨a,e⟩ a → e a` and then unified to `int` $\to$ $\langle\texttt{read}_2\rangle$ `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 $\langle \texttt{l}_1,\ldots,\texttt{l}_\texttt{n}\rangle$, yielding $\langle \texttt{l}_1,\ldots,\texttt{l}_\texttt{n} \mid \texttt{e}\rangle$. 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 Fω. The structure of this section is as follows. First, we define the syntax of kinds, types, and effect rows (Sections 3.1.1-3.1.3). Next, we define the syntax of expressions (Section 3.1.4). Lastly, we give a set of declarative type inference rules (Section 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

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

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

$$\mu ::= \alpha^{\mathsf{eff}}, \qquad l ::= c^{\mathsf{lab}}$$

Figure 1. Types, Kinds, Effect Signatures, and Effect Rows of ImpKoka

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

Signatures We define operations *op*, effect signatures *sig*, and sets of effect signatures Σ again as in System F^{ϵ} (Figure 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 Σ is defined at the top level.

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^{ϵ}, an effect row is either an empty row $\langle\, \rangle$, or an effect variable μ, or an extension $\langle l \mid \epsilon \rangle$ of an effect row ϵ with an effect label l, respectively. For example, assuming exn is an effect label representing exceptions, $\langle \mathsf{exn}, \mathsf{read}_2 \rangle$ is an effect row representing a collection of exception and reader effects. The kinds of $\langle\, \rangle$ and l are eff (representing an effect type) and lab (representing an effect label), respectively. The kind of $\langle \_ \mid \_ \rangle$ is $\mathsf{lab} \rightarrow \mathsf{eff} \rightarrow \mathsf{eff}$. 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^{ϵ} (Figure 2). The [refl] and [eq-trans] rules are the reflexivity and transitivity rules. The [eq-swap] rule says that two effect labels l_1 and l_2 can be swapped if they are distinct. The [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 $\langle \mathsf{exn} \rangle$ and $\langle \mathsf{exn}, \mathsf{exn} \rangle$, 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.

$$\frac{}{\epsilon \equiv \epsilon}\ [\text{REFL}] \qquad \frac{\epsilon_1 \equiv \epsilon_2 \quad \epsilon_2 \equiv \epsilon_3}{\epsilon_1 \equiv \epsilon_3}\ [\text{EQ-TRANS}]$$

$$\frac{l_1 \not\equiv l_2 \quad \epsilon_1 \equiv \epsilon_2}{\langle l_1, l_2 \mid \epsilon_1\rangle \equiv \langle l_2, l_1 \mid \epsilon_2\rangle}\ [\text{EQ-SWAP}] \qquad \frac{\epsilon_1 \equiv \epsilon_2}{\langle l \mid \epsilon_1\rangle \equiv \langle l \mid \epsilon_2\rangle}\ [\text{EQ-HEAD}]$$

Figure 2. Equivalence of Row Types

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

Figure 3. Expressions of ImpKoka

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

3.2. Declarative Type Inference Rules

We now turn to the declarative type inference rules with open and close (Figure 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 Ξ and typing context Γ and Δ, expression e has type σ and performs effect ϵ. Among the two contexts, Δ 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 Γ is a type assignment for all other variables.

The [Var] and [VarOpen] rules in Figure 4 are interesting. The [Var] rule concludes with a type σ that comes from either Δ or Γ, and an arbitrary effect ϵ. Note that, although a variable does not perform any effects, we cannot replace ϵ 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 [VarOpen] rule derives an open function type for a variable of a closed function type. The variable must reside in Δ, because we can only open the type of named functions. As an example, we can use [VarOpen] to make id(perform $ask\text{-}int$ ()) well-typed, because we can derive $\Xi \mid \Gamma \mid \Delta \vdash id : \mathsf{int} \rightarrow \langle \mathsf{read}_2\rangle\, \mathsf{int}$ by [VarOpen] and $\Xi \mid \Gamma \mid \Delta \vdash$ perform $ask\text{-}int$ () $: \mathsf{int} \mid \langle \mathsf{read}_2\rangle$ by [Perform] and [App].

$$\boxed{\;\Xi \mid \Gamma \mid \Delta \vdash e : \sigma \mid \epsilon \qquad \Xi \mid \Gamma \mid \Delta \vDash h : \tau \mid l \mid \epsilon\;}$$

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

$$\frac{f : \forall \overline{\alpha}^{\overline{k}}.\, \tau_1 \to \langle l_1, \ldots, l_n \rangle\, \tau_2 \in \Delta \qquad \Xi \vdash_{\mathsf{wf}} \epsilon : \mathsf{eff} \quad \Xi \vdash_{\mathsf{wf}} \epsilon' : \mathsf{eff}}{\Xi \mid \Gamma \mid \Delta \vdash f : \forall \overline{\alpha}^{\overline{k}}.\, \tau_1 \to \langle l_1, \ldots, l_n \mid \epsilon \rangle\, \tau_2 \mid \epsilon'}\ [\textsc{VarOpen}]$$

$$\frac{\begin{array}{c}\Xi \mid \Gamma, x : \tau_1 \mid \Delta \vdash e : \tau_2 \mid \epsilon \\ \Xi \vdash_{\mathsf{wf}} \tau_1 : * \quad \Xi \vdash_{\mathsf{wf}} \epsilon' : \mathsf{eff}\end{array}}{\Xi \mid \Gamma \mid \Delta \vdash \lambda x.\, e : \tau_1 \to \epsilon\, \tau_2 \mid \epsilon'}\ [\textsc{Lam}] \qquad \frac{\begin{array}{c}\Xi \mid \Gamma \mid \Delta \vdash e_1 : \tau_2 \to \epsilon\, \tau \mid \epsilon \\ \Xi \mid \Gamma \mid \Delta \vdash e_2 : \tau_2 \mid \epsilon\end{array}}{\Xi \mid \Gamma \mid \Delta \vdash e_1\, e_2 : \tau \mid \epsilon}\ [\textsc{App}]$$

$$\frac{\begin{array}{c}\Xi, \alpha^k \mid \Gamma \mid \Delta \vdash v : \sigma \mid \langle\rangle \\ k \not\equiv \mathsf{lab} \quad \alpha^k \notin \mathsf{ftv}(\Gamma, \Delta)\end{array}}{\Xi \mid \Gamma \mid \Delta \vdash v : \forall \alpha^k.\, \sigma \mid \langle\rangle}\ [\textsc{Gen}] \qquad \frac{\Xi \mid \Gamma \mid \Delta \vdash e : \forall \alpha^k.\, \sigma \mid \epsilon \quad \Xi \vdash_{\mathsf{wf}} \tau : k}{\Xi \mid \Gamma \mid \Delta \vdash e : \sigma[\alpha^k := \tau] \mid \epsilon}\ [\textsc{Inst}]$$

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

$$\frac{op : \forall \overline{\alpha}^{\overline{k}}.\, \tau_1 \to \tau_2 \in \Sigma(l) \quad \overline{\alpha}^{\overline{k}} \notin \Xi \quad \Xi \vdash_{\mathsf{wf}} \overline{\tau} : \overline{k} \quad \Xi \vdash_{\mathsf{wf}} \epsilon : \mathsf{eff} \quad \Xi \vdash_{\mathsf{ef}} \epsilon' : \mathsf{eff}}{\Xi \mid \Gamma \mid \Delta \vdash \mathsf{perform}\, op : [\overline{\alpha}^{\overline{k}} := \overline{\tau}](\tau_1 \to \langle l \mid \epsilon \rangle\, \tau_2) \mid \epsilon'}\ [\textsc{Perform}]$$

$$\frac{\Xi \mid \Gamma \mid \Delta \vDash h : \tau \mid l \mid \epsilon \quad \Xi \vdash_{\mathsf{wf}} \epsilon' : \mathsf{eff}}{\Xi \mid \Gamma \mid \Delta \vdash \mathsf{handler}\, h : (() \to \langle l \mid \epsilon \rangle\, \tau) \to \epsilon\, \tau) \mid \epsilon'}\ [\textsc{Handler}]$$

$$\frac{\overline{\alpha}_i^{\overline{k}} \notin \mathsf{ftv}(\epsilon, \tau) \quad op_i : \forall \overline{\alpha}_i^{\overline{k}}.\, \tau_1^i \to \tau_2^i \in \Sigma(l) \quad \Xi \mid \Gamma \mid \Delta \vdash v_i : \forall \overline{\alpha}_i^{\overline{k}}.\, \tau_1^i \to \epsilon\, ((\tau_2^i \to \epsilon\, \tau) \to \epsilon\, \tau) \mid \langle\rangle}{\Xi \mid \Gamma \mid \Delta \vDash \{ op_1 \to v_1, \ldots, op_n \to v_n \} : \tau \mid l \mid \epsilon}\ [\textsc{Ops}]$$

Figure 4. Declarative Type Inference Rules

Other type rules in Figure 4 are standard. The [Lam] rule derives a function type for a lambda abstraction. Observe that the effect ϵ of the body e is integrated into the type $\tau_1 \to \epsilon\, \tau_2$ in the conclusion. Note that the lambda-bound variable x is added to Γ, not Δ. Therefore, in the derivation of the body e, the type of x cannot be opened using [VarOpen]. The [App] rule requires that the function e_1, the argument e_2, and the body of the function have the same effect ϵ. The [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 ImpKoka to System $\mathrm{F}^{\epsilon} + \mathsf{restrict}$. The [Inst] rule is completely standard. The [Let] rule is used to bind values with polymorphic types. As in the [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 Δ, not Γ. Therefore, in the derivation of the body e_2, the type of x can be opened via [VarOpen].

$$\boxed{\vdash \theta : \Xi \Rightarrow \Xi' \qquad \Xi \vdash \sigma_1 \sqsubseteq \sigma_2}$$

$$\frac{}{\vdash \emptyset : \cdot \Rightarrow \Xi}\ [\textsc{Empty}] \qquad \frac{\vdash \theta : \Xi_1 \Rightarrow \Xi_2 \quad \Xi_2 \vdash \tau : k}{\vdash \theta[\alpha^k := \tau] : (\Xi_1,\, \alpha^k) \Rightarrow \Xi_2}\ [\textsc{Extend}]$$

$$\frac{\Xi \vdash_{\mathsf{wf}} \overline{\tau} : \overline{k} \quad \tau_2 = [\overline{\alpha}^{\overline{k}} := \overline{\tau}]\tau_1 \quad \overline{\beta}^{\overline{k'}} \notin \mathsf{ftv}(\forall \overline{\alpha}^{\overline{k}}.\, \tau_1)}{\Xi \vdash \forall \overline{\alpha}^{\overline{k}}.\, \tau_1 \sqsubseteq \forall \overline{\beta}^{\overline{k'}}.\, \tau_2}\ [\textsc{TypeOrdering}]$$

Figure 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 $\overline{\tau}$. The notation $op\colon \forall \overline{\alpha}^{\overline{k}}.\tau_1 \to \tau_2 \in \Sigma(l)$ of the premise means that the operation $op\colon \forall \overline{\alpha}^{\overline{k}}.\tau_1 \to \tau_2$ 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 $() \to \langle l \mid \epsilon \rangle\, \tau$ 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 $\forall \overline{\alpha}_i^{\overline{k}}.\, \tau_1^i \to \epsilon\, ((\tau_2^i \to \epsilon\, \tau) \to \epsilon\, \tau)$, where τ_1^i is the input type of the operation, and $\tau_2^i \to \epsilon\, \tau$ is the type of the continuation. The condition $\overline{\alpha}_i^{\overline{k}} \notin \mathsf{ftv}(\epsilon,\, \tau)$ is necessary for type preservation of the translation from ImpKoka to System F$^\epsilon$+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 θ replaces type variables in context Ξ_1 with types well-formed under context Ξ_2. There are two formation rules for substitutions: [Empty] forms an empty substitution, and [Extend] extends a substitution θ 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 σ_2 is more specific than type σ_1 under

$$\boxed{\quad \Xi \mid \Gamma \mid \Delta \Vdash_{\mathsf{s}} e \,:\, \tau \mid \epsilon \qquad \Xi \mid \Gamma \mid \Delta \vDash_{\mathsf{s}} h \,:\, \tau \mid l \mid \epsilon \quad}$$

$$\frac{x \,:\, \sigma \in \Gamma, \Delta \quad \Xi \vdash \sigma \sqsubseteq \tau \quad \Xi \Vdash_{\mathsf{wf}} \epsilon \,:\, \mathsf{eff}}{\Xi \mid \Gamma \mid \Delta \Vdash_{\mathsf{s}} x \,:\, \tau \mid \epsilon} \; [\textsc{Var}]$$

$$\frac{\begin{array}{l} f \,:\, \forall \overline{\alpha}^{k}.\, \tau_1 \rightarrow \langle l_1, \ldots, l_n \rangle\, \tau_2 \in \Delta \\ \quad \Xi \vdash \forall \overline{\alpha}^{k}.\, \tau_1 \rightarrow \langle l_1, \ldots, l_n \rangle\, \tau_2 \sqsubseteq \tau_1' \rightarrow \langle l_1, \ldots, l_n \rangle\, \tau_2' \\ \quad \Xi \vdash_{\mathsf{wf}} \epsilon \,:\, \mathsf{eff} \quad \Xi \vdash_{\mathsf{wf}} \epsilon' \,:\, \mathsf{eff} \end{array}}{\Xi \mid \Gamma \mid \Delta \Vdash_{\mathsf{s}} f \,:\, \tau_1' \rightarrow \langle l_1, \ldots, l_n \mid \epsilon \rangle\, \tau_2' \mid \epsilon'} \; [\textsc{VarOpen}]$$

$$\frac{\begin{array}{l} \Xi \mid \Gamma \mid \Delta \Vdash_{\mathsf{s}} v_1 \,:\, \tau_1 \mid \langle\,\rangle \quad \Xi \setminus \overline{\alpha}^{k} \mid \Gamma \mid \Delta, x \,:\, \mathsf{Close}(\sigma_1) \Vdash_{\mathsf{s}} e_2 \,:\, \tau_2 \mid \epsilon \\ \quad \sigma_1 = \mathsf{Gen}(\Gamma, \Delta, \tau_1) = \forall \overline{\alpha}^{k}.\, \tau_1 \end{array}}{\Xi \mid \Gamma \mid \Delta \Vdash_{\mathsf{s}} \mathsf{let}\, x = v_1\, \mathsf{in}\, e_2 \,:\, \tau_2 \mid \epsilon} \; [\textsc{Let}]$$

$$\frac{\begin{array}{l} op_i \,:\, \forall \overline{\alpha}_i^{k}.\, \tau_1^i \rightarrow \tau_2^i \in \Sigma(l) \quad \overline{\alpha}_i^{k} \notin \mathsf{ftv}(\epsilon, \tau) \\ \Xi, \overline{\alpha}_i^{k} \mid \Gamma \mid \Delta \Vdash_{\mathsf{s}} v_i \,:\, \tau_1^i \rightarrow \epsilon\, ((\tau_2^i \rightarrow \epsilon\, \tau) \rightarrow \epsilon\, \tau) \mid \langle\,\rangle \quad \overline{\alpha}_i^{k} \notin \mathsf{ftv}(\Gamma, \Delta) \end{array}}{\Xi \mid \Gamma \mid \Delta \vDash_{\mathsf{s}} \{ op_1 \rightarrow v_1, \ldots, op_n \rightarrow v_n \} \,:\, \tau \mid l \mid \epsilon} \; [\textsc{Ops}]$$

Figure 6. Syntax-directed Type Inference Rules (excerpt)

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

$$\begin{array}{l} \Xi \vdash \forall \alpha\,.\, \alpha \rightarrow \langle\,\rangle\, \alpha \sqsubseteq \mathsf{int} \rightarrow \langle\,\rangle\, \mathsf{int} \\ \Xi \vdash \forall \alpha\, \mu\,.\, \alpha \rightarrow \mu\, \alpha \sqsubseteq \forall \beta.\, (\beta \rightarrow \langle \mathsf{exn} \rangle\, \beta) \rightarrow \langle \mathsf{exn} \rangle\, (\beta \rightarrow \langle \mathsf{exn} \rangle\, \beta) \end{array}$$

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 Section 3.2, there are no rules corresponding to [Gen] and [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 $\mathsf{Gen}(\cdot, \cdot, \cdot)$ and $\mathsf{Close}(\cdot)$. The Gen function is the standard generalization function, with a standard definition of free type variables. The Close function closes the effect row of a function type. For example, $\mathsf{Close}(\forall \mu\, \alpha.\, \alpha \rightarrow \mu\, \alpha) = \forall \alpha.\, \alpha \rightarrow \langle\,\rangle\, \alpha$.

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

$$\begin{array}{lll} \mathsf{Close}(\forall \mu\, \overline{\alpha}^{k}.\, \tau_1 \rightarrow \langle l_1, \ldots, l_n \mid \mu \rangle\, \tau_2) = \forall \overline{\alpha}^{k}.\, \tau_1 \rightarrow \langle l_1, \ldots, l_n \rangle\, \tau_2 & \mathsf{iff}\, \mu \notin \mathsf{ftv}(\tau_1, \tau_2) \\ \mathsf{Close}(\sigma) = \sigma & \mathsf{otherwise} \end{array}$$

Among the key rules, [Var] is standard and derives the type instantiated by $\overline{\tau}$. The [VarOpen] rule derives an open function type from a closed one by inserting an arbitrary effect row. The [Let] rule generalizes the type of the bound expression using Gen, and derives the type of the body under an extended type context $\Delta,\ x :\ \mathsf{Close}(\sigma)$. In the [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 $\Xi \mid \Gamma \mid \Delta \Vdash_{\mathsf{s}} e \ :\ \tau \mid \epsilon$ 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 $\Xi' \mid \Gamma \mid \Delta \Vdash_{\mathsf{s}} e \ :\ \tau \mid \epsilon$,
where $\Xi \subseteq \Xi'$ and $\Xi' \vdash \mathsf{Gen}(\Gamma,\ \Delta,\ \tau) \sqsubseteq \sigma$.

4.4. Principles on the Use of [Let] Rule

It is important that the Close function is applied only in the [Let] rule. The reason is that, if 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 Close and [Lam] rule. We first define Close_1 as a function that closes the effect variable of a monomorphic function type. For example, $\mathsf{Close}_1(\alpha \to \mu\,\alpha) \ =\ \alpha \to \langle\,\rangle\,\alpha$, where $\Gamma\ =\ \emptyset$ and $\Delta\ =\ \emptyset$.

$$\begin{array}{ll} \mathsf{Close}_1(\tau_1 \to \langle l_1, \ldots, l_n \mid \mu\rangle\,\tau_2) \ =\ \tau_1 \to \langle l_1,\ \ldots,\ l_n\rangle\,\tau_2 & \mathsf{iff}\ \mu \notin \mathsf{ftv}(\tau_1,\ \tau_2) \\ \mathsf{Close}_1(\tau) \ =\ \tau & \mathsf{otherwise} \end{array}$$

We next define [Lam1] as a typing rule that closes the monomorphic function type of a lambda-bound variable using 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.

$$\frac{\begin{array}{c}\Xi \mid \Gamma \mid \Delta,\ x\ :\ \tau_1' \Vdash_{\mathsf{s}} e\ :\ \tau_2 \mid \epsilon \qquad \tau_1'\ =\ \mathsf{Close}_1(\tau_1) \\ \Xi \vdash_{\mathsf{wf}} \tau_1\ :\ * \qquad \Xi \vdash_{\mathsf{wf}} \epsilon'\ :\ \mathsf{eff}\end{array}}{\Xi \mid \Gamma \mid \Delta \Vdash_{\mathsf{s}} \lambda x.\, e\ :\ \tau_1 \to \epsilon\,\tau_2 \mid \epsilon'}\ [\textsc{Lam1}]$$

With [Lam1], it is possible to derive wrong effects. Consider the following expression.

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

First, we derive $\emptyset \mid \emptyset \mid \emptyset \Vdash_{\mathsf{s}} \lambda g.\ g\ ()\ :\ (() \to \mu\,\alpha) \to \langle\,\rangle\,\alpha$ by the [Lam1] and [VarOpen] rules. Next, we extend the type context Δ with $f :\ \forall \mu\,\alpha.\,(() \to \mu\,\alpha) \to \langle\,\rangle\,\alpha$ by the [Let] rule. Then, we obtain the following derivation by the [App] rule.

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

This typing judgment is clearly wrong, because it does not track the read_2 effect. The problem can be avoided if we use Close only in the [Let] rule: in that case, f will be given a correct type $\forall \mu.\,\alpha\,(() \to \mu\,\alpha) \to \mu\,\alpha$.

4.5. Fragility of [Let] Rule

An unfortunate aspect of our current rules is that the [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:

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

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

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

$$\begin{array}{lcl} \mathit{remote} & : & (() \to \langle\,\rangle\ ()) \to \langle \mathsf{read}_2 \rangle\ \mathsf{bool} \\ \mathit{remote} & = & \lambda f.\ \mathsf{perform}\ \mathit{ask\text{-}bool}\ () \\ & & \\ \mathit{foo} & = & \lambda f.\ \mathit{remote}\ f;\ f\ () \\ \mathit{bar} & = & \lambda f.\ \mathit{remote}\ f;\ \mathsf{let}\ g = f\ \mathsf{in}\ g\ () \end{array}$$

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

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 (Section 5.1), and then discuss the unification algorithm (Section 5.2) and the type inference algorithm (Section 5.3).

$$\begin{array}{ll}
\mathsf{dom} \;:\; \mathsf{Subst} \rightarrow \mathsf{TypeVars} & \mathsf{const} \;:\; \mathsf{MTypes} \rightarrow \mathsf{Cons} \\
\mathsf{dom}(\emptyset) \;=\; \emptyset & \mathsf{const}(\emptyset) \;=\; \emptyset \\
\mathsf{dom}(\theta[\alpha^k := \tau]) \;=\; \{\, \alpha^k \,\} \cup \mathsf{dom}(\theta) & \mathsf{const}(\{\, \mathsf{s}^k \,\} \cup \overline{\tau}) \;=\; \{\, \mathsf{s}^k \,\} \cup \mathsf{const}(\overline{\tau}) \\
 & \mathsf{const}(\{\, \tau \,\} \cup \overline{\tau}) \;=\; \mathsf{const}(\overline{\tau}) \\
\mathsf{codom} \;:\; \mathsf{Subst} \rightarrow \mathsf{MTypes} & \\
\mathsf{codom}(\emptyset) \;=\; \emptyset & \mathsf{tail} \;:\; \mathsf{Eff} \rightarrow \mathsf{Eff} \\
\mathsf{codom}(\theta[\alpha^k := \tau]) \;=\; \{\, \tau \,\} \cup \mathsf{dom}(\theta) & \mathsf{tail}(\langle l \mid \epsilon \rangle) \;=\; \epsilon
\end{array}$$

$$\begin{array}{lll}
(-) \;:\; (\mathsf{Subst},\ \mathsf{TypeVars}) \rightarrow \mathsf{Subst} & & \\
\emptyset - \overline{\alpha}^{\overline{k}} \;=\; \emptyset & & \\
\theta[\beta^k := \tau] \;-\; \overline{\alpha}^{\overline{k}} \;=\; \theta - \overline{\alpha}^{\overline{k}} & & (\beta^k \in \overline{\alpha}^{\overline{k}}) \\
\theta[\beta^k := \tau] \;-\; \overline{\alpha}^{\overline{k}} \;=\; (\theta - \overline{\alpha}^{\overline{k}})\,[\beta^k := \tau] & & (\beta^k \notin \overline{\alpha}^{\overline{k}})
\end{array}$$

Figure 7. Auxiliary Functions

5.1. Auxiliary Functions

In Figure 7, we define auxiliary functions. The functions $\mathsf{dom}(\,\cdot)$, $\mathsf{codom}(\,\cdot)$, and $\mathsf{tail}(\,\cdot)$ are completely standard. The dom function takes a substitution θ and returns a set of type variables that are included in the domain of θ. The codom function also takes a substitution θ and returns a set of monomorphic types that are included in the codomain of θ. The 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 const functions are used in the *unifyOp* function (Section 5.2). The former takes a substitution θ and set of type variables $\overline{\alpha}^{\overline{k}}$, and returns a new substitution with $\overline{\alpha}^{\overline{k}}$ removed from the domain of θ. The latter takes a set of monomorphic types, and returns a set of skolem constants.

5.2. Unification Algorithm

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

Let us look at the *unify* and *unifyEffect* functions. Given a triple $(\Xi,\ \epsilon,\ l)$ of a type context, an effect row type, and an effect label, *unifyEffect* returns a triple $(\Xi_1,\ \theta_1,\ \epsilon_1)$ of a new type context, a substitution, and an effect row. We can transfom ϵ into an effect row whose head effect label is l. As an example, let us consider the following unification problem.

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

This succeeds and returns the following effect row and substitution:

$$\epsilon_1 \;=\; \langle \mathsf{read}_2 \mid \mu_1 \rangle \qquad \theta_1 \;=\; \mathit{id}[\mu := \langle \mathsf{exn} \mid \mu_1 \rangle]$$

```
$unify \;:\; (\mathsf{KCtx} \times \mathsf{MType} \times \mathsf{MType}) \rightarrow (\mathsf{KCtx} \times \mathsf{Subst})$
$unify(\Xi, \alpha^k, \alpha^k) \;=\;$ return $(\Xi,\ id)$
$unify((\Xi, \alpha^k), \alpha^k,\ \tau) \;=$
        assert $\Xi \vdash_{\mathsf{wf}} \tau \;:\; k$
        return $(\Xi,\ id[\alpha^k := \tau])$
$unify((\Xi, \alpha^k), \tau, \alpha^k) \;=$
        assert $\Xi \vdash_{\mathsf{wf}} \tau \;:\; k$
        return $(\Xi,\ id[\alpha^k := \tau])$
$unify(\Xi, \langle\,\rangle, \langle\,\rangle) \;=\;$ return $(\Xi,\ id)$
$unify(\Xi,\ \langle l \mid \epsilon_1\rangle,\ \langle l' \mid \epsilon_2\rangle) \;=$
        let $(\Xi_1,\ \theta_1,\ \epsilon_3) \;=\; unifyEffect(\Xi,\ \langle l' \mid \epsilon_2\rangle,\ l)$
        assert $\mathsf{tail}(\epsilon_1) \;\notin \mathsf{dom}(\theta_1)$
        let $(\Xi_2,\ \theta_2) \;=\; unify(\Xi_1,\ \theta_1(\epsilon_1),\ \epsilon_3)$
        return $(\Xi_2,\ \theta_2 \,\circ \theta_1)$
$unify(\Xi,\ \tau_1 \rightarrow \epsilon\, \tau_2,\ \tau_1' \rightarrow \epsilon'\, \tau_2') \;=$
        let $(\Xi_1,\ \theta_1) \;=\; unify(\Xi,\ \tau_1,\ \tau_1')$
        let $(\Xi_2,\ \theta_2) \;=\; unify(\Xi_1,\ \theta_1(\epsilon),\ \theta_1(\epsilon'))$
        let $(\Xi_3,\ \theta_3) \;=\; unify(\Xi_2,\ (\theta_2 \,\circ \theta_1)(\tau_2'),\ (\theta_2 \,\circ \theta_1)(\tau_2'))$
        return $(\Xi_3,\ \theta_3 \,\circ \theta_2 \,\circ \theta_1)$

$unifyEffect \;:\; (\mathsf{KCtx} \times \mathsf{Eff} \times \mathsf{Lab}) \rightarrow (\mathsf{KCtx} \times \mathsf{Subst} \times \mathsf{Eff})$
$unifyEffect(\Xi,\ \langle l' \mid \epsilon\rangle,\ l) \;=$
    $\mid l' \equiv l \Rightarrow$ return $(\Xi,\ id,\ \epsilon)$
    $\mid l' \not\equiv l \Rightarrow$ let $(\Xi_1,\ \theta_1,\ \epsilon_1) \;=\; unifyEffect(\Xi,\ \epsilon,\ l)$
                return $(\Xi_1,\ \theta_1,\ \langle l' \mid \epsilon_1\rangle)$
$unifyEffect((\Xi,\ \mu),\ \mu,\ l) \;=$
    assume $\mu_1$ is fresh
    return $((\Xi,\ \mu_1),\ id[\mu := \langle l \mid \mu_1\rangle]),\ \mu_1)$

$unifyOp \;:\; (\mathsf{KCtx} \times \mathsf{Type} \times \mathsf{MType}) \rightarrow (\mathsf{KCtx} \times \mathsf{Subst})$
$unifyOp(\Xi,\ \forall\overline{\alpha}^{\overline{k}}.\, \tau_1,\ \tau_2) \;=$
    assume $\overline{\mathsf{s}}^{\overline{k}}$ are fresh (skolem constants)
    let $(\Xi_1,\ \theta_1) \;=\; unify(\Xi,\ id[\overline{\alpha}^{\overline{k}} := \overline{\mathsf{s}}^{\overline{k}}]\tau_1,\ \tau_2)$
    assert $\overline{\mathsf{s}}^{\overline{k}} \;\notin \mathsf{const}(\mathsf{codom}(\theta_1 \;-\; \mathsf{ftv}(\tau_2)))$
    return $((\Xi_1,\ \overline{\alpha}^{\overline{k}}),\ [\overline{\mathsf{s}^k \,\mapsto \alpha^k}] \,\circ \theta_1)$
```

Figure 8. Unification Algorithm

The *unifyEffect* function is sound. That is, if *unifyEffect*$(\Xi,\ \epsilon,\ l)$ succeeds, it returns a substitution θ_1 and an effect row ϵ_1 that satisfy $\theta_1(\epsilon) \equiv \langle l \mid \theta_1(\epsilon_1)\rangle$.

Theorem 3. (*unifyEffect is sound*)
If $\Xi \vdash_{\mathsf{wf}} \epsilon : \mathsf{eff}$, $\Xi \vdash_{\mathsf{wf}} l : \mathsf{lab}$ and *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 ϵ can be rewritten to an effect row of the form $\langle l \mid \theta(\epsilon')\rangle$ by the substitution θ, *unifyEffect*$(\Xi,\ \epsilon,\ l)$ succeeds and returns the most general substitution θ_1.

Theorem 4. (*unifyEffect is complete*)
If $\Xi \vdash_{\mathsf{wf}} \epsilon : \mathsf{eff}$, $\Xi \vdash_{\mathsf{wf}} l : \mathsf{lab}$, $\vdash \theta : \Xi \Rightarrow \Xi_2$ and $\theta(\epsilon) \equiv \langle l \mid \theta(\epsilon')\rangle$,
then *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 unifcation algorithm is *unify*$(\Xi,\ \langle l \mid \epsilon_1\rangle,\ \langle l' \mid \epsilon_2\rangle)$ that corresponds to [UNI-ROW] case in the [9]. First, *unifyEffect*$(\Xi,\ \langle l' \mid \epsilon_2\rangle,\ l)$ in the first line returns a substitution θ_1 and an effect row ϵ_3, where $\langle l \mid \epsilon_3\rangle \equiv \theta_1(\langle l' \mid \epsilon_2\rangle)$. Next, $\mathsf{tail}(\epsilon_1) \notin \mathsf{dom}(\theta_1)$ in the second line confirms that effect variable of ϵ_1 is not included in the domain of θ_1. This is needed to ensure the termination of the unification algorithm. Then, *unify*$(\Xi_1,\ \theta_1(\epsilon_1),\ \epsilon_3)$ in the third line returns a substitution θ_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 *unify*$(\Xi,\ \tau_1,\ \tau_2)$ succeeds, it returns the substitution θ_1 that unifies τ_1 and τ_2.

Theorem 5. (*unify is sound*)
If $\Xi \vdash_{\mathsf{wf}} \tau_1 : k$, $\Xi \vdash_{\mathsf{wf}} \tau_2 : k$, and *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 τ_1 and τ_2 are unifiable, *unify*$(\Xi,\ \tau_1,\ \tau_2)$ succeeds and it returns the most general substitution θ_1.

Theorem 6. (*unify is complete*)
If $\Xi \vdash_{\mathsf{wf}} \tau_1 : k$, $\Xi \vdash_{\mathsf{wf}} \tau_2 : k$, $\vdash \theta : \Xi \Rightarrow \Xi_2$ and $\theta(\tau_1) = \theta(\tau_2)$,
then *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 $(\Xi,\ \forall\overline{\alpha^k}.\,\tau_1,\ \tau_2)$ of a type variable context, a type scheme, and a monomorphic type τ, 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 $\overline{\alpha^k}$ do not escape or occur free in the substitution $\overline{[\mathsf{s}^k \mapsto \alpha^k]} \circ \theta_1$. The algorithm reads as follows. We first replace the free type variables $\overline{\alpha^k}$ of τ_1 with fresh skolem constants $\overline{\mathsf{s}^k}$, and unify the resulting type with τ_2. When we obtain a substitution θ_1 as the result, we check the codomain of $\theta_1 - \mathsf{ftv}(\tau_2)$ does not contain skolem constants $\overline{\mathsf{s}^k}$, using the const function. If this checking succeeds, we construct a new substitution by replacing the skolem constants $\overline{\mathsf{s}}^k$ in θ_1 back to type variables $\overline{\alpha}^k$, and return the resulting substitution. As an example, consider the following unification problem:

$$\mathit{unifyOp}(\Xi,\ \forall\alpha.\,\alpha \to \mathsf{exn}\ ((\alpha \to \langle\mathsf{exn}\rangle\ \mathsf{int}) \to \langle\mathsf{exn}\rangle\ \mathsf{int}),\ \beta_1 \to \mu\,\beta_2)$$

This succeeds and returns the following substitution:

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

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

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

Theorem 8. (*unifyOp is complete*)
If $\Xi \vdash_{\mathsf{wf}} \forall\overline{\alpha}^k.\,\tau_1 \ : \ k$, $\Xi \vdash_{\mathsf{wf}} \tau_2 \ : \ k$, $\vdash \ \theta \ : \ \Xi \Rightarrow \Xi_2$ and $\theta(\forall\overline{\alpha}^k.\,\tau_1) \ = \ \forall\overline{\alpha}^k.\,\theta(\tau_2)$, then $\mathit{unifyOp}(\Xi,\ \forall\overline{\alpha}^k.\,\tau_1,\ \tau_2) \ = \ (\Xi_1,\ \theta_1)$ 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 Figures 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 $\mathit{infer}(\cdot,\ \cdot,\ \cdot,\ \cdot)$ and $\mathit{inferHandler}(\cdot,\ \cdot,\ \cdot,\ \cdot)$ 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 ϵ 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 Δ, *infer* yields the most general type by opening the closed effect row to μ_1. The *infer* function also yields an arbitrary effect μ_2 as x is a value. If x does not have a closed function type, *infer* generates a new type variable $\overline{\beta}^k$ 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 Δ 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.

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

The type of f is inferred to be $\forall\alpha\,\mu.\,\alpha \to \mu\,\alpha$, where μ 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 $\mathit{infer}(\Xi,\ \Gamma,\ (\Delta,\ f : \ \forall\alpha.\,\alpha \to \langle\,\rangle\,\alpha)\,,\ f\ 1)$.

```
$\mathit{infer}\ :\ (\mathsf{KCtx} \times \mathsf{TCtx} \times \mathsf{TCtx} \times \mathsf{Exp}) \rightarrow (\mathsf{KCtx} \times \mathsf{Subst} \times \mathsf{MType} \times \mathsf{Eff})$
$\mathit{infer}(\Xi,\ \Gamma,\ \Delta,\ x)\ =$
  $\mid x\ :\ \forall \overline{\alpha}^{k}.\ \tau_1 \rightarrow \langle l_1, .., l_n \rangle\ \tau_2\ \in \Delta$
      $\Rightarrow$ $\mathsf{assume}\ \overline{\beta}^{k},\ \mu_1$ and $\mu_2$ are fresh.
          $\mathsf{return}\ ((\Xi,\ \overline{\beta}^{k},\ \mu_1,\ \mu_2),\ id,\ id[\overline{\alpha}^{k} := \overline{\beta}^{k}](\tau_1 \rightarrow \langle l_1, .., l_n \mid \mu_1 \rangle\ \tau_2),\ \mu_2)$
  $\mid x\ :\ \forall \overline{\alpha}^{k}.\ \tau\ \in \Gamma,\ \Delta$
      $\Rightarrow$ $\mathsf{assume}\ \overline{\beta}^{k}$ and $\mu$ are fresh.
          $\mathsf{return}\ ((\Xi,\ \overline{\beta}^{k},\ \mu),\ id,\ id[\overline{\alpha}^{k} := \overline{\beta}^{k}]\tau,\ \mu)$

$\mathit{infer}(\Xi,\ \Gamma,\ \Delta,\ \lambda x.\ e)\ =$
    $\mathsf{assume}\ \alpha^*$ and $\mu$ are fresh.
    let $(\Xi_1,\ \theta_1,\ \tau_1,\ \epsilon_1)\ =\ \mathit{infer}((\Xi,\ \alpha^*),\ (\Gamma,\ x\ :\ \alpha^*),\ \Delta,\ e)$
    $\mathsf{return}\ (\Xi_1,\ \theta_1,\ \theta_1(\alpha^*) \rightarrow \epsilon_1\ \tau_1,\ \mu)$

$\mathit{infer}(\Xi,\ \Gamma,\ \Delta,\ e_1\ e_2)\ =$
    $\mathsf{assume}\ \alpha^*$ is fresh.
    let $(\Xi_1,\ \theta_1,\ \tau_1,\ \epsilon_1)\ =\ \mathit{infer}(\Xi,\ \Gamma,\ \Delta,\ e_1)$
    let $(\Xi_2,\ \theta_2,\ \tau_2,\ \epsilon_2)\ =\ \mathit{infer}(\Xi_1,\ \theta_1(\Gamma),\ \theta_1(\Delta),\ e_2)$
    let $(\Xi_3,\ \theta_3)\ =\ \mathit{unify}(\Xi_2,\ \theta_2(\tau_1),\ \tau_2 \rightarrow \epsilon_2\ \alpha^*)$
    let $(\Xi_4,\ \theta_4)\ =\ \mathit{unify}(\Xi_3,\ (\theta_3 \circ \theta_2)(\epsilon_1),\ \theta_3(\epsilon_2))$
    $\mathsf{return}\ (\Xi_4,\ \theta_4 \circ \theta_3 \circ \theta_2 \circ \theta_1,\ (\theta_4 \circ \theta_3)(\alpha^*),\ (\theta_4 \circ \theta_3)(\epsilon_2))$

$\mathit{infer}(\Xi,\ \Gamma,\ \Delta,\ \mathsf{let}\ x\ =\ v_1\ \mathsf{in}\ e_2)\ =$
    let $(\Xi_1,\ \theta_1,\ \tau_1,\ \epsilon_1)\ =\ \mathit{infer}(\Xi,\ \Gamma,\ \Delta,\ v_1)$
    let $(\Xi_2,\ \theta_2)\ =\ \mathit{unify}(\Xi_1,\ \epsilon_1,\ \langle\,\rangle)$
    let $\forall \overline{\alpha}^{k}.\ \theta_2(\tau_1)\ =\ \mathsf{Gen}((\theta_2 \circ \theta_1)(\Gamma),\ (\theta_2 \circ \theta_1)(\Delta),\ \theta_2(\tau_1))$
    let $\sigma\ =\ \mathsf{Close}(\forall \overline{\alpha}^{k}.\ \theta_2(\tau_1))$
    let $(\Xi_3,\ \theta_3,\ \tau_2,\ \epsilon_2)\ =\ \mathit{infer}(\Xi_2 \setminus \overline{\alpha}^{k},\ (\theta_2 \circ \theta_1)(\Gamma),\ ((\theta_2 \circ \theta_1)(\Delta),\ x\ :\ \sigma),\ e_2)$
    $\mathsf{return}\ (\Xi_3,\ (\theta_3 \circ \theta_2 \circ \theta_1),\ \tau_2,\ \epsilon_2)$

$\mathit{infer}(\Xi,\ \Gamma,\ \Delta,\ \mathsf{perform}\ op)\ =$
    $\mathsf{assume}\ \overline{\beta}^{k},\ \mu_1$ and $\mu_2$ are fresh.
    let $\forall \overline{\alpha}^{k}.\ \tau_1 \rightarrow \tau_2\ =\ \mathsf{Op}(\Sigma,\ op)$
    $\mathsf{return}\ ((\Xi,\ \overline{\beta}^{k},\ \mu_1,\ \mu_2),\ id,\ [\overline{\alpha}^{k} := \overline{\beta}^{k}](\tau_1 \rightarrow \mu_1\ \tau_2),\ \mu_2)$

$\mathit{infer}(\Xi,\ \Gamma,\ \Delta,\ \mathsf{handler}\ h)\ =$
    $\mathsf{assume}\ \mu$ is fresh.
    let $(\Xi_1,\ \theta_1,\ \tau_1,\ l_1,\ \epsilon_1)\ =\ \mathit{inferHandler}(\Xi,\ \Gamma,\ \Delta,\ h)$
    $\mathsf{return}\ ((\Xi,\ \mu),\ \theta_1,\ (() \rightarrow \langle l_1 \mid \epsilon_1 \rangle\ \tau_1) \rightarrow \epsilon_1\ \tau_1,\ \mu)$
```

Figure 9. Type Inference Algorithm

```
$\mathit{inferHandler} : (\mathsf{KCtx} \times \mathsf{TCtx} \times \mathsf{TCtx} \times \mathsf{Hnd}) \to (\mathsf{KCtx} \times \mathsf{Subst} \times \mathsf{MType} \times \mathsf{Lab} \times \mathsf{Eff})$
$\mathit{inferHandler}(\Xi, \Gamma, \Delta, \{op_1 \to v_1, \ldots, op_n \to v_n\}) =$
    $\mathsf{assume}\ \beta^*$ and $\mu$ are fresh.
    let $l = \mathsf{Label}(\Sigma, op_1)$
    $\mathsf{assert}\ \Sigma(l) = \{op_1, \ldots, op_n\}$
    let $(\Xi_0, \theta_0) = (\Xi, id)$
    for $i \in \{1, \ldots, n\}$
      $\mathsf{assume}\ \overline{\alpha}_i^k$ are fresh.
      let $\forall \overline{\alpha}_i^k. \tau_1^i \to \tau_2^i = \mathsf{Op}(\Sigma, op_i)$
      let $(\Xi_i^1, \theta_i^1, \tau_i, \epsilon_i) = \mathit{infer}(\Xi_{i-1}^3, \theta_{i-1}(\Gamma), \theta_{i-1}(\Delta), v_i)$
      let $(\Xi_i^2, \theta_i^2) = \mathit{unify}(\Xi_i^1, \epsilon_i, \langle\rangle)$
      let $(\Xi_i^3, \theta_i^3)$
        $= \mathit{unifyOp}(\Xi_i^2, (\theta_i^2 \circ \theta_i^1 \circ \theta_{i-1})(\forall \overline{\alpha}^k. \tau_i^1 \to \mu\,((\tau_i^2 \to \mu\,\beta^*) \to \mu\,\beta^*)), \theta_i^2(\tau_i))$
      let $\theta_i = \theta_i^3 \circ \theta_i^2 \circ \theta_i^1 \circ \theta_{i-1}$
      $\mathsf{assert}\ \overline{\alpha}_i^k \notin \mathsf{ftv}(\theta_i(\Gamma), \theta_i(\Delta))$
    $\mathsf{return}\ (\Xi_n^3, \theta_n, \theta_n(\beta^*), l, \theta_n(\mu))$
```

Figure 10. Type Inference Algorithm for Handlers

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

In the handler case, we use two auxiliary functions *inferHandler* and Label. The *inferHandler* function infers the type of a handler. It receives a quadruple (Ξ, Γ, Δ, 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, τ is the return type of the continuation captured by the handler h, l is the effect label handled by h, and ϵ is the rest of the effect row. The Label function returns the effect label corresponding to the given operation.

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

$$\Sigma = \{\, l_1 : \{\, op : \forall\alpha.\, \alpha \to \alpha \,\}\}$$
$$\mathsf{handler}\,\{\, op \to \lambda x\ \ k.\ k\,(x + 1)\,\}$$

This handler should be rejected for the following reason. First, the operation *op* is defined as having type $\forall\alpha.\,\alpha \to \alpha$. Therefore, the operation clause of *op* must have a type of the form $\forall\alpha.\,\alpha \to \mu\,((\alpha \to \mu\,\beta) \to \mu\,\beta)$, 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 $\mathsf{int} \to \mu\,((\mathsf{int} \to \mu\,\beta) \to \mu\,\beta)$, where the input and output types are a concrete type int. If we use *unifyOp*, unification of $\forall\alpha.\,\alpha \to \mu\,((\alpha \to \mu\,\beta) \to \mu\,\beta$ and $\mathsf{int} \to \mu\,((\mathsf{int} \to \mu\,\beta) \to \mu\,\beta)$ fails, because the bound variable α and 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 $\alpha \to \mu\,((\alpha \to \mu\,\beta) \to \mu\,\beta)$ to *unify*, which can be unified with $\mathsf{int} \to \mu\,((\mathsf{int} \to \mu\,\beta) \to \mu\,\beta)$.

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

Theorem 10. (*infer is complete with respect to syntax-directed inference rules*)
If $\vdash\ \theta\ :\ \Xi \Rightarrow \Xi_2$ and $\Xi_2 \mid \theta(\Gamma) \mid \theta(\Delta) \Vdash_{\mathsf{s}}\ e\ :\ \tau \mid \epsilon$,
then $\mathit{infer}(\Xi,\ \Gamma,\ \Delta,\ e)\ =\ (\Xi_1,\ \theta_1,\ \tau,\ \epsilon)$, 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 $\mathit{infer}(\Xi,\ \Gamma,\ \Delta,\ e)\ =\ (\Xi_1,\ \theta,\ \tau,\ \epsilon)$, 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 $\mathit{infer}(\Xi,\ \Gamma,\ \Delta,\ e)\ =\ (\Xi_1,\ \theta_1,\ \tau,\ \epsilon)$, 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 $\lceil\cdot\rceil$ in [4] that avoids implicitly instantiating variables. However, this choice gives a gap between the formalization and implmentation of Koka, because the current Koka implicitly opens closed functions types.

Acknowledgement We would like to thank the anonymous reviwers for thier 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^ϵ, which we call System F^ϵ+restrict, and a type-directed translation from ImpKoka to System F^ϵ+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^ϵ+restrict

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

We can prove the type soundness of Sysmtem F^ϵ+restrict by showing the follwing theorems.

Theorem 13. (*progress*)
If $\emptyset \vdash e_1 : \sigma \mid \langle\,\rangle$ then either e_1 is a value or $e_1 \longmapsto e_2$.

Theorem 14. (*preservation*)
If $\emptyset \vdash e_1 : \sigma \mid \langle\,\rangle$ and $e_1 \longmapsto e_2$ then $\emptyset \vdash e_2 : \sigma \mid \langle\,\rangle$.

$$\boxed{\underset{\uparrow}{\Xi} \mid \underset{\uparrow}{\Gamma} \mid \underset{\uparrow}{\Delta} \vdash \underset{\uparrow}{e} \;:\; \underset{\uparrow}{\sigma} \mid \underset{\uparrow}{\epsilon} \leadsto \underset{\downarrow}{e'} \qquad \underset{\uparrow}{\Xi} \mid \underset{\uparrow}{\Gamma} \mid \underset{\uparrow}{\Delta} \vDash \underset{\uparrow}{h} \;:\; \underset{\uparrow}{\tau} \mid \underset{\uparrow}{l} \mid \underset{\uparrow}{\epsilon} \leadsto \underset{\downarrow}{h'}}$$

$$\frac{\begin{array}{c} x \;:\; \sigma \in \Gamma, \Delta \quad \Xi \vdash_{\mathsf{wf}} \epsilon \;:\; \mathsf{eff} \\ \sigma \text{ not a closed function type} \end{array}}{\Xi \mid \Gamma \mid \Delta \vdash x \;:\; \sigma \mid \epsilon \leadsto x}\;[\textsc{Var}]$$

$$\frac{\begin{array}{c} f \;:\; \forall \overline{\alpha}^{\overline{k}}.\, \tau_1 \to \langle l_1, \ldots, l_n \rangle\, \tau_2 \in \Delta \\ \Xi \vdash_{\mathsf{wf}} \epsilon \;:\; \mathsf{eff} \quad \Xi \vdash_{\mathsf{wf}} \epsilon' \;:\; \mathsf{eff} \end{array}}{\begin{array}{c} \Xi \mid \Gamma \mid \Delta \vdash f \;:\; \forall \overline{\alpha}^{\overline{k}}.\, \tau_1 \to \langle l_1, \ldots, l_n \mid \epsilon \rangle\, \tau_2 \mid \epsilon' \\ \leadsto \Lambda \overline{\alpha}^{\overline{k}}.\, \lambda^{\langle l_1, \ldots, l_n \mid \epsilon \rangle}\, x \;:\; \tau_1.\, \mathsf{restrict}^{\langle l_1, \ldots, l_n \rangle}\, f[\overline{\alpha}^{\overline{k}}]\, x \end{array}}\;[\textsc{VarOpen}]$$

$$\frac{\Xi \mid \Gamma, x \;:\; \tau_1 \mid \Delta \vdash e \;:\; \tau_2 \mid \epsilon \leadsto e' \quad \Xi \vdash_{\mathsf{wf}} \tau_1 \;:\; * \quad \Xi \vdash_{\mathsf{wf}} \epsilon' \;:\; \mathsf{eff}}{\Xi \mid \Gamma \mid \Delta \vdash \lambda x.\, e \;:\; \tau_1 \to \epsilon\, \tau_2 \mid \epsilon' \leadsto \lambda^{\epsilon}\, x \;:\; \tau_1.\, e'}\;[\textsc{Lam}]$$

$$\frac{\Xi \mid \Gamma \mid \Delta \vdash e_1 \;:\; \tau_2 \to \epsilon\, \tau \mid \epsilon \leadsto e_1' \quad \Xi \mid \Gamma \mid \Delta \vdash e_2 \;:\; \tau_2 \mid \epsilon \leadsto e_2'}{\Xi \mid \Gamma \mid \Delta \vdash e_1\, e_2 \;:\; \tau \mid \epsilon \leadsto e_1'\, e_2'}\;[\textsc{App}]$$

$$\frac{\Xi, \alpha^k \mid \Gamma \mid \Delta \vdash v \;:\; \sigma \mid \langle\rangle \leadsto v' \quad k \not\equiv \mathsf{lab}}{\Xi \mid \Gamma \mid \Delta \vdash v \;:\; \forall \alpha^k.\, \sigma \mid \langle\rangle \leadsto \Lambda \alpha^k.\, v'}\;[\textsc{Gen}]$$

$$\frac{\Xi \mid \Gamma \mid \Delta \vdash e \;:\; \forall \alpha^k.\, \sigma \mid \epsilon \leadsto e' \quad \Xi \vdash_{\mathsf{wf}} \tau \;:\; k}{\Xi \mid \Gamma \mid \Delta \vdash e \;:\; \sigma[\alpha^k := \tau] \mid \epsilon \leadsto e'[\tau]}\;[\textsc{Inst}]$$

$$\frac{\Xi \mid \Gamma \mid \Delta \vdash v_1 \;:\; \sigma_1 \mid \langle\rangle \leadsto v_1' \quad \Xi \mid \Gamma \mid \Delta, x \;:\; \sigma_1 \vdash e_2 \;:\; \tau_2 \mid \epsilon \leadsto e_2'}{\Xi \mid \Gamma \mid \Delta \vdash \mathsf{let}\, x = v_1 \,\mathsf{in}\, e_2 \;:\; \sigma_2 \mid \epsilon \leadsto (\lambda^{\epsilon}\, x \;:\; \sigma_1.\, e_2')\, v_1'}\;[\textsc{Let}]$$

$$\frac{op \;:\; \forall \overline{\alpha}^{\overline{k}}.\, \tau_1 \to \tau_2 \in \Sigma(l) \quad \overline{\alpha}^{\overline{k}} \notin \Xi \quad \Xi \vdash_{\mathsf{wf}} \overline{\tau} \;:\; \overline{k} \quad \Xi \vdash_{\mathsf{wf}} \epsilon' \;:\; \mathsf{eff}}{\Xi \mid \Gamma \mid \Delta \vdash \mathsf{perform}\, op \;:\; (\tau_1 \to \langle l \mid \epsilon \rangle\, \tau_2)[\overline{\alpha}^{\overline{k}} := \overline{\tau}] \mid \epsilon' \leadsto \mathsf{perform}^{\epsilon}\, op\, \overline{\tau}}\;[\textsc{Perform}]$$

$$\frac{\Xi \mid \Gamma \mid \Delta \vDash h \;:\; \tau \mid l \mid \epsilon \leadsto h' \quad \Xi \vdash_{\mathsf{wf}} \epsilon' \;:\; \mathsf{eff}}{\Xi \mid \Gamma \mid \Delta \vdash \mathsf{handler}\, h \;:\; (() \to \langle l \mid \epsilon \rangle\, \tau) \to \epsilon\, \tau) \mid \epsilon' \leadsto \mathsf{handler}^{\epsilon}\, h'}\;[\textsc{Handler}]$$

$$\frac{\begin{array}{c} op_i \;:\; \forall \overline{\alpha}_i^{\overline{k}}.\, \tau_1^i \to \tau_2^i \in \Sigma(l) \quad \overline{\alpha}_i^{\overline{k}} \notin \mathsf{ftv}(\epsilon, \tau) \\ \Xi \mid \Gamma \mid \Delta \vdash v_i \;:\; \forall \overline{\alpha}_i^{\overline{k}}.\, \tau_1^i \to \epsilon\, ((\tau_2^i \to \epsilon\, \tau) \to \epsilon\, \tau) \mid \langle\rangle \leadsto v_i' \end{array}}{\Xi \mid \Gamma \mid \Delta \vDash \{ op_1 \to v_1, \ldots, op_n \to v_n \} \;:\; \tau \mid l \mid \epsilon \leadsto \{ op_1 \to v_1', \ldots, op_n \to v_n' \}}\;[\textsc{Ops}]$$

Figure 11. Translation to System F^{ϵ}+restrict

Types

$$\begin{array}{rcll} \sigma & ::= & \alpha^k & \text{(type variables of kind } k\text{)} \\ & \mid & c^k\,\tau\ldots\tau & \text{(type constructor of kind } k\text{)} \\ & \mid & \sigma \rightarrow \epsilon\,\sigma & \text{(function type)} \\ & \mid & \forall\alpha^k.\,\sigma & \text{(quantified type)} \end{array}$$

Kinds

$$\begin{array}{rcll} k & ::= & * & \text{(value type)} \\ & \mid & k \rightarrow k & \text{(type constructors)} \\ & \mid & \mathsf{eff} & \text{(effect type } (\mu,\epsilon)\text{)} \\ & \mid & \mathsf{lab} & \text{(basic effect } (l)\text{)} \end{array}$$

$$\begin{array}{llcll} \text{Effect signature} & sig & ::= & \{\, op_1 \;:\; \forall\overline{\alpha}_1.\,\sigma_1 \rightarrow \sigma_1',\;\ldots,\; op_n \;:\; \forall\overline{\alpha}_n.\,\sigma_n \rightarrow \sigma_n' \} & \\ \text{Effect signatures} & \Sigma & ::= & \{ l_1 \;:\; sig_1,\;\ldots,\; l_n \;:\; sig_n \} & \\ \text{Type Constructors} & \langle\,\rangle & : & \mathsf{eff} & \text{empty effect row (total)} \\ & \langle\_ \mid \_\rangle & : & \mathsf{lab} \rightarrow \mathsf{eff} \rightarrow \mathsf{eff} & \text{effect row extension} \end{array}$$

$$\begin{array}{llcl} \text{Syntax} & \langle l_1,\ldots,l_n\rangle & \doteq & \langle l_1 \mid \ldots \mid \langle l_n \mid \langle\,\rangle\rangle \ldots\rangle \\ & \langle l_1,\ldots,l_n \mid \mu\rangle & \doteq & \langle l_1 \mid \ldots \mid \langle l_n \mid \mu\rangle \ldots\rangle \\ & \multicolumn{3}{l}{\epsilon ::= \sigma^{\mathsf{eff}}, \quad \mu ::= \alpha^{\mathsf{eff}}, \quad l ::= c^{\mathsf{lab}}} \end{array}$$

Figure 12. Types of System F^ϵ+$\mathsf{restrict}$

Expressions

$$\begin{array}{rcll} e & ::= & v & \text{(value)} \\ & \mid & e\,e & \text{(application)} \\ & \mid & e[\sigma] & \text{(type application)} \\ & \mid & \mathsf{handle}\,h\,e & \text{(handler instance)} \\ & \mid & \mathsf{restrict}^{\langle l_1,\ldots,l_n\rangle}\;v & \text{(restrict)} \end{array}$$

Values

$$\begin{array}{rcll} v & ::= & x & \text{(variables)} \\ & \mid & \lambda^\epsilon x \;:\; \sigma.\,e & \text{(abstraction)} \\ & \mid & \Lambda\alpha^k.\,e & \text{(type abstraction)} \\ & \mid & \mathsf{handler}^\epsilon\;h & \text{(effect handler)} \\ & \mid & \mathsf{perform}^\epsilon\;op\,\overline{\sigma} & \text{(operation)} \end{array}$$

$$\begin{array}{llcl} \text{Handlers} & h & ::= & \{ op_1 \rightarrow f_1,\;\ldots,\; op_n \rightarrow f_n \} \\ \text{Evaluation Context} & \mathsf{F} & ::= & \square \mid \mathsf{F}\,e \mid v\,\mathsf{F} \mid \mathsf{F}\,\sigma \\ & & \mid & \mathsf{restrict}^{\langle l_1,\ldots,l_n\rangle}\;\mathsf{F} \\ & \mathsf{E} & ::= & \square \mid \mathsf{E}\,e \mid v\,\mathsf{E} \mid \mathsf{E}\,\sigma \\ & & \mid & \mathsf{handle}\,h\,\mathsf{E} \mid \mathsf{restrict}^{\langle l_1,\ldots,l_n\rangle}\;\mathsf{E} \end{array}$$

$$\begin{array}{llcl} (\mathit{app}) & (\lambda^\epsilon x \;:\; \sigma.\,e)\;v & \longrightarrow & e[x{:=}v] \\ (\mathit{handler}) & (\mathsf{handler}^\epsilon\;h)\;v & \longrightarrow & \mathsf{handle}\,h\,(v\,()) \\ (\mathit{return}) & \mathsf{handle}\,h\,v & \longrightarrow & v \\ (\mathit{perform}) & \mathsf{handle}\,h\,\mathsf{E}[\mathsf{perform}\;op\,\overline{\sigma}\;v] & \longrightarrow & f[\overline{\sigma}]\;v\,k \quad \text{iff } op \notin \mathsf{bop}(\mathsf{E}) \land (op \rightarrow f) \in h \\ & & & \text{where } op \;:\; \forall\overline{\alpha}.\,\sigma_1 \rightarrow \sigma_2 \in \Sigma(l) \\ (\mathit{restrict}) & \mathsf{restrict}^{\langle l_1,\ldots,l_n\rangle}\;v & \longrightarrow & v \end{array}$$

$$\frac{e \longrightarrow e'}{\mathsf{E}[e] \longmapsto \mathsf{E}[e']}\;[\textsc{step}]$$

$$\begin{array}{lcl} \mathsf{bop}(\square) & = & \emptyset \\ \mathsf{bop}(\mathsf{E}\,e) & = & \mathsf{bop}(\mathsf{E}) \\ \mathsf{bop}(v\,\mathsf{E}) & = & \mathsf{bop}(\mathsf{E}) \\ \mathsf{bop}(\mathsf{handle}\,h\,\mathsf{E}) & = & \mathsf{bop}(\mathsf{E}) \cup \{ op \mid (op \rightarrow f) \in h \} \\ \mathsf{bop}(\mathsf{restrict}^{\langle l_1,\ldots,l_n\rangle}\;\mathsf{E}) & = & \mathsf{bop}(\mathsf{E}) \end{array}$$

Figure 13. Expressions and Operational Sematics of F^ϵ+$\mathsf{restrict}$

$$\underset{\uparrow}{\Gamma} \vdash \underset{\underset{\mathsf{F}^{\epsilon}}{\uparrow}}{e} : \underset{\downarrow}{\sigma} \mid \underset{\uparrow}{\epsilon} \qquad \underset{\uparrow}{\Gamma} \vdash_{\mathsf{val}} \underset{\underset{\mathsf{F}^{\epsilon}}{\uparrow}}{v} : \underset{\downarrow}{\sigma} \qquad \underset{\uparrow}{\Gamma} \vdash_{\mathsf{ops}} \underset{\underset{\mathsf{F}^{\epsilon}}{\uparrow}}{h} : \underset{\downarrow}{\sigma} \mid \underset{\downarrow}{l} \mid \underset{\uparrow}{\epsilon}$$

$$\frac{\Gamma \vdash e : \sigma \mid \langle l_1, \ldots, l_n \rangle \quad \Gamma \vdash_{\mathsf{wf}} \epsilon : \mathsf{eff}}{\Gamma \vdash \mathsf{restrict}^{\langle l_1, \ldots, l_n \rangle}\, e : \sigma \mid \langle l_1, \ldots, l_n \mid \epsilon \rangle}\ [\textsc{Restrict}] \qquad \frac{x : \sigma \in \Gamma}{\Gamma \vdash_{\mathsf{val}} x : \sigma}\ [\textsc{Var}]$$

$$\frac{\Gamma, x : \sigma_1 \vdash e : \sigma_2 \mid \epsilon \quad \Gamma \vdash_{\mathsf{wf}} \sigma_1 : *}{\Gamma \vdash_{\mathsf{val}} \lambda^{\epsilon}\, x : \sigma_1 .\, e : \sigma_1 \to \epsilon\, \sigma_2}\ [\textsc{Lam}] \quad \frac{\Gamma \vdash_{\mathsf{val}} v : \sigma \quad \Gamma \vdash_{\mathsf{wf}} \epsilon : \mathsf{eff}}{\Gamma \vdash v : \sigma \mid \epsilon}\ [\textsc{Val}]$$

$$\frac{\Gamma \vdash e_1 : \sigma_2 \to \epsilon\, \sigma \mid \epsilon \quad \Gamma \vdash e_2 : \sigma_2 \mid \epsilon}{\Gamma \vdash e_1\, e_2 : \sigma \mid \epsilon}\ [\textsc{App}]$$

$$\frac{\Gamma \vdash e : \sigma \mid \langle\, \rangle \quad \alpha^k \notin \mathsf{ftv}(\Gamma) \quad k \not\equiv \mathsf{lab}}{\Gamma \vdash_{\mathsf{val}} \Lambda \alpha^k . e : \forall \alpha .\, \sigma}\ [\textsc{TAbs}]$$

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

$$\frac{op : \forall \overline{\alpha}^{\overline{k}} .\, \sigma_1 \to \sigma_2 \in \Sigma(l) \quad \overline{\alpha}^{\overline{k}} \notin \mathsf{ftv}(\Gamma) \quad \Gamma \vdash_{\mathsf{wf}} \epsilon : \mathsf{eff}}{\Gamma \vdash_{\mathsf{val}} \mathsf{perform}^{\epsilon}\, \mathsf{op}\, \overline{\sigma} : \sigma_1[\overline{\alpha}^{\overline{k}} := \overline{\sigma}] \to \langle l \mid \epsilon \rangle\, \sigma_2[\overline{\alpha}^{\overline{k}} := \overline{\sigma}]}\ [\textsc{Perform}]$$

$$\frac{\Gamma \vdash_{\mathsf{ops}} h : \sigma \mid l \mid \epsilon \quad \Gamma \vdash e : \sigma \mid \langle l \mid \epsilon \rangle}{\Gamma \vdash \mathsf{handle}^{\epsilon}\, h\, e : \sigma \mid \epsilon}\ [\textsc{Handle}]$$

$$\frac{\Gamma \vdash_{\mathsf{ops}} h : \sigma \mid l \mid \epsilon}{\Gamma \vdash_{\mathsf{val}} \mathsf{handler}^{\epsilon}\, h : (() \to \langle l \mid \epsilon \rangle\, \sigma) \to \epsilon\, \sigma}\ [\textsc{Handler}]$$

$$\frac{op_i : \forall \overline{\alpha}^{\overline{k}} .\, \sigma_1 \to \sigma_2 \in \Sigma(l) \quad \overline{\alpha}^{\overline{k}} \notin \mathsf{ftv}(\Gamma) \quad \Gamma \vdash_{\mathsf{val}} v_i : \forall \overline{\alpha}^{\overline{k}} .\, \sigma_1 \to \epsilon\, ((\sigma_2 \to \epsilon\, \sigma) \to \sigma)}{\Gamma \vdash_{\mathsf{ops}} \{ op_1 \to v_1, \ldots, op_n \to v_n \} : \sigma \mid l \mid \epsilon}\ [\textsc{OPs}]$$

Figure 14. Typing Rules of F^{ϵ}+restrict

B. Type-Directed Translation to System F$^\epsilon$ + restrict

In Figure 11, we next define the type-directed translation from ImpKoka to System F$^\epsilon$+restrict. The judgment $\Xi \mid \Gamma \mid \Delta \vdash e : \sigma \mid \epsilon \rightsquigarrow e'$ states that an expression e has type σ and effect ϵ under the type variable context Ξ and typing contexts Γ and Δ, and translates to an expression e' of System F$^\epsilon$+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] Andrej Bauer, and Matija Pretnar. “Programming with Algebraic Effects and Handlers.” *Journal of Logical and Algebraic Methods in Programming* 84 (1). Elsevier: 108–123. 2015.

[2] Lukas Convent, Sam Lindley, Conor McBride, and Craig McLaughlin. “Doo Bee Doo Bee Doo.” *In the Journal of Functional Programming*, January. Jan. 2020. To appear in the special issue on algebraic effects and handlers.

[3] Luís Damas, and Robin Milner. “Principal Type-Schemes for Functional Programs.” In *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages, Albuquerque, New Mexico, USA, January 1982*, edited by Richard A. DeMillo, 207–212. ACM Press. 1982. doi:10.1145/582153.582176.

[4] Frank Emrich, Sam Lindley, Jan Stolarek, James Cheney, and Jonathan Coates. “FreezeML: Complete and Easy Type Inference for First-Class Polymorphism.” In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, edited by Alastair F. Donaldson and Emina Torlak, 423–437. ACM. 2020. doi:10.1145/3385412.3386003.

[5] Daniel Hillerström, and Sam Lindley. “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*, edited by James Chapman and Wouter Swierstra, 15–27. ACM. 2016. doi:10.1145/2976022.2976033.

[6] Mark P. Jones. “A Theory of Qualified Types.” In *4th. European Symposium on Programming (ESOP’92)*, 582:287–306. Lecture Notes in Computer Science. Springer-Verlag, Rennes, France. Feb. 1992. doi:10.1007/3-540-55253-7_17.

[7] Ohad Kammar, and Gordon D. Plotkin. “Algebraic Foundations for Effect-Dependent Optimisations.” In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 349–360. POPL ’12. ACM, New York, NY, USA. 2012. doi:10.1145/2103656.2103698.

[8] Georgios Karachalias, Matija Pretnar, Amr Hany Saleh, Stien Vanderhallen, and Tom Schrijvers. “Explicit Effect Subtyping.” *J. Funct. Program.* 30: e15. 2020. doi:10.1017/S0956796820000131.

[9] Daan Leijen. “Extensible Records with Scoped Labels.” In *Proceedings of the 2005 Symposium on Trends in Functional Programming*, 297–312. 2005.

[10] Daan Leijen. "HMF: Simple Type Inference for First-Class Polymorphism." In *Proceedings of the 13th ACM Symposium of the International Conference on Functional Programming*. ICFP'08. Victoria, Canada. Sep. 2008. doi:10.1145/1411204.1411245.

[11] Daan Leijen. "Koka: Programming with Row Polymorphic Effect Types." In *MSFP'14, 5th Workshop on Mathematically Structured Functional Programming*. 2014. doi:10.4204/EPTCS.153.8.

[12] Daan Leijen. "Type Directed Compilation of Row-Typed Algebraic Effects." In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL'17)*, 486–499. Paris, France. Jan. 2017. doi:10.1145/3009837.3009872.

[13] Sam Lindley, Connor McBride, and Craig McLaughlin. "Do Be Do Be Do." In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL'17)*, 500–514. Paris, France. Jan. 2017. doi:10.1145/3009837.3009897.

[14] Simon Peyton Jones, Geoffrey Washburn, and Stephanie Weirich. *Wobbly Types: Type Inference For Generalised Algebraic Data Types.* MS-CIS-05-26. Jul. 2004. Microsoft Research.

[15] Gordon D. Plotkin, and John Power. "Algebraic Operations and Generic Effects." *Applied Categorical Structures* 11 (1): 69–94. 2003. doi:10.1023/A:1023064908962.

[16] Matija Pretnar. "Inferring Algebraic Effects." *Log. Methods Comput. Sci.* 10 (3). 2014. doi:10.2168/LMCS-10(3:21)2014.

[17] Didier Rémy. "Type Inference for Records in Natural Extension of ML." In *Theoretical Aspects of Object-Oriented Programming*, 67–95. 1994. doi:10.1.1.48.5873.

[18] John Alan Robinson. "A Machine-Oriented Logic Based on the Resolution Principle." *J. ACM* 12 (1): 23–41. 1965. doi:10.1145/321250.321253.

[19] Andrew K. Wright. "Simple Imperative Polymorphism." *LISP Symb. Comput.* 8 (4): 343–355. 1995.

[20] Ningning Xie, Jonathan Brachthauser, Daniel Hillerstrom, Philipp Schuster, and Daan Leijen. "Effect Handlers, Evidently." In *The 25th ACM SIGPLAN International Conference on Functional Programming (ICFP)*. ACM SIGPLAN. Aug. 2020. `https://www.microsoft.com/en-us/research/publication/effect-handlers-evidently/`.

[21] Ningning Xie, and Daan Leijen. "Generized 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)*. ICFP '21. Virtual. Aug. 2021.