Towards Efficient Adjustment of Effect Rows

Naoya Furudono, Youyou Cong, Hidehiko Masuhara, and Daan Leijen

[1] Tokyo Institute of Technology, Tokyo, Japan
naoyafurudono@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 with native support for algebraic effects and handlers. To implement effect handler operations efficiently, Koka employs a semantics where the handlers in scope are passed down to each function as an evidence vector. At runtime, these evidence vectors are adjusted using the open constructs to match the evidence for a particular function. All these adjustments can cause significant runtime overhead. In this paper, we present a novel transformation on the Koka core calculus that we call *open floating*. This transformation aims to float up open constructs and combine them in order to minimize the adjustments needed at runtime. Open floating improves performance by 2.5× in an experiment. Furthermore, we formalize an aspect of row-based effect typing, including the *closed prefix* relation on effect rows, which clarifies the constraint on open floating.

Keywords: Algebraic effect and handlers · Type and effect system · Compiler optimization

1. Introduction

Algebraic effect handlers [14] are a language feature for user-defined effects. For instance, using effect handlers, we can support exception, asynchronous programming[8], nondeterminism, and so on, not as builtin features but as libraries. While effect handlers are convenient, they incur more runtime overhead compared to native effects. In order to fill in the gap of the performance, suitable semantics have been explored [2, 15, 19, 20].

In this study, we focus on the evidence passing semantics [20], which is employed by the Koka language [7, 9, 20]. The key idea of the semantics is to pass around a vector of handler implementations, called an *evidence vector*, and propagate it to algebraic effect operation calls, exposing optimization opportunities. The row-based type-and-effect system ensures the correctness of the dynamic semantics, where the static effect row type corresponds directly to the shape of the dynamic evidence vector at runtime.

The Koka compiler automatically inserts open constructs during type inference to adjust evidence vectors at runtime. Unfortunately, each adjustment incurs runtime cost and, being type-directed, the automatic insertion tends to generate many redundant open calls around function applications.

In this paper, we present the *open floating* algorithm to remove such opens as a compile-time optimization. The algorithm first removes existing opens in a top-down way and then re-assigns effect adjustment constructs back in a bottom-up way. Unlike type inference phase insertion, the re-assignment is driven by the now explicit effect types, ensuring preservation of the meaning of programs. To give the reader a rough idea of the algorithm, we present programs before and after open floating.

$$
\begin{array}{l}
\mathsf{handler}\,\{\,\mathit{ask}\mapsto \lambda x.\,\lambda k.\,k\,3\,\}\,\lambda\_. \\
\quad \mathsf{let}\,x = \mathsf{open}\,\langle \mathit{read}\rangle\,\mathit{safediv}(3,2)\,\mathsf{in} \\
\quad \mathsf{let}\,y = \mathsf{open}\,\langle \mathit{read}\rangle\,\mathit{safediv}(3,x)\,\mathsf{in} \\
\quad \mathsf{open}\,\langle \mathit{read}\rangle\,\mathit{safediv}(3,\,y)
\end{array}
\quad\rightsquigarrow\quad
\begin{array}{l}
\mathsf{handler}\,\{\,\mathit{ask}\mapsto \lambda x.\,\lambda k.\,k\,3\,\}\,\lambda\_. \\
\quad \mathsf{restrict}\,\langle\,\rangle\,(\\
\quad\quad \mathsf{let}\,x = \mathit{safediv}(3,2)\,\mathsf{in} \\
\quad\quad \mathsf{let}\,y = \mathit{safediv}(3,x)\,\mathsf{in} \\
\quad\quad \mathit{safediv}(3,\,y))
\end{array}
$$

The one on the right is faster than the one on the left, because the former has less adjustment operations open and restrict. Both of them perform the same adjustment.

The difference is that open is used solely to functions, while restrict can be applied to general expressions.

This is essential to share a single adjustment over multiple function applications.

In Section 2, we give an overview of our study, and in the subsequent sections, we discuss the following contributions.

- We define System F_{pwo}, a system of effect handlers with the open construct. The system is a core calculus of the Koka language, and is an extension of System F_{pw} by Xie and Leijen [20]. We elaborate on effect typing in the extended system, showing
  - the way the type system checks the use of effect handlers (Section 3.3)
  - an advantage of using rows for effect types rather than sets, which are popular in effect handler calculi [1, 16] (Section 3.4)
  - an effect type restriction on open and restrict, which we call the *closed prefix* relation (Section 3.4).

  In particular, the closed prefix relation clarifies what program transformations are allowed. This helps us define open floating.
- We give a definition of the open floating algorithm, which floats up redundant opens (Section 4) by analyzing the effect type of expressions.
- We implemented open floating in the Koka compiler [11].
- We evaluated open floating via a preliminary benchmark (Section 5), which improved performance by 2.5×. Based on the results, we make it clear what kind of programs benefit from open floating.

We discuss future work in Section 6 and related work in Section 7.

2. Overview

In this section, we give an overview of this thesis. We first give an overview of the calculus of study, and explain the need for open. We then show an example with redundant opens and describe the idea of our solution.

2.1. Effect Handlers

Algebraic effects are declared with an effect label l and a list of operation signatures. For instance, suppose we have a *read* effect with a single operation *ask*, which takes a unit argument and returns an integer value.

$$\mathit{read} \ : \ \{\, \mathit{ask} \ : \ () \to \mathit{int} \,\}$$

An effect handler for *read* specifies what the *ask* operation should do when it is called in the handled expression.

$$\mathsf{handler}\,\{\, \mathit{ask} \to \lambda\_.\, \lambda k.\, k\,3 \,\}\, \lambda\_.\, \mathsf{perform}\ \mathit{ask}\,() \ + \ \mathsf{perform}\ \mathit{ask}\,()$$

In this example, $\mathsf{perform}\ \mathit{ask}\,()$ calls operation *ask* with argument $()$, which is evaluated to 3 according to the handler. The behavior of an operation call is formalized as follows: (1) find the innermost handler of the effect, (2) capture the *resumption* – continuation delimited by the handler –, and (3) apply the handler clause to the argument and the resumption.

1. $\mathsf{handler}\,\{\, \mathit{ask} \to \lambda\_.\, \lambda k.\, k\,3 \,\}\, \lambda\_.\, \mathsf{perform}\ \mathit{ask}\,() \ + \ \mathsf{perform}\ \mathit{ask}\,()$
2. $\mathit{resume} \ = \ \lambda z.\mathsf{handler}\,\{\, \mathit{ask} \to \lambda\_.\, \lambda k.\, k\,3 \,\}\, \lambda\_.\, z \ + \ \mathsf{perform}\ \mathit{ask}\,()$
3. $(\lambda\_.\, \lambda k.\, k\,3)\,()\ \mathit{resume}$

The handler resumes the resumption with argument 3, so that the example is evaluated to the following.

$$\mathsf{handler}\,\{\, \mathit{ask} \to \lambda\_.\, \lambda k.\, k\,3 \,\}\, \lambda\_.\, 3 \ + \ \mathsf{perform}\ \mathit{ask}\,()$$

The two occurrences of $\mathsf{perform}\ \mathit{ask}\,()$ are both evaluated to 3, therefore the whole expression is evaluated to 6.

Using effect handlers, it is easy to combine different effects in a single program. Let us combine *read* with the exception effect *exn*, which has an operation *throw* of type $\forall\alpha.\mathit{int} \to \alpha$

$$\mathit{exn} \ : \ \{\, \mathit{throw} \ : \ \forall\alpha.\mathit{int} \to \alpha \,\}$$

Using a handler for *exn* as we did for *read*, we can perform the *throw* operation.

$$\begin{array}{l}\mathsf{handler}\,\{\, \mathit{ask} \mapsto \lambda x.\, \lambda k.\, k\,3 \,\}\, (\lambda\_. \\ \quad \mathsf{handler}\,\{\, \mathit{throw} \mapsto \lambda x.\, \lambda k.\, x \,\}\, (\lambda\_. \\ \quad\quad \mathsf{perform}\ \mathit{ask}\,() \ + \ \mathsf{perform}\ \mathit{ask}\,() \ + \ \mathsf{perform}\ \mathit{throw}\ 1))\end{array}$$

In this program, $\mathsf{perform}\ \mathit{ask}\,()$ is evaluated to 3 as before, but the entire expression is evaluated to 1 due to $\mathsf{perform}\ \mathit{throw}\ 1$. The handler for the *throw* operation discards the resumption k and returns the argument x, which exits the computation out of the handler expression.

2.2. Evidence Passing Semantics and Row-based Effect System

Among different formalizations of the dynamic semantics of effect handlers, we adopt the *evidence passing semantics* (EPS) [20], which exploits the invariants derived from the row-based effect system and allows the compiler to translate programs to efficient code.

Under the EPS, we pass all handlers in scope to the handled expression so that operation calls can access their handler locally. The operation clauses passed to expressions are represented as an *evidence vector* [19]. For instance, in the *read* and *exn* examples discussed above, the expression $\mathsf{perform}\ \mathit{ask}\,()$ is performed with the evidence vector of the form $\langle\!\langle\, \mathit{exn} : \{\mathit{throw} \mapsto \ldots\},\ \mathit{read} : \{\mathit{ask} \mapsto \ldots\} \,\rangle\!\rangle$. Xie and Leijen [20] show that EPS often allow us to avoid lookups and resumption captures, which are one of main sources of inefficiency in effect handler execution.

The static semantics of our calculus is defined by a *row-based effect system*. In the effect system, every expression is related to an effect row type in addition to a usual type. Effect rows indicate what kind of evidence vector is provided from the context to evaluate the expression. For instance, in the example with *read* and *exn*, the expression $\mathsf{perform}\ \mathit{ask}\,() + \mathsf{perform}\ \mathit{ask}\,() + \mathsf{perform}\ \mathit{throw}\ 1$ has type *int* and effect row $\langle \mathit{exn}, \mathit{read} \rangle$.

The typing rules maintain the correspondence between evidence vectors and effect rows. For instance, a function application uses the same evidence vector for the function, the argument, and the β-reduced expression. Correspondingly, the typing rule for function application requires the effect rows of the three parts (occurrences of ϵ in the premises) to agree with that of the entire expression (occurrence of ϵ in the conclusion).

$$\frac{\Gamma \vdash e_1 : \sigma_1 \to \epsilon\, \sigma_2 \mid \epsilon \qquad \Gamma \vdash e_2 : \sigma_1 \mid \epsilon}{\Gamma \vdash e_1\, e_2 : \sigma_2 \mid \epsilon}\ [\textsc{app}]$$

We ignore the order of labels in effect rows (except for *parameterized effect labels* that are discussed in Section 3.1.2 and Section 3.4). For instance, we regard two rows $\langle \mathit{exn}, \mathit{read} \rangle$ and $\langle \mathit{read}, \mathit{exn} \rangle$ as equivalent. This flexible row equivalence allows the type system to ignore the order of handlers in evaluation contexts. As a consequence, both programs below are judged well-typed as one would expect.

$$\mathsf{handler}\,\{\, \mathit{ask} \mapsto \lambda x.\, \lambda k.\, k\,3 \,\}\,(\lambda\_.\ \mathsf{handler}\,\{\, \mathit{throw} \mapsto \lambda x.\, \lambda k.\, x \,\}\, f)$$

$$\mathsf{handler}\,\{\, \mathit{throw} \mapsto \lambda x.\, \lambda k.\, x \,\}\,(\lambda\_.\ \mathsf{handler}\,\{\, \mathit{ask} \mapsto \lambda x.\, \lambda k.\, k\,3 \,\}\, f)$$

where $\quad f = \lambda\_.\mathsf{perform}\ \mathit{ask}\,() + \mathsf{perform}\ \mathit{ask}\,() + \mathsf{perform}\ \mathit{throw}\ 1$

We formally define the effect row equivalence in Section 3.1.2 and discuss it in Section 3.4.

2.3. Effect Type Adjustment for Function Types

The typing rule [APP] is too restrictive on some occasions. Consider a function *safediv* of type $(\mathit{int}, \mathit{int}) \to \langle\,\rangle\, \mathit{maybe}\langle \mathit{int} \rangle$, which returns *Nothing* if the divider

is 0, instead of throwing an exception. This function causes no effect, hence we should be able to call the function in any context. However, the type system prevents us from calling *safediv* in certain contexts. For instance, the following expression is judged ill-typed.

$$\mathsf{handler}\,\{\,\mathit{ask}\mapsto \lambda x.\,\lambda k.\,k\,3\,\}\,\lambda\_.\mathit{safediv}(3,\,2)$$

The expression $\mathit{safediv}(3, 2)$ expects an evidence vector of type $\langle\,\rangle$, whereas the context provides an evidence vector of type $\langle \mathit{read}\rangle$. Due to this inconsistency, the expression is rejected by the type system.

To call functions with a "smaller" effect, we introduce the expression $\mathsf{open}\,\epsilon'\,v$ into the calculus. At compile time, open allows a function to have a "bigger" effect type according to the following typing rule.

$$\frac{\Gamma \vdash_{\mathsf{val}}\ v\ :\ \sigma_1 \to \epsilon\,\sigma_2 \qquad \epsilon \leqslant \epsilon'}{\Gamma \vdash_{\mathsf{val}}\ \mathsf{open}\,\epsilon'\,v\ :\ \sigma_1 \to \epsilon'\,\sigma_2}\ [\textsc{open}]$$

At runtime, open adjusts evidence vectors so that the callee receives an evidence vector of the expected shape and thus runs correctly. Using open, we can make the above example well-typed.

$$\mathsf{handler}\,\{\mathit{ask}\mapsto \lambda x.\,\lambda k.\,k3\}\,\lambda\_.(\mathsf{open}\,\langle \mathit{read}\rangle\,\mathit{safediv})(3,\,2)$$

We call the smaller effect row a *closed prefix* of the larger one. The closed prefix relation is defined as:

$$\langle l_1, \ldots, l_n\,\rangle \leqslant \langle l_1, \ldots, l_n \mid \epsilon\,\rangle \quad (n \geq 0)$$

It turns out that different formulations are possible but some seemingly benign generalizations can make the type system unsound –

we discuss the closed prefix relation in detail in Section 3.4.

2.4. Motivating Open Floating

Our calculus is designed as an intermediate language of a compiler. This means the user does not need to explicitly write opens; they are automatically inserted by the type inferencer. The user expression $\mathsf{handler}\,\{\,\mathit{ask}\mapsto \ldots\,\}\,\lambda\_.\mathit{safediv}(3,2)$ is translated to explicitly typed $\mathsf{handler}^{\langle\,\rangle}\,\{\,\mathit{ask}\mapsto \ldots\,\}\,\lambda\_.(\mathsf{open}\,\langle \mathit{read}\rangle\,\mathit{safediv})\,(3,2)$, for example.

Unfortunately, naive insertion of opens makes programs inefficient. Consider a program that calls *safediv* three times. The compiler inserts opens into each function call as follows.

$$\begin{aligned}&\mathsf{handler}^{\langle\,\rangle}\,\{\,\mathit{ask}\mapsto \lambda x\,:\,\mathit{int}.\,\lambda k\,:\,\mathit{int}\to\langle\,\rangle\,\mathit{int}.\,k\,3\,\}\,\lambda\_.\\ &\quad \mathsf{let}\,x\,=\,\mathsf{open}\,\langle \mathit{read}\rangle\,\mathit{safediv}(3,2)\,\mathsf{in}\\ &\quad \mathsf{let}\,y\,=\,\mathsf{open}\,\langle \mathit{read}\rangle\,\mathit{safediv}(3,x)\,\mathsf{in}\\ &\quad \mathsf{open}\,\langle \mathit{read}\rangle\,\mathit{safediv}(3,\,y)\end{aligned}$$

As open causes evidence vector adjustment at runtime, having many open calls makes execution slow. In order to avoid this inefficiency, we design the *open*

floating optimization that eliminates redundant open calls. By open floating, the above program is transformed to the following one.

```
handler⟨⟩ { ask↦ λx : int. λk : int→⟨⟩ int. k3 } λ⟨read⟩_.
  restrict ⟨⟩ (
    let x = safediv(3, 2) in
    let y = safediv(3, x) in
    safediv(3, y))
```

Here, restrict ϵ e allows e to be typed with effect ϵ, which is smaller than the effect of the context. In this particular example, e is typed with $\langle\,\rangle$, not $\langle read\rangle$. At runtime, as open does, restrict ϵ e changes the shape of the evidence vector to fit ϵ and pass it to e.

In general, open floating erases open in a β-redex and re-assigns appropriate open or restrict to make the whole expression type check in a bottom-up way. The closed prefix relation plays an essential role in determining the new effect type of each sub-expression.

We formally define open, restrict, and the closed prefix relation in Section 3, present the open floating algorithm in Section 4, and discuss a preliminary benchmark in Section 5.

3. System F_{pwo}

In this section, we present System F_{pwo}, a calculus with algebraic effect handlers and the open. The calculus is an extension of System F_{pw} [20], an explicitly typed polymorphic lambda calculus with effect handlers. The semantics is based on *evidence passing semantics* [20], which leads to an efficient implementation of effect handlers. Furthermore, both calculi have row-based effect types, which denote the static meaning of evidence vectors. We extend F_{pw} with open, restrict and parameterized effect labels. These features have previously been discussed by Leijen [10]. In that work, the idea of open was formalized as a typing rule, and parameterized effect labels were formalized in a way that does not fit well to our calculus.

We have confirmed the soundness of the type system through the compiler implementation. We are currently developing formal proofs. We first introduce the syntax, dynamic semantics, and static semantics of F_{pwo}. After that, we describe the typing with effect rows, including the closed prefix relation. Effect typing plays an important role in this study, since open floating depends heavily on it.

3.1. Syntax

The syntax is defined in Figure 1.

Expressions Expressions e include values v, applications $e\,e$, type applications $e\,\sigma$, let-bindings let $x = e$ in e, prompt prompt $m\,h\,e$, yield yield $m\,v$, and restrict restrict ϵ e. Prompts and yields are *internal constructs* that only appear as an intermediate result of evaluation. We will give the definition of internal constructs in Definition 1 more precisely.

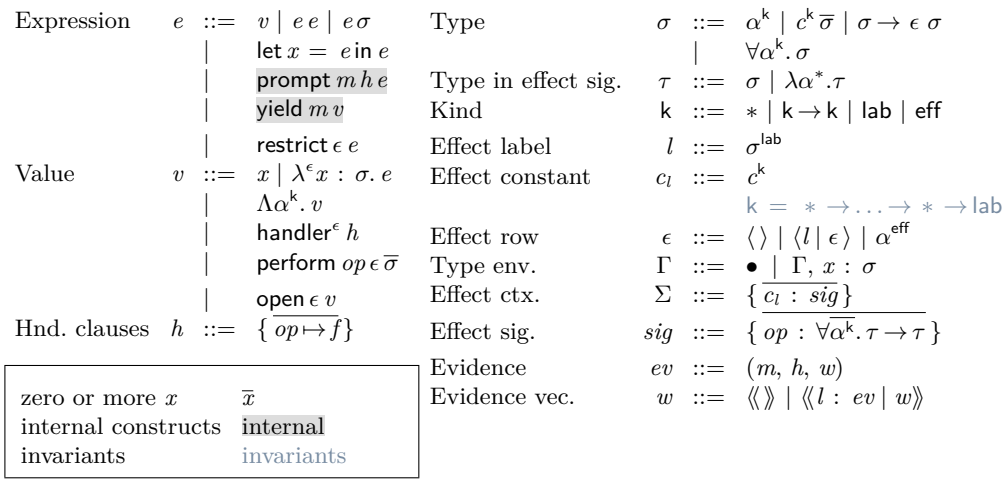

Figure 1. Syntax of System F_{pwo}

Values v include variables x, lambda abstractions $\lambda^{\epsilon} x : \sigma.e$, type abstractions $\Lambda\alpha^{\mathsf{k}}.v$, effect handlers $\mathsf{handler}^{\epsilon}\ h$, operation calls $\mathsf{perform}\ op\,\epsilon\,\overline{\sigma}$, and open $\mathsf{open}\ \epsilon\ v$.

Handlers clauses h consist of a sequence of pairs of an operation name op and a function value f. The meta-variable f is syntactically a value, but we use it specifically as a function, which takes (type) arguments. The type system maintains the intention.

Types Types σ include type variables α^{k} of kind k, type application for type constructors $c^{\mathsf{k}}\,\overline{\sigma}$ (where c^{k} is applied to arguments $\overline{\sigma}$), function types $\sigma_1 \to \epsilon\ \sigma_2$ (indicating the body of the function can cause effect ϵ), and polymorphic types $\forall\alpha^{\mathsf{k}}.\sigma$.[1] By *extending* σ with type-level lambdas, we obtain a set of types τ that appear in effect signatures. Effect signatures are explained later in this section.

Kinds k include the regular kind $*$, functions $\mathsf{k}{\to}\mathsf{k}$, effect labels lab, and effect rows eff. Types of the function kind $\mathsf{k}{\to}\mathsf{k}$ are either an effect constant c_l, a row type constructor $\langle \_ \mid \_ \rangle$, or a type τ in an effect signature. Effect labels l are types of kind lab.

Effect constants c_l are of kind lab parameterized with zero or more types of regular kind. Effect labels are used to structure effect rows and evidence vectors, while effect constants are used for effect contexts and effect labels.

Effect rows ϵ include the empty effect $\langle\,\rangle$, extension with effect label $\langle\, l \mid \epsilon\,\rangle$, and type variables α^{eff} of kind eff. We use the following abbreviation for rows: $\langle l_1, \ldots, l_n \mid \epsilon \rangle \doteq \langle l_1 \mid \ldots \langle\, l_n \mid \epsilon\,\rangle \ldots \rangle$ and $\langle l_1, \ldots, l_n \rangle \doteq \langle l_1, \ldots, l_n \mid \langle\,\rangle\,\rangle$. We use μ to denote type variable of kind eff. The equivalence of effect rows is defined in Figure 2. We can ignore the order of occurrences of two effect labels in a row

[1] Kind annotations directly relate their types to the specific symbols associated with them, such as μ and c_l. This allows us to use these symbols just as aliases.

$$\frac{}{\epsilon \cong \epsilon}\ [\text{EQ-REFL}] \qquad \frac{\epsilon_1 \cong \epsilon_2 \quad \epsilon_2 \cong \epsilon_3}{\epsilon_1 \cong \epsilon_3}\ [\text{EQ-TRANS}] \qquad \frac{\epsilon_1 \cong \epsilon_2}{\langle l \mid \epsilon_1 \rangle \cong \langle l \mid \epsilon_2 \rangle}\ [\text{EQ-HEAD}]$$

$$\frac{l_1 \neq l_2}{\langle l_1 \mid \langle l_2 \mid \epsilon \rangle \rangle \cong \langle l_2 \mid \langle l_1 \mid \epsilon \rangle \rangle}\ [\text{EQ-SWAP}] \qquad \frac{c_{l1} \neq c_{l2}}{c_{l1}\,\overline{\sigma}_1 \not\equiv c_{l2}\,\overline{\sigma}_2}\ [\text{UNEQ-LAB}]$$

Figure 2. Effect Row Type Equivalence

if two labels consist of different effect constants. See Section 3.4 for details of effect rows.

An effect context Σ is a sequence of pairs of an effect constant and an effect signature. It maintains the relation between the name of an effect and the type of its operations. We assume Σ is given externally in this calculus, while in practical language one may define Σ by top-level definitions.

Effect signatures *sig* are a sequence of a pair of an operation name and its type. The type τ in an effect signature takes zero or more type arguments of regular kind. The type arguments will be passed if the effect is parameterized. We will show an example with type rule [PERFORM] in Section 3.3.

Evidence Vectors Evidence vectors w include the empty vector $\langle\!\langle\,\rangle\!\rangle$ and extension $\langle\!\langle\, l : ev \mid w \,\rangle\!\rangle$ with a pair of an effect label l and an evidence ev. Evidences ev are a triple (m, h, w) of a marker m, a handler h, and an evidence vector w where h is defined. The evidence vector in the triple is key to general use of effect handlers, but the discussion is out of the scope of this paper. See [20] for details.

3.2. Dynamic Semantics

The dynamic semantics is defined in Figure 3. The semantics consists of three rules: stepping $\longmapsto$, multi-stepping $\longmapsto^*$, and reduction $\longrightarrow$.

Evaluation Steps The rules ($*stepwR$) and ($*stepwT$) defines multi-stepping as the reflexive transitive closure of stepping. The rules (*step*) and (*stepw*) reduce a redex without and with an evidence vector w, respectively. In these rules, the evaluation context of the redex must be F, not E. F excludes prompt frames and restrict frames.

The (*promptw*) rule extends the evidence vector. Conversely, the (*restrictw*) rule shortens the evidence vector using the *select* meta-function so that the shape of the new evidence vector fits the effect row ϵ' of the restrict frame.

Reduction Rules The (*app*), (*let*) and (*tapp*) rules are standard. The (*handler*) rule reduces a handler application by calling the passed function f under prompt with fresh marker m. The marker acts as a control delimiter [3]. The (*promptv*) rule removes the prompt frame if the handled expression is a value.

Operation call is divided into two rules: (*perform*) and (*prompt*). The (*perform*) rule prepares the marker m and handler clause f using the evidence vector. In the right-hand side of the (*prompt*), the handler clause is applied to (type) arguments and wrapped by a lambda to take a resumption. The (*prompt*) rule

Evaluation Contexts:

$$\begin{array}{lll} \mathsf{E} & ::= & \Box \mid \mathsf{E}\, e \mid v\mathsf{E} \mid \mathsf{E}\,\sigma \mid \mathsf{let}\, x = \mathsf{E}\, \mathsf{in}\, e \mid \mathsf{prompt}\, m\, h\, \mathsf{E} \mid \mathsf{restrict}\, \epsilon\, \mathsf{E} \\ \mathsf{F} & ::= & \Box \mid \mathsf{F}\, e \mid v\mathsf{F} \mid \mathsf{F}\,\sigma \mid \mathsf{let}\, x = \mathsf{F}\, \mathsf{in}\, e \end{array}$$

Evaluation Steps:

$$\frac{e \longrightarrow e'}{w \vdash \mathsf{F}[e] \longmapsto \mathsf{F}[e']}\ (step) \qquad \frac{w \vdash e \longrightarrow e'}{w \vdash \mathsf{F}[e] \longmapsto \mathsf{F}[e']}\ (stepw) \qquad \frac{}{w \vdash e \longmapsto^* e}\ (*stepwR)$$

$$\frac{\langle\!\langle\, l : (m, h, w) \mid w \,\rangle\!\rangle \vdash e \longmapsto e'}{w \vdash \mathsf{F}[\mathsf{prompt}\, m\, h\, e] \longmapsto \mathsf{F}[\mathsf{prompt}\, m\, h\, e']}\ (promptw)$$

$$\frac{\begin{array}{l}\mathsf{restrict}\, \epsilon'\ e\ :\ \sigma \mid \epsilon \quad \epsilon' \leqslant \epsilon \quad \vdash w\ :\ \epsilon \\ select\, \epsilon'\, w \vdash e \longmapsto e'\end{array}}{w \vdash \mathsf{F}[\mathsf{restrict}\, \epsilon'\ e] \longmapsto \mathsf{F}[\mathsf{restrict}\, \epsilon'\ e']}\ (restrictw) \qquad \frac{w \vdash e \longmapsto^* e' \quad w \vdash e' \longmapsto e''}{w \vdash e \longmapsto^* e''}\ (*stepwT)$$

Evidence Vector Operations:

$$\begin{array}{lll} select\, \langle\,\rangle\, w & = & \langle\!\langle\,\rangle\!\rangle \\ select\, \langle\, l \mid \epsilon\,\rangle\, w & = & \langle\!\langle\, l\ :\ w.l \mid select\, \epsilon\, (w - l)\,\rangle\!\rangle \end{array} \qquad \begin{array}{lll} \langle\!\langle\, l\ :\ ev \mid w\,\rangle\!\rangle.l & = & ev \\ \langle\!\langle\, l_1\ :\ ev \mid w\,\rangle\!\rangle.l_2 & = & w.l_2 \quad \text{if } l_1 \neq l_2 \end{array}$$

$$\begin{array}{lll} \langle\!\langle\, l\ :\ ev \mid w\,\rangle\!\rangle - l & = & w \\ \langle\!\langle\, l_1\ :\ ev \mid w\,\rangle\!\rangle - l_2 & = & w - l_2 \quad \text{if } l_1 \neq l_2 \end{array}$$

Reduction Rules:

$$\begin{array}{llll} (app) & (\lambda^{\epsilon} x : \sigma.\, e)\, v & \longrightarrow & e[x{:=}v] \\ (let) & \mathsf{let}\, x = v\, \mathsf{in}\, e & \longrightarrow & e[x{:=}v] \\ (tapp) & (\Lambda\alpha^{\mathsf{k}}.\, v)\, \sigma & \longrightarrow & v[\alpha{:=}\sigma] \\ (handler) & \mathsf{handler}^{\epsilon}\, h f & \longrightarrow & \mathsf{prompt}\, m\, h\, (f()) \quad \text{with unique } m \\ (promptv) & \mathsf{prompt}\, m\, h\, v & \longrightarrow & v \\ (perform) & w \vdash \mathsf{perform}\, op\ \epsilon_0\, \overline{\sigma}\, v & \longrightarrow & \mathsf{yield}\, m\, (\lambda^{\epsilon} k\colon (\sigma_2\, \overline{\sigma'})[\overline{\alpha}{:=}\overline{\sigma}] \to \epsilon\, \sigma.\, f\overline{\sigma}\, v\, k) \\ & & \text{with} & (m, h, \_) = w.l \wedge (op \mapsto f) \in h \quad l = c_l\, \overline{\sigma'} \\ & & & (op : \forall \overline{\alpha}.\, \sigma_{in} \to \sigma_{out}) \in \Sigma(c_l) \wedge \bullet \vdash h\ :\ \sigma \mid c_l\, \overline{\sigma'} \mid \epsilon \\ (prompt) & \mathsf{prompt}\, m\, h\, \mathsf{E}[\mathsf{yield}\, m\, f] & \longrightarrow & f(\lambda^{\epsilon} x\colon \sigma_2.\, \mathsf{prompt}\, m\, h\, \mathsf{E}[x]) \\ & & \text{with} & \bullet \vdash_{\mathsf{val}} f\ :\ (\sigma_2 \to \epsilon\, \sigma) \to \epsilon\, \sigma \\ (open) & (\mathsf{open}\, \epsilon'\, f)\, v & \longrightarrow & \mathsf{restrict}\, \epsilon\, (f\, v) \\ & & \text{with} & \epsilon = effectof(f) \quad \bullet \vdash f\ :\ \sigma_1 \to \epsilon\, \sigma_2 \\ (restrictv) & \mathsf{restrict}\, \epsilon\, v & \longrightarrow & v \end{array}$$

Effect Annotation Extractor:

$$\begin{array}{lll} effectof(\lambda^{\epsilon}\ x\ :\ \sigma.e) & = & \epsilon \\ effectof(\mathsf{handler}^{\epsilon}\ h) & = & \epsilon \end{array} \qquad effectof(\mathsf{perform}\, \epsilon\, \overline{\sigma}\, op) = \epsilon$$

Figure 3. Dynamic Semantics of System F_{pwo}

captures the resumption $\lambda^{\epsilon} x\colon \sigma_2.\, \mathsf{prompt}\, m\, h\, \mathsf{E}[x]$ by finding the marker m and applies the operation clause to it, which is instantiated by the (*perform*) rule.

The (*open*) rule generates a restrict frame using the effect annotation of the function value. Here, *effectof* meta-function is used to extract the effect type from the function value, which is either a lambda abstraction, an operation call,

or a handler. The effect row of an open expression is used for type checking. The (*restrictv*) rule removes the restrict frame if the sub-expression is a value.

3.3. Static Semantics

The static semantics is mutually defined with three relations $\vdash$, $\vdash_{\mathsf{val}}$, and $\vdash_{\mathsf{ops}}$ in Figure 4.

- $\Gamma \vdash e : \sigma \mid \epsilon$ means expression e is typed σ under type environment Γ and contextual effect ϵ, i.e., the type of the evidence vector provided for evaluation of e.
- $\Gamma \vdash_{\mathsf{val}} v : \sigma$ means value v is typed σ under type environment Γ. Note that if value v can be typed with $\vdash_{\mathsf{val}}$ relation, then it can be typed with *any* effect type ϵ with $\vdash$ relation, according to type rule [VAL].
- $\Gamma \vdash_{\mathsf{ops}} h : \sigma \mid l \mid \epsilon$ means that the sequence of operation clauses h has return type σ and handles effect operation of label l under effect type ϵ.

We also use well-formedness relation $\vdash_{\mathsf{wf}}$ and definitional equality of types $\vdash_{\mathsf{eq}}$ defined in Appendix.

Let us now look at the typing rules (Figure 4). These rules are syntax directed in the sense that the syntax of the expressions determines the applicable type rule. The [VAL] rule types values as expressions with any effect type ϵ. The [VAR] rule is usual. The [ABS] rule type checks the body e with the effect annotation ϵ of the lambda abstraction. The [APP] is standard except for the effect type: the operator (e_1) and operand (e_2) need to be typed under the contextual effect of entire expression $(e_1\ e_2)$. Furthermore, the effect type of the body of the operator also needs to agree with the one of the entire expression. The restriction guarantees that the evidence vector is passed correctly to sub-expressions. This may seem too restrictive, but the open construct liberates the restriction. The [BIND] rule is similar to the [APP] rule; both sub-expressions are required to be typed under the contextual effect ϵ. The [TABS] and [TAPP] rules are standard except for bound type variables, which cannot have kind lab.

The [PERFORM] rule determines the type of the operation call referring to the effect context Σ and the effect row $\langle cl\ \overline{\sigma'} \mid \epsilon \rangle$. The signature of the operation is found in the effect context and the type arguments $\overline{\sigma}$ are substituted for the type variables in the argument type τ_1 and the result type τ_2. Furthermore, the type arguments $\overline{\sigma'}$ from the effect row are applied to $\tau_1[\overline{\alpha}{:=}\overline{\sigma}]$ and $\tau_2[\overline{\alpha}{:=}\overline{\sigma}]$.

As an example, assuming $\Sigma = \{\, exn : \{\, throw : \forall\alpha.string \to \alpha \,\}\}$, we can write $\lambda^{\langle exn \rangle}\, x : string.\, 1 + \mathsf{perform}\ throw\, \langle exn \rangle\ int\, x$ as a well-typed function. The operation call in the body is typed with an instance of the [PERFORM] rule as follows.

$$\frac{\begin{array}{c} throw : \forall\beta.string \to \beta \in \Sigma(exn) \\ \vdash_{\mathsf{eq}} string[\beta{:=}int] \equiv string \qquad \vdash_{\mathsf{eq}} \beta[\beta{:=}int] \equiv int \end{array}}{x : string \vdash_{\mathsf{val}} \mathsf{perform}\ throw\, \langle exn \rangle\ int : string \to \langle exn \rangle\ int}$$

In this case, $c_l = exn$ and $\overline{\sigma'}$ is an empty sequence of types. If we replace *throw* with *polythrow string*, $c_l = polyexn$ and $\overline{\sigma'}$ is a singleton sequence *string*.

$$\boxed{\Gamma \vdash e : \sigma \mid \epsilon \qquad \Gamma \vdash_{\mathsf{val}} v : \sigma \qquad \Gamma \vdash_{\mathsf{ops}} h : \sigma \mid l \mid \epsilon}$$

$$\frac{\Gamma \vdash_{\mathsf{val}} v : \sigma}{\Gamma \vdash v : \sigma \mid \epsilon}\ [\textsc{val}] \qquad \frac{\Gamma,\ x : \sigma_1 \vdash e : \sigma_2 \mid \epsilon}{\Gamma \vdash_{\mathsf{val}} \lambda^{\epsilon} x : \sigma_1.\, e : \sigma_1 \to \epsilon\, \sigma_2}\ [\textsc{abs}]$$

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash_{\mathsf{val}} x : \sigma}\ [\textsc{var}] \qquad \frac{\Gamma \vdash e_1 : \sigma_1 \to \epsilon\, \sigma \mid \epsilon \quad \Gamma \vdash e_2 : \sigma_1 \mid \epsilon}{\Gamma \vdash e_1\, e_2 : \sigma \mid \epsilon}\ [\textsc{app}]$$

$$\frac{\Gamma \vdash e_1 : \sigma_1 \mid \epsilon \quad \Gamma,\, x : \sigma_1 \vdash e_2 : \sigma \mid \epsilon}{\Gamma \vdash \mathsf{let}\, x = e_1\, \mathsf{in}\, e_2 : \sigma \mid \epsilon}\ [\textsc{bind}]$$

$$\frac{\Gamma \vdash v : \sigma \quad \mathsf{k} \neq \mathsf{lab} \quad \alpha \notin \mathsf{ftv}(\Gamma)}{\Gamma \vdash_{\mathsf{val}} \Lambda\alpha^{\mathsf{k}}.\ v : \forall\alpha^{\mathsf{k}}.\sigma}\ [\textsc{tabs}] \qquad \frac{\Gamma \vdash e : \forall\alpha^{\mathsf{k}}.\sigma_1 \mid \epsilon \quad \vdash_{\mathsf{wf}} \sigma : \mathsf{k}}{\Gamma \vdash e\,\sigma : \sigma_1[\alpha{:=}\sigma] \mid \epsilon}\ [\textsc{tapp}]$$

$$\frac{\begin{array}{l} op : \forall\overline{\alpha}.\tau_1 \to \tau_2 \in \Sigma(c_l) \\ \vdash_{\mathsf{eq}} (\tau_1[\overline{\alpha}{:=}\overline{\sigma}]\ \overline{\sigma'}) \equiv \sigma_1 : * \qquad \vdash_{\mathsf{eq}} (\tau_2[\overline{\alpha}{:=}\overline{\sigma}]\ \overline{\sigma'}) \equiv \sigma_2 : * \end{array}}{\Gamma \vdash_{\mathsf{val}} \mathsf{perform}\, op\, \langle c_l\ \overline{\sigma'} \mid \epsilon \rangle\, \overline{\sigma} : \sigma_1 \to \langle c_l\, \overline{\sigma'} \mid \epsilon \rangle\, \sigma_2}\ [\textsc{perform}]$$

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

$$\frac{\begin{array}{l} \{op_i : \forall\overline{\alpha}_i.\tau_i^{in} \to \tau_i^{out}\}_{i=1}^{n} = \Sigma(c_l) \quad \overline{\alpha}_i \not\cap \mathsf{ftv}(\epsilon, \overline{\sigma}, \overline{\sigma'}) \\ \vdash_{\mathsf{eq}} \tau_i^{in}\, \overline{\sigma'} \equiv \sigma_i^{in} : * \quad \vdash_{\mathsf{eq}} \tau_i^{out}\, \overline{\sigma'} \equiv \sigma_i^{out} : * \\ \Gamma \vdash_{\mathsf{val}} f_i : \forall\overline{\alpha}_i.\sigma_i^{in} \to \epsilon\, ((\sigma_i^{out} \to \epsilon\, \sigma) \to \epsilon\, \sigma) \end{array}}{\Gamma \vdash_{\mathsf{ops}} \{op_1 \to f_1, \ldots, op_n \to f_n\} : \sigma \mid c_l\, \overline{\sigma'} \mid \epsilon}\ [\textsc{ops}]$$

$$\frac{\Gamma \vdash_{\mathsf{val}} f : (\sigma \to \epsilon'\, \sigma') \to \epsilon'\, \sigma'}{\Gamma \vdash \mathsf{yield}\, m f : \sigma \mid \epsilon}\ [\textsc{yield}] \qquad \frac{\Gamma \vdash_{\mathsf{ops}} h : \sigma \mid l \mid \epsilon \quad \Gamma \vdash e : \sigma \mid \langle l \mid \epsilon \rangle}{\Gamma \vdash \mathsf{prompt}\, m\, h\, e : \sigma \mid \epsilon}\ [\textsc{prompt}]$$

$$\frac{\Gamma \vdash_{\mathsf{val}} v : \sigma_1 \to \epsilon'\, \sigma_2 \quad \epsilon' \leqslant \epsilon}{\Gamma \vdash_{\mathsf{val}} \mathsf{open}\, \epsilon\, v : \sigma_1 \to \epsilon\, \sigma_2}\ [\textsc{open}] \qquad \frac{\Gamma \vdash e : \sigma \mid \epsilon' \quad \epsilon' \leqslant \epsilon}{\Gamma \vdash \mathsf{restrict}\, \epsilon'\, e : \sigma \mid \epsilon}\ [\textsc{restrict}]$$

Closed Prefix: $\langle l_1, \ldots, l_n \rangle \leqslant \langle l_1, \ldots, l_n \mid \epsilon \rangle \quad (n \geq 0)$

Figure 4. Typing Rules of System F_{pwo}

$$\frac{\begin{array}{c} \mathit{polythrow} \ : \ \forall\beta.(\lambda\alpha.\alpha) \to (\lambda\alpha.\beta) \ \in \Sigma(\mathit{polyexn}) \\ \vdash_{\mathsf{eq}} \ (\lambda\alpha.\alpha)[\beta{:=}\mathit{int}]\ \mathit{string} \equiv \mathit{string} \quad \vdash_{\mathsf{eq}} \ (\lambda\alpha.\beta)[\beta{:=}\mathit{int}]\ \mathit{string} \equiv \mathit{int} \end{array}}{x \ : \ \mathit{string} \vdash_{\mathsf{val}} \ \mathsf{perform}\ \mathit{polythrow}\,\langle\,\rangle\ \mathit{int} \ : \ \mathit{string} \to \langle \mathit{polyexn}\ \mathit{string}\rangle\ \mathit{int}}$$

The [HANDLER] rule is defined for handler expressions. A handler takes a computation of type $(() \to \langle l \mid \epsilon \rangle\, \sigma)$ and handles the effect l. Hence, the effect row of the entire function type is ϵ, not $\langle l \mid \epsilon \rangle$.

The [OPS] rule determines the effect labels $c_l\,\overline{\sigma'}$, which indicate the handled effect. Each operation clause f_i takes an operation argument of type σ_i^{in} and a resumption of type $\sigma_i^{out} \to \epsilon\, \sigma$. The result type (σ) of the handler is the result type of all resumptions and operation clauses, because handlers in System F_{pwo} are *deep* ones. The condition $\overline{\alpha}_i \ \sharp \mathsf{ftv}(\epsilon, \sigma, \overline{\sigma'})$ avoids unexpected binding in the type of f_i. The arguments of effect constants $\overline{\sigma'}$ are derived from the typing of each operation clause. By combining them with the effect constant c_l, we derive the effect label $c_l\,\overline{\sigma'}$.

The [YIELD] rule requires careful reading. Recall that f is a wrapped handler clause that will be applied to a resumption. The result type of f, which is the result type of the handler clause, must agree with the result type of the resumption. Therefore the two σ' need to agree. The two σ indicate that the input type of f, which is the type of the "result" of the operation call, must agree with the type of the yield expression. Note that the effect type ϵ of yield is not related to the effect ϵ' of the operation clause, as the evaluation contexts of yield and prompt (in which f will be evaluated) are different in general.

The [PROMPT] rule extends the contextual effect ϵ with the effect label l to type check sub-expression e. The result type of e and that of handler clauses h need to agree, and it becomes the type of the entire prompt expression.

The [OPEN] rule opens (make bigger) the effect type to the given effect ϵ. The original effect type ϵ' must be a *closed prefix* of the resulting effect type ϵ. An effect row ϵ_1 is a closed prefix of ϵ_2 if and only if ϵ_1 consists of labels in a prefix of ϵ_2 and ends in $\langle\,\rangle$. We discuss the definition of the closed prefix relation in the next section. The [RESTRICT] rule is similar to [OPEN] and allows expression e to be typed under a closed prefix ϵ'.

3.4. Effect Rows and Closed Prefix Relation

In this section, we discuss how the type system exploits effect rows to perform type checking against effect handlers and discuss requirement for open and restrict to entail type safety.

Recall the typing rule [PERFORM] for operation calls. The conclusion of the rule has an effect row $\langle c_l\,\overline{\sigma'} \mid \epsilon \rangle$, which tells us that evaluation of the operation call needs to access a handler of effect label $c_l\,\overline{\sigma'}$. The accessibility of the required handler is guaranteed by the row equivalence rules defined in Figure 2. For instance, among the two examples below, the first one is correctly rejected due to the inapplicability of [EQ-SWAP], and the second one is accepted as desired.

$\mathsf{handler}\ h^{\mathit{polyexn}\ \mathit{int}}\,(\mathsf{handler}\ h^{\mathit{polyexn}\ \mathit{string}}\,(\lambda\_.\ \mathit{throw}\ 1))$
$\mathsf{handler}\ h^{\mathit{polyexn}\ \mathit{int}}\,(\mathsf{handler}\ h^{\mathit{polyexn}\ \mathit{string}}\,(\lambda\_.\ \mathit{throw}\,\text{"}\mathit{hello}\text{"}))$

We design the closed prefix relation so that restrict does not increase handlers accessible from the sub-expressions.

This is stated as the following property.

Proposition 1.
If $\epsilon.l$ is defined and $\epsilon \leqslant \epsilon'$, then $\epsilon'.l$ is also defined and $\epsilon.l = \epsilon'.l$.

It is obvious that the closed prefix relation satisfies this property, but what about other candidates? Initially, we considered an *open prefix* relation defined as follows.[2]

$$\langle l_1, \ldots, l_k \mid \mu \rangle \leqslant_? \langle l_1, \ldots, l_k, l_{k+1}, \ldots, l_n \mid \mu \rangle$$

Here, μ is a type variable of kind eff. The open prefix relation leads to loss of type safety, as shown by the following example.

$$(\Lambda\mu.\lambda^{\langle polyexn\, int|\mu \rangle} f : ()\to\mu\,().\, \mathsf{restrict}\,\mu\,(f()))$$
$$\langle polyexn\, string \rangle\, \lambda^{\langle polyexn\, string \rangle} \_.polythrow"blame!"$$

Here, the example would be accepted with an open prefix relation because $\mu \leqslant_? \langle polyexn\, int \mid \mu \rangle$ holds. However, the effect annotation $\langle polyexn\, int \mid \mu \rangle$ indicates that the innermost *polyexn* handler expects *polyexn int*, while *polythrow* raises a string value, causing a type mismatch at runtime! Fortunately, using the closed prefix relation will reject the example and preserve soundness.

4. Open Floating

In this section, we present open floating, a transformation algorithm defined on System F_{pwo}. We first provide the design of open floating intuitively (Section 4.1). We next describe how to implement the idea with defining auxiliary definitions (Section 4.2). We then detail the definition of open floating (Section 4.3). Lastly, we show an example of the transformation (Section 4.4).

If the reader is interested in the overall idea of the algorithm instead of the details, the reader may skip the formal sections (Sections 4.2 and 4.3).

4.1. Design of Algorithm

Open floating is designed under the strategy *assign minimum effect for each sub-expression*. The first half of this section motivates the strategy and the latter half explains a post-process of open floating. In order to reduce open constructs, we first float up open constructs and then collapse redundant ones. We call the first phase open floating, and the second phase the post-process.

The aim of open floating is to fuse multiple open constructs into one. We saw an example in Section 1 that shows open floating fuses two opens into one restrict. There is one constraint that must be satisfied by open floating: the transformed program is well-typed. The simplicity of the constraint is due to the formalization of open and restrict we saw in Section 3. Now the question is:

[2] In the Koka-related literature [9, 19, 20], an effect row is called closed when it ends with $\langle\rangle$, and open when its tail ends with a type variable. We adopt the naming convention of the above literature.

how can we reduce open constructs while satisfying this constraint? Our answer is to *assign minimum effect for each sub-expression*.

Before open floating, open constructs are used to assign the maximum effect type to each expression. Effect adjustments are never inserted in the middle of ASTs. Instead, every expression is assigned a big effect, and if a function call requires effect adjustment, an open construct is inserted to the function call. If adjacent sub-expressions perform the same effect adjustment, we can avoid adjustments at function calls by making the adjustment surround those sub-expressions, which makes the effect types of sub-expressions *smaller*. Based on this idea, in open floating, we assign the minimum effect type to each sub-expression.

Unfortunately, the above strategy is naive in that it may insert redundant restrict in a corner case. For instance, the following transformation increases the number of open/restrict constructs from three to four. The problem is the restrict in line 3 of the transformed program. We eliminate it by the post-process.

$$f\ :\ ()\to\langle l_1\rangle\,(),\quad g\ :\ ()\to\langle l_2\rangle\,(),\quad h\ :\ ()\to\langle l_3\rangle\,()$$

$$\begin{array}{l}\lambda^{\langle l_1,l_2,l_3\rangle}\_.\\ \mathsf{let}\ x\ =\ \mathsf{open}\,\langle l_1,l_2,l_3\rangle\,f()\ \mathsf{in}\\ \mathsf{let}\ y\ =\ \mathsf{open}\,\langle l_1,l_2,l_3\rangle\,g()\ \mathsf{in}\\ \mathsf{open}\,\langle l_1,l_2,l_3\rangle\,h()\end{array}\quad\rightsquigarrow\quad\begin{array}{l}\lambda^{\langle l_1,l_2,l_3\rangle}\_.\\ \mathsf{let}\ x\ =\ \mathsf{restrict}\,\langle l_1\rangle\,f()\ \mathsf{in}\\ \mathsf{restrict}\,\langle l_2,l_3\rangle\,(\\ \quad\mathsf{let}\ y\ =\ \mathsf{restrict}\,\langle l_2\rangle\,g()\ \mathsf{in}\\ \quad\mathsf{restrict}\,\langle l_3\rangle\,h())\end{array}$$

It is not hard to see the validity of this removal: the necessary effect adjustment is performed by the restricts on lines 4 and 5. In general, a restrict is redundant if removing it does not force the sub-expressions to use additional open or restrict. We can detect such restricts by checking the sub-expressions of restrict-expressions.

4.2. Organization of Definition

The algorithm consists of three kinds of rules, taking care of expressions, values, and operation clauses.

- $\Gamma\vdash\ e\mid\epsilon\rightsquigarrow\ e'\ :\ \sigma\mid\varphi$
- $\Gamma\vdash_{\mathsf{val}}\ v\rightsquigarrow\ v'\ :\ \sigma$
- $\Gamma\vdash_{\mathsf{ops}}\ h\rightsquigarrow\ h'\ :\ \sigma\mid l\mid\epsilon$

In this study, proper programs are well-typed and *internal-safe* expressions.

Definition 1. (*Internal-free expression, Internal-safe expression*)
An expression is *internal-safe* if it is (1) internal-free (contains no prompt or yield expressions) or (2) reduced from an internal-safe expression.

Open floating takes a proper program fragment and outputs proper program fragment with an *effect requirement* φ (defined in Section 4.3). To define open floating, we represent the "minimum effect" of an expression as an effect requirement. It is an extension of effect row types with the bottom (least) value $\varnothing$. Open floating calculates the effect type of the transformed expression in a

Effect Requirement: $\varphi ::= \varnothing \mid \epsilon$, with an order defined as:

$$\frac{}{\varnothing \sqsubseteq \varphi}\ [\textsc{bottom}] \qquad \frac{}{\varphi \sqsubseteq \varphi}\ [\textsc{refl}] \qquad \frac{\epsilon_1 \leqslant \epsilon_2}{\epsilon_1 \sqsubseteq \epsilon_2}\ [\textsc{prefix}]$$

$$\begin{array}{l} \mathit{sup} \ : \ \mathit{Requirement} \longrightarrow 2^{\mathit{Requirement}} \longrightarrow \mathit{Requirement} \\ \mathit{sup}_{\epsilon}\, \{\varphi_i\}_i \ = \ \mathsf{min}\, \{\, \varphi \mid \varphi_i \sqsubseteq \varphi \sqsubseteq \epsilon \,\} \end{array}$$

Smart open and restrict:

$$\begin{array}{lll} \mathit{open}'\ \varphi\ \varphi'\ e \ = \ e & (\mathit{if} \varphi \cong \varphi') \\ \mathit{open}'\ \varnothing\ \varphi'\ e \ = \ e \\ \mathit{open}'\ \epsilon\ \epsilon'\ e \ = \\ \quad \mathsf{let}\ x = e\ \mathsf{in}\ \mathsf{open}\ \epsilon'\ x & (\mathit{if} \epsilon \leqslant \epsilon') \\ \mathit{open}'\ \varphi\ \varphi'\ e \ = \ \mathit{undefined} & (\mathit{otherwise}) \end{array} \qquad \begin{array}{lll} \mathit{restrict}'\ \varphi\ \varphi'\ e \ = \ e & (\mathit{if} \varphi \cong \varphi') \\ \mathit{restrict}'\ \varnothing\ \varphi'\ e \ = \ e \\ \mathit{restrict}'\ \epsilon\ \epsilon'\ e \ = \ \mathsf{restrict}\ \epsilon\ e & (\mathit{if} \epsilon \leqslant \epsilon') \\ \mathit{restrict}'\ \varphi\ \varphi'\ e \ = \ \mathit{undefined} & (\mathit{otherwise}) \end{array}$$

Figure 5. Effect Requirement and Auxiliary Definitions

bottom-up way with effect requirements. If the effect requirement of the expression is an effect row type, then the expression is assigned that effect type as the output of the algorithm. If the effect requirement of the expression is the bottom value, the expression is assigned an appropriate effect type determined later by the surrounding context.

When we assign effect types to expressions, we need to maintain well-typedness of expressions. The auxiliary functions defined in Figure 5 help us do so. For instance, in the case of a function application, the effect types of the operation, the operand, and the body of operation must be the same. To ensure this, we first calculate the effect requirement of the whole function application using the *sup* function, and then make the effect types consistent by inserting open constructs using open' and $\mathit{restrict}'$. Note that open' may produce let bindings to ensure that the function is syntactically a value, which is required by the syntax rule of open.

4.3. The Definition

We define the open floating algorithm in Figure 6 following the idea described in the previous section. The light propositions are invariants, not side conditions; we write them to clarify the intention of the algorithm design. The transformation is syntax directed, except for applications $(e\ e)$, in which case both [Open-App] and [App] rules may apply.

Rule [Var] simply returns the variable with its type. Rule [Lam] recursively applies the algorithm to the body e. We do not simply return $\lambda^{\varphi} x : \sigma_1 . e'$ but return $\lambda^{\epsilon}\ \mathit{restrict}'\ \varphi\ \epsilon\ e'$ in order to preserve the type of the lambda abstraction. Rule [Val] assigns a null requirement $\varnothing$ to value. Rule [App] treats three effect requirements: the requirements of the two sub-expressions (φ_1 and φ_2) and the effect type of the function body. The rule uses the supremum of these requirements for the result. Rule [Bind] also takes the supremum of the two effect rows. Rules [TAbs], [TApp], [Perform], [Handler], and [Ops] simply recursively call the algorithm and propagate the results.

$$\boxed{\Gamma \vdash e \mid \epsilon \rightsquigarrow e' : \sigma' \mid \varphi \qquad \Gamma \vdash_{\mathsf{val}} v \rightsquigarrow v' : \sigma \qquad \Gamma \vdash_{\mathsf{ops}} h \rightsquigarrow h' : \sigma \mid l \mid \epsilon}$$

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash_{\mathsf{val}} x \rightsquigarrow x : \sigma} \ [\textsc{Var}] \qquad \frac{\Gamma \vdash e \mid \epsilon \rightsquigarrow e' : \forall \alpha^k.\sigma_1 \mid \varphi}{\Gamma \vdash e\,[\sigma'] \mid \epsilon \rightsquigarrow e'\,[\sigma'] : \sigma[\alpha \coloneqq \sigma'] \mid \varphi} \ [\textsc{TApp}]$$

$$\frac{\Gamma \vdash_{\mathsf{val}} v \rightsquigarrow v' : \sigma}{\Gamma \vdash v \rightsquigarrow v' : \sigma \mid \varnothing} \ [\textsc{Val}] \qquad \frac{\Gamma, \alpha^k \vdash_{\mathsf{val}} v \rightsquigarrow v' : \sigma}{\Gamma \vdash_{\mathsf{val}} \Lambda\alpha^k.v \rightsquigarrow \Lambda\alpha^k.v' : \forall\alpha.\,\sigma} \ [\textsc{TAbs}]$$

$$\frac{\Gamma, x : \sigma_1 \vdash e \mid \epsilon \rightsquigarrow e' : \sigma_2 \mid \varphi \quad \varphi \sqsubseteq \epsilon}{\Gamma \vdash_{\mathsf{val}} \lambda^{\epsilon} x : \sigma_1.\, e \ \rightsquigarrow \lambda^{\epsilon} x : \sigma_1.\, \mathit{restrict}'\, \varphi\, \epsilon\, e' : \sigma_1 \to \epsilon\, \sigma_2} \ [\textsc{Lam}]$$

$$\frac{\begin{array}{l}\Gamma \vdash e_1 \mid \epsilon \rightsquigarrow e_1' : \sigma_2 \to \epsilon_{\mathsf{body}}\, \sigma \mid \varphi_1 \quad \Gamma \vdash e_2 \mid \epsilon \rightsquigarrow e_2 : \sigma_2 \mid \varphi_2 \\ \epsilon_s = \mathit{sup}_{\epsilon}\, \{\epsilon_{\mathsf{body}}, \varphi_1, \varphi_2\}\end{array}}{\Gamma \vdash e_1\, e_2 \mid \epsilon \rightsquigarrow (\mathit{open}'\, \epsilon_{\mathsf{body}}\, \epsilon_s\, (\mathit{restrict}'\, \varphi_1\, \epsilon_s\, (e_1'))\,)\, (\mathit{restrict}'\, \varphi_2\, \epsilon_s\, e_2') : \sigma \mid \epsilon_s} \ [\textsc{App}]$$

$$\frac{\Gamma \vdash e_1 \mid \epsilon \rightsquigarrow e_1' : \sigma_1 \mid \varphi_1 \quad \Gamma \vdash e_2 \mid \epsilon \rightsquigarrow e_2' : \sigma_2 \mid \varphi_2 \quad \varphi = \mathit{sup}_{\epsilon}\, \{\varphi_1, \varphi_2\}}{\Gamma \vdash \mathsf{let}\, x = e_1\, \mathsf{in}\, e_2 \mid \epsilon \rightsquigarrow \mathsf{let}\, x = \mathit{restrict}'\, \varphi_1\, \varphi\, e_1'\, \mathsf{in}\, \mathit{restrict}'\, \varphi_2\, \varphi\, e_2' : \sigma_2 \mid \varphi} \ [\textsc{Bind}]$$

$$\frac{\begin{array}{l}\mathit{op} : \forall\overline{\alpha}.\tau_1 \to \tau_2 \in \Sigma(c_l) \qquad \overline{\alpha} \not\cap \mathsf{ftv}(\Gamma) \\ \vdash_{\mathsf{eq}} (\tau_1[\overline{\alpha} \coloneqq \overline{\sigma}]\ \overline{\sigma'}) \equiv \sigma_1 : * \quad \vdash_{\mathsf{eq}} (\tau_2[\overline{\alpha} \coloneqq \overline{\sigma}]\ \overline{\sigma'}) \equiv \sigma_2 : *\end{array}}{\Gamma \vdash_{\mathsf{val}} \mathsf{perform}\ \mathit{op}\, \langle c_l\, \overline{\sigma'} \mid \epsilon \rangle\, \overline{\sigma} \rightsquigarrow \mathsf{perform}\ \mathit{op}\, \langle c_l\, \overline{\sigma'} \mid \epsilon \rangle\, \overline{\sigma} : \sigma_1 \to \langle c_l\, \overline{\sigma'} \mid \epsilon \rangle\, \sigma_2} \ [\textsc{Perform}]$$

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

$$\frac{\begin{array}{l}\{\mathit{op}_i : \forall\overline{\alpha}_i.\tau_i^{in} \to \tau_i^{out}\}_{i=1}^{n} = \Sigma(c_l) \quad \overline{\alpha}_i \not\cap \mathsf{ftv}(\epsilon, \overline{\sigma}, \overline{\sigma'}) \\ \vdash_{\mathsf{eq}} \tau_i^{in}\, \overline{\sigma'} \equiv \sigma_i^{in} : * \qquad \vdash_{\mathsf{eq}} \tau_i^{out}\, \overline{\sigma'} \equiv \sigma_i^{out} : * \\ \Gamma \vdash_{\mathsf{val}} f_i \rightsquigarrow f_i' : \forall\overline{\alpha_i}.\sigma_i^{in} \to \epsilon\, ((\sigma_i^{out} \to \epsilon\, \sigma) \to \epsilon\, \sigma)\end{array}}{\Gamma \vdash_{\mathsf{ops}} \{\mathit{op}_1 \to f_1, \ldots, \mathit{op}_n \to f_n\} \rightsquigarrow \{\mathit{op}_1 \to f_1', \ldots, \mathit{op}_n \to f_n'\} : \sigma \mid c_l\, \overline{\sigma'} \mid \epsilon} \ [\textsc{Ops}]$$

$$\frac{\Gamma \vdash v\, e \mid \epsilon \rightsquigarrow e' : \sigma \mid \varphi \quad \varphi \sqsubseteq \epsilon}{\Gamma \vdash (\mathsf{open}\, \epsilon\, v)\, e \mid \epsilon \rightsquigarrow e' : \sigma \mid \varphi} \ [\textsc{Open-App}]$$

$$\frac{\Gamma \vdash v \rightsquigarrow v' : \sigma_1' \to \epsilon'\, \sigma_2' \quad \epsilon' \leqslant \epsilon}{\Gamma \vdash_{\mathsf{val}} \mathsf{open}\, \epsilon\, v \rightsquigarrow \mathsf{open}\, \epsilon\, v' : \sigma_1 \to \epsilon\, \sigma_2} \ [\textsc{Open-Preserve}]$$

$$\frac{\Gamma \vdash e \mid \epsilon_1 \rightsquigarrow e' : \sigma \mid \varphi}{\Gamma \vdash \mathsf{restrict}\, \epsilon_1\, e \mid \epsilon \rightsquigarrow e' : \sigma \mid \varphi} \ [\textsc{Restrict}]$$

Figure 6. Open Floating Algorithm

Rule [Open-App] processes application of an opened function. It removes open and may assign a smaller effect to the expression. This rule conflicts with the [App] rule, hence we give priority to [Open-App]. Rule [Open-Preserve] recursively

calls the algorithm while keeping open. This rule is required for function arguments, for instance. Let us consider the following example.

$$\begin{array}{l} \mathit{safemap} \ : \ \mathit{list}\langle \mathit{int} \rangle \to \langle\,\rangle \, (\mathit{int} \to \langle \mathit{exn} \rangle \, \mathit{int}) \to \langle\,\rangle \, \mathit{list}\langle \mathit{int} \rangle \\ \mathit{safemap} \, [1,\, 2,\, 3,\, 4] \, (\mathsf{open} \, \langle \mathit{exn} \rangle \, \mathit{addone}) \end{array}$$

The *safemap* function expects a function argument of $\mathit{int} \to \langle \mathit{exn} \rangle \, \mathit{int}$ and handles the exception effect. If we want to pass a function of effect $\langle\,\rangle$ as the argument (*addone* in this case), we need to open it to make the entire program type check. The last rule [Restrict] ignores the existing restrict.

4.4. Example

Recall from Section 4.1 that open floating assigns the minimum effect to each sub-expression. Let us observe how this principle is implemented by the definition. Assume we have the following bindings in the type environment, where l_1, l_2, and l_3 are distinct effect labels.

$$\Gamma \ = \ f \ : \ \mathit{int} \to \langle l_1, l_2 \rangle \, \mathit{int}, \ g \ : \ \mathit{int} \to \langle l_2, l_3 \rangle \, \mathit{int}$$

Then, open floating performs the following transformation.

$$\begin{array}{l} \lambda^{\langle l_1, l_2, l_3 \rangle} x \ : \ \mathit{int}. \\ \quad \mathsf{let} \ y \ = \ (\mathsf{open} \, \langle l_1, l_2, l_3 \rangle \, f) \, x \, \mathsf{in} \\ \quad \mathsf{let} \ z \ = \ (\mathsf{open} \, \langle l_1, l_2, l_3 \rangle \, g) \, y \, \mathsf{in} \\ \quad (\mathsf{open} \, \langle l_1, l_2, l_3 \rangle \, g) \, z \end{array} \quad \rightsquigarrow \quad \begin{array}{l} \lambda^{\langle l_1, l_2, l_3 \rangle} x \ : \ \mathit{int}. \\ \quad \mathsf{let} \ y \ = \ \mathsf{restrict} \, \langle l_1, l_2 \rangle \, f x \, \mathsf{in} \\ \quad \mathsf{restrict} \, \langle l_2, l_3 \rangle \, (\\ \quad\quad \mathsf{let} \ z \ = \ g \, y \, \mathsf{in} \\ \quad\quad g \, z) \end{array}$$

First, rules such as [Abs] and [Bind] are applied to the lambda abstraction and let-expressions. The function application $(\mathsf{open} \, \langle l_1, l_2, l_3 \rangle \, g) \, z$ in the last line is processed by rule [Open-App]; it outputs expression $g \, z$ with effect requirement $\langle l_2, l_3 \rangle$. The let-expression in the second-last line is processed by rule [Bind]; it outputs expression $\mathsf{let} \, z \ = \ g \, y \, \mathsf{in} \, g \, z$ with effect requirement $\langle l_2, l_3 \rangle$. Notice that two opens in this let-expression are eliminated. This is because the effect type of the original program is $\langle l_1, l_2, l_3 \rangle$, while the transformed one has effect type $\langle l_2, l_3 \rangle$, which agrees with the effect of function g.

Focusing our attention on the top-most let-expression, we see that two restricts are inserted. The transformed definition of the let-expression is $f x$ with effect requirement $\langle l_1, l_2 \rangle$. The whole let-expression need to require the effect that "satisfies" the requirements of both sub-expressions. This invariant is maintained by the [Bind] rule, which calculates the overall requirement using the *sup* function and inserts restrict using the $\mathit{restrict}'$ function if needed. Here, both the definition and the body require different effects than the entire let-expression, hence restricts are inserted (the inner let-expression does not require restrict as there is no gap). Lastly, the [Abs] rule (which is the first rule applied to the program) outputs the overall result of the transformation. Thus, open floating floats up open constructs by determining the effect type of each sub-expression in a bottom-up way.

```
fun test-one() : ⟨exn,read₁,read₂⟩ int
  val x = square(1) + ... + square(1)  // call `square(1)` 20 times
  val y = square-ask₁()
  val z = square-ask₂()
  x + y + z

noinline fun square( i : int ) : ⟨exn, read₁⟩ int
  if True then i * i else throw("impossible: " ++ ask₁().show)
noinline fun square-ask₁() : ⟨read₁⟩ int
  ask₁() * ask₁()
noinline fun square-ask₂() : ⟨read₂⟩ int
  ask₂() * ask₂()
```

Open floating	Exec. time (sec.)	Open call (static)	Open call (dynamic)
Enabled	0.957	3	30,000
Disabled	2.388	22	220,000

Figure 7. Benchmarking open floating (the ideal case)

5. Evaluation

We implemented our open floating algorithm in the Koka compiler [11] and evaluated open floating with artificial programs. The results show that, in the best case, open floating makes programs about 2.5 times faster, while in some cases, it makes programs slower. In this section, we summarize the experiments and discuss what kind of programs are made faster by open floating.

5.1. Ideal Case

Figure 7 includes a fragment of a small benchmark with the execution times. The number of open calls is the sum of open and restrict expressions in the program. As we can see, open floating is very effective here and the enabling open floating improves performance by 2.5×. The execution times are averaged over 3 runs, on an Intel Core i5 at 3Ghz with 8GiB memory running macOS 11.6.3, with Koka v2.3.9 extended with open floating. In the program, we use three kinds of effects: `read`$_1$, `read`$_2$, and `exn`. The function `test-one` has the effect ⟨`read`$_1$`,read`$_2$`,exn`⟩, while the other functions use smaller effects. In the body of `test-one`, the Koka type inferencer inserts the following open calls around the `square`, `square-ask`$_1$, and `square-ask`$_2$ functions:

$$\begin{array}{l}\mathsf{let}\, x = \mathsf{open}(\mathit{square})(1) + \ldots + \mathsf{open}(\mathit{square})(1) \quad (\text{call}\, \mathsf{open}(\mathit{square})(1)\, 20\ \text{times}) \\ \mathsf{let}\, y = \mathsf{open}(\mathit{squareask}_1)() \\ \mathsf{let}\, z = \mathsf{open}(\mathit{squareask}_2)() \\ x + y + z\end{array}$$

This program contains 22 open calls initially and open floating reduces them to 3 restricts as follows:

$$\begin{array}{l}\mathsf{let}\, x = \mathsf{restrict}(\, \mathit{square}(1) + \ldots + \mathit{square}(1)\,) \quad (\text{call}\, \mathit{square}(1)\, 20\ \text{times}) \\ \mathsf{let}\, y = \mathsf{restrict}(\mathit{squareask}_1()) \\ \mathsf{let}\, z = \mathsf{restrict}(\mathit{squareask}_2()) \\ x + y + z\end{array}$$

Open floating	Exec. time (sec.)	Open call (static)	Open call (dynamic)
Enabled	0.500	3	30,000
Disabled	0.182	22	220,000

Figure 8. Benchmark of Failure Case

Here all individual open calls around each *square* invocation are floated up to a single restrict, leading to the improved performance.

5.2. Failure Case

However, making a small modification to the program can make open floating less effective. Consider a modified version of the `square` function in Figure 7.

```
noinline fun square'( i : int ) : ⟨exn⟩ int
  if True then i * i else throw("impossible: ")
```

We have removed `++ ask1().show` and changed the effect from ⟨`ask1, exn`⟩ to ⟨`exn`⟩. Just as before, the open floating reduces the number of open calls in the `test-one` function, but now the program runs slightly slower as shown in Figure 8.

The reason why this happens is that the opened function (`square'`) has an effect row type of length one. In such case, the Koka runtime already optimizes the use of open by avoiding allocating an explicit evidence vector and directly using the single evidence *as is*. Since no allocation happens, this can be faster, in particular since the current implementation of the restrict operation is not optimized in a similar fashion yet. Instead, it is implemented combining a lambda abstraction and an open operation, as $\mathsf{restrict}\, e \doteq \mathsf{open}(\lambda\_.\, e)()$. We plan to improve the implementation of restrict in which case it should always be beneficial to perform open floating.

6. Future Work

Even though our open floating algorithm is effective, there are still situations where it can be improved. In particular, for certain higher-order programs, such as calls of map and fold, open floating can be improved. Consider the following program.

```
fun map( xs : list⟨a⟩, f : a → e b ) : e list⟨b⟩
fun g( x : int) : ⟨ask, ndet⟩ int

fun f() : ⟨ask, exn, ndet⟩ int
  ...
  map(lst, g)
  ...
```

The Koka compiler wraps the function g with an open call to make the program type check. This gives us the following expression.

$$\mathit{map}\ \mathit{int}\ \mathit{int}\langle \mathit{ask}, \mathit{exn}, \mathit{ndet}\rangle\ \mathit{lst}\ (\mathsf{open}\langle \mathit{ask}, \mathit{exn}, \mathit{ndet}\rangle\ g)$$

The current open floating algorithm does not change the program due to the rule [Open-Preserve] as we discussed in Section 4.3. At runtime, the opened function is applied to each element in the list `lst` by the `map` function. It would be better

to float the open to surround the entire *map* call which reduces the number of open calls to one like restrict⟨*ask*, *ndet*⟩(*map int int*⟨*ask*, *ndet*⟩ *lst g*).

7. Related Work

Our work is in the context of passing dynamic evidence vectors at runtime that correspond to the static effect types, as described by Xie and Leijen [20]. The idea of passing dynamic runtime evidence for static properties is not new and is a standard way of implementing qualified types and type classes [5, 18]. Here, the evidence takes the form of a dictionary of overloaded operations and corresponds to the qualified type constraints. In our work, open adjusts the runtime evidence vectors, while with type classes *instances* are used to modify runtime dictionaries. For example, a function with a `Show a` constraint may call a function with a `Show [a]` constraint. To call this, the received dictionary for `Show a` is transformed at runtime to a `Show [a]` dictionary using the `instance Show a => Show [a]` declaration. Usually, these "evidence adjustments" are called context reduction and generalized by the entailment relation in the theory of qualified types [6]. Peyton Jones et al. [13] explore the design space of sound context reduction in Haskell.

Gaster and Jones [4] present a system for extensible records based on the theory of qualified types. Here, a *lacks* constraint l/r corresponds to a runtime evidence, providing the offset in the record r where the label l would be inserted. When modifying the record, the evidence is also adjusted at runtime to reflect a new offset. For example, if another label is inserted before l, its offset is incremented. A similar mechanism is used in the system of type-indexed rows developed by Shields and Meijer [17].

In all of the above examples, we can imagine transformations similar to open floating that try to minimize the evidence adjustments, although we are not aware of any previous work that addresses this issue specifically.

Our formalization of effect row can roughly be understood as an instance of scoped rows discussed by Morris and McKinna [12]. They define a general *row theory* and *row algebra* with qualified types. Scoped rows are shown as an instance of them. Closed prefix relation is almost represented as left containment relation. Note that our calculus uses polymorphic effect rows such as $\forall\mu.\langle l_1,\ l_2 \mid \mu \rangle$ while scoped rows in [12] do not seem to entail it.

8. Conclusion

In this paper, we formalized open and restrict with their restriction of the closed prefix relation. The formalization clarifies the nature of the Koka language and the constraint of open floating. We also defined the open floating algorithm on the formalized calculus and developed an implementation in Koka. The benchmark shows the effectiveness of open floating and points out room to improve the implementation.

Acknowledgement We acknowledge the reviewers' efforts put into evaluation of our paper. We also appreciate the FLOPS 2022 reviewers' feedback on an earlier version of the paper. Lastly, we thank members of Masuhara laboratory.This work was supported in part by JSPS KAKENHI under Grant No. JP19K24339.

Appendix

We present the well-formedness relation $\vdash_{\mathsf{wf}}$ and definitional equality of types $\vdash_{\mathsf{eq}}$ in Figure 9. The type rules (Figure 4) use these relations.

$$\frac{\vdash_{\mathsf{eq}} \ \lambda\alpha^{\mathsf{k}_2}.\tau_1 \equiv \tau_1' \ : \ \mathsf{k}_2 \to \mathsf{k} \quad \vdash_{\mathsf{eq}} \ \tau_2 \equiv \tau_2' \ : \ \mathsf{k}_2}{\vdash_{\mathsf{eq}} \ \tau_1 \, \tau_2 \equiv \tau_n[\alpha^{\mathsf{k}_2} \coloneqq \tau_2'] \ : \ \mathsf{k}} \ [\textsc{kapp}]$$

$$\frac{}{\vdash_{\mathsf{eq}} \ \alpha^{\mathsf{k}} \equiv \alpha^{\mathsf{k}} \ : \ \mathsf{k}} \ [\textsc{kvar}]$$

$$\frac{}{\vdash_{\mathsf{eq}} \ c^{\mathsf{k}} \equiv c^{\mathsf{k}} \ : \ \mathsf{k}} \ [\textsc{kconst}]$$

$$\frac{\vdash_{\mathsf{eq}} \ \tau_1 \equiv \tau_1' \ : \ \mathsf{k}_1}{\vdash_{\mathsf{eq}} \ \lambda\alpha^*.\tau_1 \equiv \lambda\alpha^*.\tau_1' \ : \ * \to \mathsf{k}_1} \ [\textsc{kabs}]$$

$$\frac{\vdash_{\mathsf{wf}} \ \tau \ : \ *}{\vdash_{\mathsf{eq}} \ \tau \equiv \tau \ : \ *}$$

$$\frac{\vdash_{\mathsf{eq}} \ \tau \equiv \tau \ : \ *}{\vdash_{\mathsf{wf}} \ \tau \ : \ *}$$

$$\frac{\vdash_{\mathsf{wf}} \ \sigma_1 \ : \ * \quad \vdash_{\mathsf{wf}} \ \sigma_2 \ : \ * \quad \vdash_{\mathsf{wf}} \ \epsilon \ : \ \mathsf{eff}}{\vdash_{\mathsf{wf}} \ \sigma_1 \to \epsilon \, \sigma_2 \ : \ *} \ [\textsc{kfun}]$$

$$\frac{\vdash_{\mathsf{wf}} \ \sigma \ : \ *}{\vdash_{\mathsf{wf}} \ \forall\alpha^{\mathsf{k}}.\sigma \ : \ *} \ [\textsc{kall}]$$

$$\frac{\vdash_{\mathsf{wf}} \ l \ : \ \mathsf{lab} \quad \vdash_{\mathsf{wf}} \ \epsilon \ : \ \mathsf{eff}}{\vdash_{\mathsf{wf}} \ \langle l \mid \epsilon \rangle \ : \ \mathsf{eff}} \ [\textsc{keffect}]$$

Figure 9. Well-formedness and Definitional Equality of Types of System F_{pwo}

References

[1] Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. *Effekt: Lightweight Effect Polymorphism for Handlers*. Technical Report. University of Tübingen, Germany. 2020.

[2] Stephen Dolan, Leo White, KC Sivaramakrishnan, Jeremy Yallop, and Anil Madhavapeddy. "Effective Concurrency through Algebraic Effects." In *OCaml Workshop*, 13. 2015.

[3] Mattias Felleisen. "The Theory and Practice of First-Class Prompts." In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 180–190. POPL '88. Association for Computing Machinery, New York, NY, USA. 1988. doi:10.1145/73560.73576.

[4] Ben R. Gaster, and Mark P. Jones. *A Polymorphic Type System for Extensible Records and Variants*. NOTTCS-TR-96-3. University of Nottingham. 1996.

[5] 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.

[6] Mark P. Jones. "Simplifying and Improving Qualified Types." In *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture*, 160–169. FPCA '95. La Jolla, California, USA. 1995. doi:10.1145/224164.224198.

[7] 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.

[8] Daan Leijen. "Structured Asynchrony with Algebraic Effects." In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Type-Driven Development*, 16–29. TyDe 2017. Oxford, UK. 2017. doi:10.1145/3122975.3122977.

[9] Daan Leijen. "Type Directed Compilation of Row-Typed Algebraic Effects." In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, 486–499. 2017.

[10] 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.

[11] Daan Leijen. "Koka Repository." 2019. https://github.com/koka-lang/koka.

[12] J Garrett Morris, and James McKinna. "Abstracting Extensible Data Types: Or, Rows by Any Other Name." *Proceedings of the ACM on Programming Languages* 3 (POPL). ACM New York, NY, USA: 1–28. 2019.

[13] Simon Peyton Jones, Mark Jones, and Erik Meijer. "Type Classes: An Exploration of the Design Space." In *In Haskell Workshop*. 1997.

[14] Gordon D. Plotkin, and Matija Pretnar. "Handling Algebraic Effects." In *Logical Methods in Computer Science*, volume 9. 4. 2013. doi:10.2168/LMCS-9(4:23)2013.

[15] Matija Pretnar, Amr Hany Shehata Saleh, Axel Faes, and Tom Schrijvers. *Efficient Compilation of Algebraic Effects and Handlers*. CW Reports. Department of Computer Science, KU Leuven; Leuven, Belgium. 2017. https://lirias.kuleuven.be/retrieve/472230.

[16] Amr Hany Shehata Saleh, Georgios Karachalias, Matija Pretnar, and Tom Schrijvers. "Explicit Effect Subtyping." In *27th European Symposium on Programming (ESOP)*. 2018.

[17] Mark Shields, and Erik Meijer. "Type-Indexed Rows." In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 261–275. POPL'01. London, United Kingdom. 2001. doi:10.1145/360204.360230.

[18] Philip Wadler, and Stephen Blott. "How to Make Ad-Hoc Polymorphism Less Ad Hoc." In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 60–76. POPL '89. ACM, Austin, Texas, USA. 1989. doi:10.1145/75277.75283.

[19] Ningning Xie, Jonathan Immanuel Brachthäuser, Daniel Hillerström, Philipp Schuster, and Daan Leijen. "Effect Handlers, Evidently." *Proceedings of the ACM on Programming Languages* 4 (ICFP). ACM New York, NY, USA: 1–29. 2020.

[20] Ningning Xie, and Daan Leijen. "Generalized Evidence Passing for Effect Handlers: Efficient Compilation of Effect Handlers to C." *Proc. ACM Program. Lang.* 5 (ICFP). Association for Computing Machinery, New York, NY, USA. Aug. 2021. doi:10.1145/3473576.