Typed Equivalence of Labeled Effect Handlers and Labeled Delimited Control Operators

Kazuki Ikemori
ikemori.k.aa@prg.is.titech.ac.jp
School of Computing,
Tokyo Institute of Technology
Tokyo, Japan

Youyou Cong
cong@c.titech.ac.jp
School of Computing,
Tokyo Institute of Technology
Tokyo, Japan

Hidehiko Masuhara
masuhara@acm.org
School of Computing,
Tokyo Institute of Technology
Tokyo, Japan

ABSTRACT

Algebraic effect handlers and delimited control operators are language facilities for expressing computational effects. Their labeled variations can express multiple kinds of exceptions, multiple states, and so on. We prove that labeled effect handlers and labeled control operators have equal expressive power. To show this, we develop a type-sound calculus for each facility and define macro translations between the typed calculi. The established equivalence can be used to understand and implement one facility in terms of the other.

CCS CONCEPTS

• **Theory of computation → Control primitives**; **Functional constructs**; **Type structures**; **Operational semantics**.

KEYWORDS

algebraic effect handlers, delimited control operators, effect instances, prompt tags, macro expressibility

ACM Reference Format:
Kazuki Ikemori, Youyou Cong, and Hidehiko Masuhara. 2023. Typed Equivalence of Labeled Effect Handlers and Labeled Delimited Control Operators. In *International Symposium on Principles and Practice of Declarative Programming (PPDP 2023), October 22–23, 2023, Lisboa, Portugal.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3610612.3610616

1 INTRODUCTION

Algebraic effect handlers [20, 21] and delimited control operators [4, 6] are being introduced into programming languages as a facility for expressing various computational effects. For example, the OCaml team released a new version that provides effect handlers as a primitive construct[1]. The Haskell steering committee has recently accepted a GHC proposal to add delimited control primitives[2]. The Scala community is also developing similar facilities for implementing suspensions[3].

[1]https://v2.ocaml.org/releases/5.0/manual/effects.html
[2]https://downloads.haskell.org/ghc/9.6.2/docs/users_guide/9.6.1-notes.html
[3]https://github.com/lampepfl/async

Author copy; to appear in Proceedings of PPDP 2023.

Effect handlers and control operators in the above-mentioned languages are *labeled*, allowing the programmer to specify the intended association between effect-yielding constructs and effect-delimiting constructs. These labels are called *effect instances* [1, 2] in the context of the effect handlers, and *prompt tags* in the context of control operators [9, 22]. Labels are useful for mixing different kinds of effects, such as exceptions and state, and for distinguishing between different instances of the same effect, like multiple states.

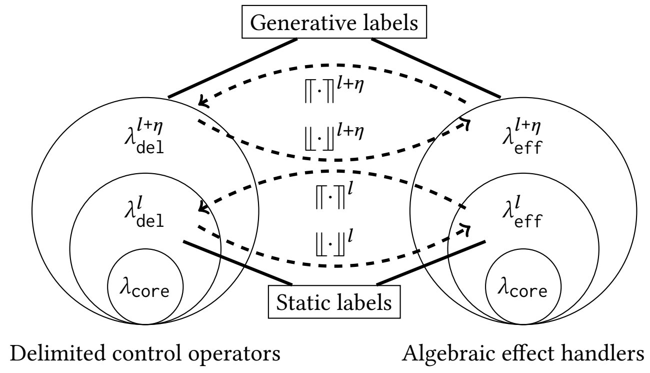
While labeled effect handlers and labeled control operators have the same basic functionality, namely capturing a continuation between two specific points, their precise relationship has not yet been established. This makes it unclear, for instance, whether one can correctly implement one facility in terms of the other. Fortunately, in the unlabeled setting, it has already been proved that effect handlers and control operators have equal expressive power [10, 19]. The result is witnessed by a pair of macro translations [6] between a calculus with effect handlers and a calculus with control operators. Thus, we would naturally expect that the equivalence holds in the labeled setting as well, and that it can be established following a similar approach to previous work.

Establishing the equivalence between labeled facilities requires solving a challenge called the *name escaping problem* [25]. That is, labels may escape their scope during evaluation if not handled carefully by the type system. For labeled effect handlers, there are several sound type systems that address the issue of name escaping [1, 25], but for labeled control operators, there is currently no satisfactory type system.

The aim of this paper is to show the equivalence of expressive power between labeled effect handlers and labeled control operators. The equivalence can be used for both theoretical and implementation purposes. As an example, one can derive the CPS translation of labeled effect handlers from the CPS translation of labeled control operators, as Cong and Asai [3] do in the unlabeled setting. As a different example, one can implement labeled effect handlers using labeled control operators, along the same lines as Kammar et al.'s [12] implementation of unlabeled effect handlers by unlabeled control operators.

To establish the equivalence, we first formalize typed calculi that have labeled effect facilities. We consider two flavors of labels: *static* ones, which are chosen from a globally defined set, and *generative* ones, which are generated during evaluation. For each combination of effect facilities and labels, we design a type system that prevents escaping of labels. Then, for each pair of calculi with the same flavor of labels, we define macro translations [6] between them.

In this paper, we make the following contributions. A pictorial summary can be found in Figure 1.



Generative labels
$\llbracket\cdot\rrbracket^{l+\eta}$
$\llfloor\cdot\rrfloor^{l+\eta}$
$\lambda^{l+\eta}_{\mathtt{del}}$
$\lambda^{l+\eta}_{\mathtt{eff}}$
$\llbracket\cdot\rrbracket^{l}$
$\llfloor\cdot\rrfloor^{l}$
$\lambda^{l}_{\mathtt{del}}$
$\lambda^{l}_{\mathtt{eff}}$
$\lambda_{\mathtt{core}}$
$\lambda_{\mathtt{core}}$
Static labels
Delimited control operators
Algebraic effect handlers

Figure 1: Summary of Contributions

- We formalize four calculi ($\lambda^{l}_{\mathtt{eff}}, \lambda^{l}_{\mathtt{del}}, \lambda^{l+\eta}_{\mathtt{eff}}$, and $\lambda^{l+\eta}_{\mathtt{del}}$) covering all combinations of effect handlers/control operators and static/generative labels, and prove their soundness.
- We define four macro translations ($\llbracket\cdot\rrbracket^{l}$, $\llfloor\cdot\rrfloor^{l}$, $\llbracket\cdot\rrbracket^{l+\eta}$ and $\llfloor\cdot\rrfloor^{l+\eta}$) between effect handlers and control operators with the same flavor of labels, and prove their meaning and type preservation.

The rest of the paper is structured as follows. In Section 2, we give a gentle introduction to effect handlers, control operators, and macro expressibility. In Section 3, we formalize $\lambda_{\mathtt{core}}$, a calculus that serve as the common basis for the extended calculi. Then in Sections 4 and 5, we extend $\lambda_{\mathtt{core}}$ with different labeled effect facilities and investigate the relationship between the extended calculi. We discuss related work in Section 6 and conclude the paper in Section 7.

For space reasons, we do not include the proofs of theorems in the paper, as a technical appendix, which is available at: https://prg.is.titech.ac.jp/en/projects/typesystem/translations-between-effectful-calculi/.

2 BACKGROUND

2.1 Algebraic Effect Handlers

Algebraic effect handlers [20, 21] are a generalization of exception handlers. The facility consists of operations that cause effects and handlers that determine the interpretation of operations. When an operation performs an effect, it is handled by the innermost handler surrounding it. Then, the handler executes the corresponding operation clause. For simplicity, we consider a setting where we have only one operation **do**. This does not lead to loss of expressiveness when effect handlers are labeled, as discussed in [2].

As an example, let us consider a reader effect that represents a read-only state.

$$\begin{array}{ll} & \textbf{handle do } () \textbf{ with } \{x, r.r\ 1; x.x+1\} \\ \to^* & r\ 1 \quad (r = \lambda z.\textbf{handle } z \textbf{ with } \{x, r'.r'\ 1; x.x+1\}) \\ \to & \textbf{handle } 1 \textbf{ with } \{x, r'.r'\ 1; x.x+1\} \\ \to & x+1 \quad (x=1) \\ \to & 2 \end{array}$$

The reduction goes as follows. First, the operation call **do** () is handled by the surrounding handler. The handler executes the operation clause $x, r.r\ 1$, which says it resumes evaluation with the value 1 using the continuation r. The continuation r represents the rest of the computation from the operation call to the innermost handler. The value 1 is the current value of the read-only state. Next, the application $r\ 1$ reduces to **handle** 1 **with** $\{x, r'.r'\ 1; x.x+1\}$. Then, the handler executes the return clause $x.x+1$, which specifies the post-processing of the value that the handled expression reduces to. Therefore, the whole program reduces to 2.

2.2 Delimited Control Operators

Delimited control operators [6, 8, 11] are a traditional tool for manipulating control flow. The facility consists of a control operator that captures the current continuation and a delimiter that specifies the extent of the continuation to be captured. Among different variations of control operators, we consider $\mathtt{shift}_0$ and dollar [18], where the former is a control operator and the latter is a delimiter. Note that we use $\langle\cdot \mid \cdot\rangle$ for the notation of the dollar operator.

As an example, let us consider the following expression, which again encodes a reader effect.

$$\begin{array}{ll} & \langle \textbf{shift}_0\ k.\ k \mid x.\ x+1\rangle\ 1 \\ \to^* & k\ 1 \quad (k = \lambda z.\langle z \mid x.\ x+1\rangle) \\ \to & \langle 1 \mid x.\ x+1\rangle \\ \to & x+1\ (x=1) \\ \to & 2 \end{array}$$

The reduction goes as follows. First, the $\mathtt{shift}_0$ operator captures the continuation k up to the innermost dollar operator, and executes the body k. Second, the application $k\ 1$ reduces to $\langle 1 \mid x.\ x+1\rangle$. Then, the dollar operator executes the return clause $x.x+1$, which specifies the post-processing of the value that the handled expression reduces to. Thus, the whole program reduces to 2.

2.3 Algebraic Effect Handlers with Effect Instances

Plain effect handlers are inconvenient for dealing with multiple instances of the same effect because any operation is handled by the innermost corresponding handler. In other words, we do not have an intuitive means to express our intention of which operation is handled by which handler. Consider the following expression, which has two operations and two different handlers for the same reader effect. Let $f_1 = x, r.r\ 1$, $f_2 = x, r.r\ 2$, and $r_{id} = x.x$.

$$\begin{array}{ll} & \textbf{handle handle do } () + \textbf{do } () \textbf{ with } \{f_1; x\} \textbf{ with } \{f_2; x\} \\ \to^* & \textbf{handle } r\ 1 \textbf{ with } \{f_2; r_{id}\} \\ & \quad (r = \lambda z.\textbf{handle } z + \textbf{do } () \textbf{ with } \{f_1; r_{id}\}) \\ \to & \textbf{handle handle } 1 + \textbf{do } () \textbf{ with } \{f_1; r_{id}\} \textbf{ with } \{f_2; r_{id}\} \\ \to^* & \textbf{handle } r\ 1 \textbf{ with } \{f_2; r_{id}\} \\ & \quad (r = \lambda z.\textbf{handle } 1 + z \textbf{ with } \{f_1; r_{id}\}) \\ \to^* & \textbf{handle handle } 1 + 1 \textbf{ with } \{f_1; r_{id}\} \textbf{ with } \{f_2; r_{id}\} \\ \to^* & 2 \end{array}$$

In the above reduction steps, both operation calls **do** () are handled by the innermost handler, whose operation clause is $x, r.r\ 1\ (= f_1)$. Hence, the expression is reduced to 2 (= 1+1) instead of 3 (= 1+2). If we wish the second operation to be handled by the outer handler, this is not the result we want.

There are several ways of allowing more flexible association between operations and handlers. One way is to use a special operator (called "mask" in [16] and "lift" in [2, 19]) for skipping intervening handlers. This is however unsatisfactory, as it requires the programmer to know the exact number and order of handlers surrounding each operation. A different and more convenient way is to use *effect instances* [1, 2, 25]. This feature allows the programmer to specify the intended handler using a label, without requiring them to know what other handlers are surrounding an operation.

Let us rewrite our previous example using two effect instances l and l'.

$$\begin{array}{l}
\textbf{handle}\langle l'\rangle\ \textbf{handle}\langle l\rangle \\
\quad \textbf{do}\langle l\rangle\ () + \textbf{do}\langle l'\rangle\ ()\ \textbf{with}\{f_1; r_{id}\}\ \textbf{with}\{f_2; r_{id}\} \\
\to^* \textbf{handle}\langle l'\rangle\ r\ 1\ \textbf{with}\{f_2; r_{id}\} \\
\quad (r = \lambda z.\textbf{handle}\langle l\rangle\ z + \textbf{do}\langle l'\rangle\ ()\ \textbf{with}\{f_1; r_{id}\}) \\
\to \textbf{handle}\langle l'\rangle\ \textbf{handle}\langle l\rangle\ 1 + \textbf{do}\langle l'\rangle\ ()\ \textbf{with}\{f_1; r_{id}\}\ \textbf{with}\{f_2; r_{id}\} \\
\to^* r\ 2 \\
\quad (r = \lambda z.\textbf{handle}\langle l'\rangle\ \textbf{handle}\langle l\rangle\ 1 + z\ \textbf{with}\{f_1; r_{id}\}\ \textbf{with}\{f_2; r_{id}\}) \\
\to \textbf{handle}\langle l'\rangle\ \textbf{handle}\langle l\rangle\ 1 + 2\ \textbf{with}\{f_1; r_{id}\}\ \textbf{with}\{f_2; r_{id}\} \\
\to^* 3
\end{array}$$

In the above reduction steps, the operation $\textbf{do}\langle l\rangle$ () is handled by the inner handler, and the operation $\textbf{do}\langle l'\rangle$ () is handled by the outer handler. Hence, the whole expression reduces to 3 (= 1 + 2).

2.4 Delimited Control Operators with Prompt Tags

Similar to plain effect handlers, plain delimited control operators are inconvenient for expressing advanced control flow. The reason is that any $\texttt{shift}_0$ operator captures the continuation up to the innermost dollar operator. In other words, we do not have an intuitive means to express our intention of which $\texttt{shift}_0$ operator captures the continuation up to which dollar operator. Consider the following expression, which has two $\texttt{shift}_0$ operators and two dollar operators.

$$\begin{array}{ll}
 & \langle\langle \textbf{shift}_0\ k.\ k + \textbf{shift}_0\ k.\ k \mid x.\ x\rangle\ 1 \mid x.\ \lambda y.x\rangle\ 2 \\
\to^* & \langle k\ 1 \mid x.\ \lambda y.x\rangle\ 2 \qquad (k = \lambda z.\langle z + \textbf{shift}_0\ k.\ k \mid x.\ x\rangle) \\
\to & \langle\langle 1 + \textbf{shift}_0\ k.\ k \mid x.\ x\rangle \mid x.\ \lambda y.x\rangle\ 2 \\
\to^* & \langle k\ 1 \mid x.\ \lambda y.x\rangle\ 2 \qquad (k = \lambda z.\langle 1 + z \mid x.\ x\rangle) \\
\to & \langle\langle 1 + 1 \mid x.\ x\rangle \mid x.\ \lambda y.x\rangle\ 2 \\
\to^* & 2
\end{array}$$

In the above reduction steps, both $\texttt{shift}_0$ operators capture the continuation up to the innermost dollar operator. Hence, the expression reduces to 2 (= 1 + 1) instead of 3 (= 1 + 2). If we wish the second operator to capture the continuation up to the outer delimiter, this is not the result we want.

To allow more flexible association between operators and delimiters, several programming languages provide *prompt tags* [11, 14], just like how we extended effect handlers with effect instances. Below is a modification of our running example, which uses two prompt tags l and l'.

$$\begin{array}{ll}
 & \langle\langle \textbf{shift}_0\langle l\rangle\ k.\ k + \textbf{shift}_0\langle l'\rangle\ k.\ k \mid x.\ x\rangle_l\ 1 \mid x.\ x\rangle_{l'}\ 2 \\
\to^* & \langle k\ 1 \mid x.\ x\rangle_{l'}\ 2 \qquad (k = \lambda z.\langle z + \textbf{shift}_0\langle l'\rangle\ k.\ k \mid x.\ x\rangle_l) \\
\to & \langle\langle 1 + \textbf{shift}_0\langle l'\rangle\ k.\ k \mid x.\ x\rangle_l \mid x.\ x\rangle_{l'}\ 2 \\
\to^* & k\ 2 \qquad (k = \lambda z.\langle\langle 1 + z \mid x.\ x\rangle_l \mid x.\ x\rangle_{l'}) \\
\to & \langle\langle 1 + 2 \mid x.\ x\rangle \mid x.\ x\rangle \\
\to^* & 3
\end{array}$$

In the above reduction steps, the first $\textbf{shift}_0\langle l\rangle\ k.\ k$ operator captures the continuation up to the inner dollar operator, and the second $\textbf{shift}_0\langle l'\rangle\ k.\ k$ operator captures the continuation up to the outer dollar operator. Hence, the whole expression reduces to 3 (= 1 + 2).

2.5 Macro Translations between Algebraic Effect Handlers and Delimited Control Operators

By comparing effect handlers and control operators, we can easily see that they have similar functionalities. This makes us wonder whether the two facilities can express each other. The question has been answered affirmatively by previous work [10, 19], in terms of Felleisen's macro expressibility [7].

The concept of macro expressibility allows us to compare the expressive power between different languages. Intuitively, macro expressibility is defined as follows. Suppose we have two languages, L and L_+. Here, L_+ is an extension of L with some additional features. Now, if there exists a syntax-directed, meaning-preserving translation from L_+ to L, we say that L_+ is macro expressible by L. We call the translation a macro translation, and we view its existence as a witness that the two languages have equal expressive power. Thus, we can show the equivalence between two calculi by defining a pair of macro translations between them.

In this paper, we extend the macro translations of Piróg et al. [19] with effect instances and prompt tags. In the following sections, we formalize individual calculi and macro translations between different calculi.

3 $\lambda_{\textsf{core}}$: CORE CALCULUS

In this section, we present $\lambda_{\textsf{core}}$, which serves as the basis of the effectful calculi discussed in later sections. The calculus is based on the core calculus of Piróg et al. [19] and features subtyping, polymorphism, and effect rows. In the definition of the syntax and typing, we highlight the changes from Piróg et al. with a gray background.

3.1 Syntax

Kinds, Types and Effect Rows. Figure 2 top defines the syntax of kinds, types, and effect rows of $\lambda_{\textsf{core}}$. Kinds include the value type kind **T**, the effect type kind **E**, and the effect row kind **R**. Types include type variables α, function types $\sigma_1 \to_\rho \sigma_2$, and quantified types $\forall\alpha :: \kappa.\sigma$. Among the three components of function types, σ_1 represents the input type, σ_2 represents the output type, and ρ

Kind	κ	::=	$\mathbf{T}$	(value type)
		$\mid$	$\mathbf{E}$	(effect type)
		$\mid$	$\mathbf{R}$	(effect row type)
Type	σ, τ	::=	α	(type variable)
		$\mid$	$\sigma \rightarrow_{\rho} \sigma$	(function type)
		$\mid$	$\forall \alpha :: \kappa.\sigma$	(quantified type)
Effect row	ρ	::=	ι	(empty row)
		$\mid$	α	(row variable)
		$\mid$	$\epsilon \cdot \rho$	(extended row)
Expression	e	::=	v	(value)
		$\mid$	$e\ e$	(application)
Value	v	::=	x	(variable)
		$\mid$	$\lambda x.e$	(function)
Type variable env	Δ	::=	$\emptyset \mid \Delta, \alpha :: \kappa$	
Variable env	Γ	::=	$\emptyset \mid \Gamma,\ x : \sigma$	
Label env	Σ	::=	$\emptyset \mid \Sigma, l$	

Type Var $\ni \alpha, \beta, \ldots$ Expression Var $\ni x, y, \ldots$ Label $\ni l, l_1, \ldots$

Evaluation Context

$$E ::= \square \mid v\ E \mid E\ e$$

Reduction Rules

$$\frac{}{(\lambda x.e)\ v \mapsto e\{v/x\}}\ [\textsc{Beta}] \qquad \frac{e \mapsto e'}{E[e] \longrightarrow E[e']}\ [\textsc{Step}]$$

Figure 2: Syntax and Semantics of $\lambda_{\mathtt{core}}$

represents the effects of the function body. Following Leijen [16, 17] and Xie et al. [24], we define an effect row as either an empty row ι, a type variable α with an effect row kind $\mathbf{R}$, or an extension $\epsilon \cdot \rho$ of an effect row ρ with an effect ϵ. Effects take different forms in different calculi, but they all consist of several types and a label, which is a uniform representation of effect instances and prompt tags. Note that an effect row cannot be extended with a type variable of kind $\mathbf{E}$. We will discuss the reason in Section 4.1.4.

Expressions and Values. We define the syntax of expressions and values of $\lambda_{\mathtt{core}}$ in Figure 2. The definition is exactly that of the standard call-by-value lambda calculus.

3.2 Kinding Rules

Figure 3 top defines the kinding rules of $\lambda_{\mathtt{core}}$. We use a kinding judgment of the form $\Delta \mid \Sigma \vdash \tau :: \kappa$. The judgment states that a type τ has kind κ under type variable environment Δ and label environment Σ. We explicitly write Σ because it is necessary to state the preservation property (Theorems 9 and 11 in Sections 5.1.4 and 5.2.4, respectively), where the label environments before and after evaluation may be different. The kinding rules are standard.

3.3 Equivalence Rules

Figure 3 upper middle defines the equivalence rules of $\lambda_{\mathtt{core}}$. We use an equivalence judgment of the form $\Delta \mid \Sigma \vdash \sigma \equiv \sigma'$. The judgment states that types σ and σ' are equivalent under type variable environment Δ and label environment Σ. Roughly speaking, these rules allow us to swap effects with distinct labels. Note that these rules do not exist in the calculus of Piróg et al. [19]

The equivalence rules are mostly standard, except the ESwap rule. It allows swapping of effects ϵ_1 and ϵ_2 only if they have different labels. Here, $\lceil \cdot \rceil$ is a meta-level function that extracts the label of an effect. We will elaborate more on this rule in Section 4.1.4.

3.4 Subtyping Rules

Figure 3 lower middle defines the subtyping rules of $\lambda_{\mathtt{core}}$. We use a subtyping judgment of the form $\Delta \mid \Sigma \vdash \sigma <: \sigma'$. The judgment states that types σ and σ' have a subtype relation under type variable environment Δ and label environment Σ.

The subtyping rules consist of reflexivity, transitivity, and structural subtyping rules. The rules are all standard. One thing to note here is that the SRefl rule has an extra premise $\Delta \mid \Sigma \vdash \sigma_1 \equiv \sigma_2$ compared to the corresponding rule of Piróg et al. This allow us to derive, for instance, $\Delta \mid \Sigma \vdash \epsilon_2 \cdot \epsilon_1 \cdot \rho <: \epsilon_1 \cdot \epsilon_2 \cdot \rho$ using $\Delta \mid \Sigma \vdash \epsilon_2 \cdot \epsilon_1 \cdot \rho \equiv \epsilon_1 \cdot \epsilon_2 \cdot \rho$, which intuitively holds. Without the additional premise, we cannot derive the above subtyping relation as the original rules of Piróg et al. do not allow swapping of effects.

3.5 Typing Rules

We define the typing rules of $\lambda_{\mathtt{core}}$ in Figure 3 bottom. We use a typing judgment of the form $\Delta \mid \Gamma \mid \Sigma \vdash e : \tau/\rho$. The judgment states that expression e has type τ under type variable environment Δ, variable environment Γ, and label environment Σ, and may perform effects ρ.

The typing rules are mostly standard. The only difference from Piŕog et al. is that our rules restrict polymorphism to value types and effect rows: observe that the kind κ in Gen and Inst must be either $\mathbf{T}$ or $\mathbf{R}$. We will detail the design rationale in Section 4.1.4.

3.6 Operational Semantics

Figure 2 bottom defines the operational semantics of $\lambda_{\mathtt{core}}$. The semantics is based on the call-by-value evaluation strategy, and is associated with an inductive definition of evaluation contexts. The definition is completely standard and identical to Piróg et al.

4 STATIC EFFECT INSTANCES AND PROMPT TAGS

In this section, we present two extensions of the core calculus and a pair of macro translations between the two extended calculi. One of the extended calculi is called $\lambda^{l}_{\mathtt{eff}}$, which features effect handlers and static effect instances. The other calculus is called $\lambda^{l}_{\mathtt{del}}$, which features control operators and static prompt tags.

4.1 $\lambda^{l}_{\mathtt{eff}}$: Algebraic Effect Handlers with Static Effect Instances

4.1.1 Syntax. We define the syntax of $\lambda^{l}_{\mathtt{eff}}$ in Figure 4. For brevity, we only show the changes to the $\lambda_{\mathtt{core}}$ syntax. Kinds are extended

$$\boxed{\Delta \mid \Sigma \vdash \tau :: \kappa}$$

$$\frac{\alpha :: \kappa \in \Delta}{\Delta \mid \Sigma \vdash \alpha :: \kappa}\;[\text{KVar}] \qquad \frac{\Delta, \alpha :: \kappa \mid \Sigma \vdash \tau :: \mathbf{T}}{\Delta \mid \Sigma \vdash \forall \alpha :: \kappa.\tau :: \mathbf{T}}\;[\text{KGen}] \qquad \frac{}{\Delta \mid \Sigma \vdash \iota :: \mathbf{R}}\;[\text{KEmpty}]$$

$$\frac{\Delta \mid \Sigma \vdash \tau_1 :: \mathbf{T} \qquad \Delta \mid \Sigma \vdash \rho :: \mathbf{R} \qquad \Delta \mid \Sigma \vdash \tau_2 :: \mathbf{T}}{\Delta \mid \Sigma \vdash \tau_1 \rightarrow_\rho \tau_2 :: \mathbf{T}}\;[\text{KArrow}] \qquad \frac{\Delta \mid \Sigma \vdash \epsilon :: \mathbf{E} \qquad \Delta \mid \Sigma \vdash \rho :: \mathbf{R}}{\Delta \mid \Sigma \vdash \epsilon \cdot \rho :: \mathbf{R}}\;[\text{KRow}]$$

$$\boxed{\Delta \mid \Sigma \vdash \sigma \equiv \sigma'}$$

$$\frac{\Delta \mid \Sigma \vdash \sigma :: \kappa}{\Delta \mid \Sigma \vdash \sigma \equiv \sigma}\;[\text{ERefl}] \qquad \frac{\Delta, \alpha :: \kappa \mid \Sigma \vdash \tau_1 \equiv \tau_2}{\Delta \mid \Sigma \vdash \forall \alpha :: \kappa.\tau_1 \equiv \forall \alpha :: \kappa.\tau_2}\;[\text{EGen}] \qquad \frac{\Delta \mid \Sigma \vdash \rho_1 \equiv \rho_2 \qquad \Delta \mid \Sigma \vdash \epsilon :: \mathbf{E}}{\Delta \mid \Sigma \vdash \epsilon \cdot \rho_1 \equiv \epsilon \cdot \rho_2}\;[\text{ERow}]$$

$$\frac{\Delta \mid \Sigma \vdash \tau_2^1 \equiv \tau_1^1 \qquad \Delta \mid \Sigma \vdash \rho_1 \equiv \rho_2 \qquad \Delta \mid \Sigma \vdash \tau_1^2 \equiv \tau_2^2}{\Delta \mid \Sigma \vdash \tau_1^1 \rightarrow_{\rho_1} \tau_2^1 \equiv \tau_1^2 \rightarrow_{\rho_2} \tau_2^2}\;[\text{EArrow}] \qquad \frac{\Delta \mid \Sigma \vdash \rho_1 \equiv \rho_2 \qquad \Delta \mid \Sigma \vdash \rho_2 \equiv \rho_3}{\Delta \mid \Sigma \vdash \rho_1 \equiv \rho_3}\;[\text{ETrans}]$$

$$\frac{\Delta \mid \Sigma \vdash \rho_1 \equiv \rho_2 \qquad \Delta \mid \Sigma \vdash \epsilon_1 :: \mathbf{E} \qquad \Delta \mid \Sigma \vdash \epsilon_2 :: \mathbf{E} \qquad \lceil \epsilon_1 \rceil \neq \lceil \epsilon_2 \rceil}{\Delta \mid \Sigma \vdash \epsilon_1 \cdot \epsilon_2 \cdot \rho_1 \equiv \epsilon_2 \cdot \epsilon_1 \cdot \rho_2}\;[\text{ESwap}]$$

$$\boxed{\Delta \mid \Sigma \vdash \sigma <: \sigma'}$$

$$\frac{\Delta \mid \Sigma \vdash \sigma_1 \equiv \sigma_2}{\Delta \mid \Sigma \vdash \sigma_1 <: \sigma_2}\;[\text{SRefl}] \qquad \frac{\Delta, \alpha :: \kappa \mid \Sigma \vdash \tau_1 <: \tau_2}{\Delta \mid \Sigma \vdash \forall \alpha :: \kappa.\tau_1 <: \forall \alpha :: \kappa.\tau_2}\;[\text{SGen}] \qquad \frac{\Delta \mid \Sigma \vdash \rho :: \mathbf{R}}{\Delta \mid \Sigma \vdash \iota <: \rho}\;[\text{SEmpty}]$$

$$\frac{\Delta \mid \Sigma \vdash \tau_1^2 <: \tau_1^1 \qquad \Delta \mid \Sigma \vdash \rho_1 <: \rho_2 \qquad \Delta \mid \Sigma \vdash \tau_2^1 <: \tau_2^2}{\Delta \mid \Sigma \vdash \tau_1^1 \rightarrow_{\rho_1} \tau_2^1 <: \tau_1^2 \rightarrow_{\rho_2} \tau_2^2}\;[\text{SArrow}]$$

$$\frac{\Delta \mid \Sigma \vdash \rho_1 <: \rho_2 \qquad \Delta \mid \Sigma \vdash \epsilon :: \mathbf{E}}{\Delta \mid \Sigma \vdash \epsilon \cdot \rho_1 <: \epsilon \cdot \rho_2}\;[\text{SRow}] \qquad \frac{\Delta \mid \Sigma \vdash \rho_1 <: \rho_2 \qquad \Delta \mid \Sigma \vdash \rho_2 <: \rho_3}{\Delta \mid \Sigma \vdash \rho_1 <: \rho_3}\;[\text{STrans}]$$

$$\boxed{\Delta \mid \Gamma \mid \Sigma \vdash e : \tau / \rho}$$

$$\frac{x : \tau \in \Gamma}{\Delta \mid \Gamma \mid \Sigma \vdash x : \tau/\iota}\;[\text{Var}] \qquad \frac{\Delta \mid \Sigma \vdash \tau_1 <: \tau_2 \qquad \Delta \mid \Sigma \vdash \rho_1 <: \rho_2 \qquad \Delta \mid \Gamma \mid \Sigma \vdash e : \tau_1/\rho_1}{\Delta \mid \Gamma \mid \Sigma \vdash e : \tau_2/\rho_2}\;[\text{Sub}]$$

$$\frac{\Delta, \alpha :: \kappa \mid \Gamma \mid \Sigma \vdash e : \tau/\iota \qquad \kappa \in \{\mathbf{T}, \mathbf{R}\}}{\Delta \mid \Gamma \mid \Sigma \vdash e : \forall \alpha :: \kappa.\tau/\iota}\;[\text{Gen}] \qquad \frac{\Delta \mid \Sigma \vdash \sigma :: \kappa \qquad \Delta \mid \Gamma \mid \Sigma \vdash e : \forall \alpha :: \kappa.\tau/\rho \qquad \kappa \in \{\mathbf{T}, \mathbf{R}\}}{\Delta \mid \Gamma \mid \Sigma \vdash e : \tau\{\sigma/\alpha\}/\rho}\;[\text{Inst}]$$

$$\frac{\Delta \mid \Gamma \mid \Sigma \vdash e_1 : \tau_1 \rightarrow_\rho \tau_2/\rho \qquad \Delta \mid \Gamma \mid \Sigma \vdash e_2 : \tau_1/\rho}{\Delta \mid \Gamma \mid \Sigma \vdash e_1\ e_2 : \tau_2/\rho}\;[\text{App}] \qquad \frac{\Delta \mid \Sigma \vdash \tau_1 :: \mathbf{T} \qquad \Delta \mid \Gamma, x : \tau_1 \mid \Sigma \vdash e : \tau_2/\rho}{\Delta \mid \Gamma \mid \Sigma \vdash \lambda x.e : \tau_1 \rightarrow_\rho \tau_2/\iota}\;[\text{Abs}]$$

Figure 3: Kinding, Equivalence, Subtyping and Typing Rules of λ_{core}

with **L** for label types. Types are extended with label types l. Effects take the form $(\Delta'.\tau_1 \Rightarrow \tau_2)_l$, which reads: the operation has input and output types τ_1 and τ_2 (which may refer to type variables in Δ'), and it must be handled by a handler with label l. Labels are chosen from a set Σ of statically defined labels. Expressions are extended with operations and handlers. An operation $\mathbf{do}\langle l\rangle\ v$ performs an effect when given an input v and a label l specifying the matching handler. A handler $\mathbf{handle}\langle l\rangle\ e\ \mathbf{with}\ \{x, r.e_h; x.e_r\}$ handles the computation e with either the operation clause $x, r.e_h$ (if e performs an operation with label l) or the return clause (if e does not perform an operation).

4.1.2 Kinding, Equivalence, and Typing Rules. We define the kinding, equivalence, and typing rules of λ^l_{eff} in Figure 5. The KLabel rule assigns kind **L** to static labels, whereas the KIEff rule assigns kind **E** to effects. The EIEff rule determines whether two effects are equivalent or not. The Do rule introduces an effect $(\Delta'.\tau_1 \Rightarrow \tau_2)_l$ indexed by label l, while instantiating the input and output types τ_1 and τ_2 of an operation via type variable substitution δ. Following Piróg et al., we define the well-formedness of type variable substitution as follows.

Definition 1 (Well-formedness of type variable substitutions)**.**
$\Delta \mid \Sigma \vdash \delta :: \Delta'$
$\overset{def}{\iff} \mathbf{dom}(\delta') = \mathbf{dom}(\Delta') \ \wedge\ \forall \alpha \in \mathbf{dom}(\delta), \Delta \mid \Sigma \vdash \delta(\alpha) :: \Delta'(\alpha)$

The Handle rule discharges an effect indexed by the label l, leaving the residual effects ρ to be handled by outer handlers.

4.1.3 Operational Semantics. We define the operational semantics of λ^l_{eff} in Figure 4. The EReturn rule states that a handler construct reduces to the return clause $e_r\{v/x\}$ if the handled expression is a

Kind	κ	::=	$\ldots$	
		$\mid$	$\mathbf{L}$	(label type)
Type	σ, τ	::=	$\cdots$	
		$\mid$	ℓ	(label)
Effect	ϵ	::=	$(\Delta'.\tau_1 \Rightarrow \tau_2)_\ell$	
Label	ℓ	::=	l	(static label)
Expression	e	::=	$\ldots$	
		$\mid$	$\mathbf{do}\langle\ell\rangle\ v$	(do)
		$\mid$	$\mathbf{handle}\langle l\rangle\ e$ $\mathbf{with}\ \{x, r.e_h; x.e_r\}$	(handle)

Evaluation Context

$$E \ ::= \ \cdots \mid \mathbf{handle}\langle l\rangle\ E\ \mathbf{with}\ \{x, r.e_h; x.e_r\}$$

Reduction Rules

$$\frac{}{\mathbf{handle}\langle l\rangle\ v\ \mathbf{with}\ \{x, r.e_h; x.e_r\} \mapsto e_r\{v/x\}}\ [\text{EReturn}]$$

$$\frac{l \notin \lceil E\rceil \quad v_c = \lambda z.\mathbf{handle}\langle l\rangle\ E[z]\ \mathbf{with}\ \{x, r.e_h; x.e_r\}}{\begin{array}{r}\mathbf{handle}\langle l\rangle\ E[\mathbf{do}\langle l\rangle\ v]\ \mathbf{with}\ \{x, r.e_h; x.e_r\}\\ \mapsto e_h\{v/x\}\{v_c/r\}\end{array}}\ [\text{EHandle}]$$

Figure 4: Syntax and Semantics of $\lambda^l_{\mathtt{eff}}$(extensions)

value. The EHandle rule states that an operation $\mathbf{do}\langle l\rangle\ v$ is handled by the innermost handler indexed by the same label l. The premise $l \notin \lceil E\rceil$ states that there is no handler indexed by the label l in the evaluation context E. Here, $\lceil\cdot\rceil$ is a meta-level function for extracting labels from evaluation context, and is defined as follows.

Definition 2 (Label extraction)**.**

$$\lceil\square\rceil \ ::= \ \emptyset \quad \lceil E\ e\rceil \ ::= \ \lceil E\rceil \quad \lceil v\ E\rceil \ ::= \ \lceil E\rceil$$
$$\lceil\mathbf{handle}\langle l\rangle\ E\ \mathbf{with}\ \{x, r.e_h; x.e_r\}\rceil \ ::= \ l, \lceil E\rceil$$
$$\lceil\iota\rceil \ ::= \ \bullet, \quad \lceil\epsilon \cdot \rho\rceil \ ::= \ \lceil\epsilon\rceil, \lceil\rho\rceil \quad \lceil(\Delta'.\tau_1 \Rightarrow \tau_2)_\ell\rceil \ ::= \ \ell$$

4.1.4 Reasons Behind Design Decisions.

Restriction on ESwap. As we mentioned in Section 3.3, we do not allow swapping of effects with the same label. The reason is that, without this restriction, the type system would be unsound. More specifically, it would accept expressions that get stuck during evaluation. Below is an example of such an expression.

$$\begin{array}{ll} & \mathbf{handle}\langle l\rangle \\ & \quad \mathbf{handle}\langle l\rangle\ \mathbf{do}\langle l\rangle\ ()\ \mathbf{with}\ \{x, r.r\ (x+1); x.x\} \\ & \mathbf{with}\ \{x, r.r\ 1; x.x\} \\ \rightarrow & \mathbf{handle}\langle l\rangle\ r\ (() + 1)\ \mathbf{with}\ \{x, r.r\ 1; x.x\} \\ & \quad (r = \lambda z.\mathbf{handle}\langle l\rangle\ z\ \mathbf{with}\ \{x, r.r\ (x+1); x.x\}) \\ \not\rightarrow & \end{array}$$

Observe that the evaluation of the above expression gets stuck, since we cannot add a unit value and an integer value. This stuck state is caused by the fact that the unit-taking operation $\mathbf{do}\langle l\rangle\ ()$ is surrounded by a handler with an int-taking operation clause $x, r.r\ (x+1)$.

The above expression would be wrongly judged well-typed if we replaced ESwap with BadESwap.

$$\frac{\Delta \mid \Sigma \vdash \rho_1 \equiv \rho_2 \quad \Delta \mid \Sigma \vdash \epsilon_1 :: \mathbf{E} \quad \Delta \mid \Sigma \vdash \epsilon_2 :: \mathbf{E}}{\Delta \mid \Sigma \vdash \epsilon_1 \cdot \epsilon_2 \cdot \rho_1 \equiv \epsilon_2 \cdot \epsilon_1 \cdot \rho_2}\ [\text{BadESwap}]$$

Let us explain how we type the expression step by step. First, we use the Do and Sub rules to type the operation call. Note that the top-most effect has input type $()$, meaning that the operation must be surrounded by a handler that has a unit-taking operation clause.

$$\emptyset \mid \emptyset \mid l \vdash \mathbf{do}\langle l\rangle\ () : \mathtt{int}/(\emptyset.() \Rightarrow \mathtt{int})_l \cdot (\emptyset.\mathtt{int} \Rightarrow \mathtt{int})_l$$

Next, we use the BadESwap and Sub rules to swap the effects. Observe that the top-most effect now has input type $\mathtt{int}$, meaning that the operation can be surrounded by a handler that has an int-taking operation clause.

$$\emptyset \mid \emptyset \mid l \vdash \mathbf{do}\langle l\rangle\ () : \mathtt{int}/(\emptyset.\mathtt{int} \Rightarrow \mathtt{int})_l \cdot (\emptyset.() \Rightarrow \mathtt{int})_l$$

Then, we use the Handle rule to obtain the following judgment, where $E_0 = \mathbf{handle}\langle l\rangle\ \square\ \mathbf{with}\ \{x, r.r\ 1; x.x\}$.

$$\emptyset \mid \emptyset \mid l \vdash E_0[\mathbf{handle}\langle l\rangle\ \mathbf{do}\langle l\rangle\ ()\ \mathbf{with}\ \{x, r.r\ (x+1); x.x\}] : \mathtt{int}/\iota$$

Thus, we arrive at the conclusion that the expression is well-typed.

On the other hand, using the ESwap rule from Figure 3, we can correctly reject the above expression. The reason is that we cannot wrap a handler that has an int-talking operation clause around a unit-taking operation by swapping the effects.

Restriction on Syntax of Effect Rows. As we mentioned in Section 3.1, we do not allow extension of an effect row with a type variable that has kind **E**. The restriction is necessary for maintaining type soundness. We have already seen that swapping effects with the same label is problematic. To see whether two labels are the same or not, we need to extract labels from effects. However, if the effect is a type variable, we cannot extract labels from it.

Restriction on Gen and Inst Rules. As we mentioned in Section 3.5, we do not allow polymorphism on types of kind **E** in rules Gen and Inst. The reason for this restriction is the same as the reason behind the restriction on row extension. That is, if we allowed generalization and instantiation of type variables of kind **E**, we would not always be able to extract labels from effects.

4.1.5 Type Soundness. We prove the type soundness of $\lambda^l_{\mathtt{eff}}$ by showing the progress and preservation theorems [23]. Both theorems can be proved by induction on the typing derivation.

Theorem 1 (Preservation of $\lambda^l_{\mathtt{eff}}$)**.**
If $\Delta \mid \Gamma \mid \Sigma \vdash e : \tau/\iota$ and $e \longrightarrow e'$ then $\Delta \mid \Gamma \mid \Sigma \vdash e' : \tau/\iota$.

Theorem 2 (Progress of $\lambda^l_{\mathtt{eff}}$)**.**
If $\emptyset \mid \emptyset \mid \Sigma \vdash e : \tau/\iota$ then e is a value or there exists e' such that $e \longrightarrow e'$.

4.2 $\lambda^l_{\mathtt{del}}$: Delimited Control Operators with Static Prompt Tags

4.2.1 Syntax. We define the syntax of $\lambda^l_{\mathtt{del}}$ in Figure 6. Similar to $\lambda^l_{\mathtt{eff}}$, kinds and types are extended with kind **L** for label types and label types l, respectively. Effects take the form $(\Delta'.\ \tau/\rho)_l$ which reads: a continuation has an output type τ and an effect

$\boxed{\Delta \mid \Sigma \vdash \tau :: \kappa}$

$$\frac{l \in \Sigma}{\Delta \mid \Sigma \vdash l :: \mathbf{L}}\,[\textsc{KLabel}] \qquad \frac{\Delta, \Delta' \mid \Sigma \vdash \tau_1 :: \mathbf{T} \qquad \Delta, \Delta' \mid \Sigma \vdash \tau_2 :: \mathbf{T} \qquad \Delta \mid \Sigma \vdash l :: \mathbf{L}}{\Delta \mid \Sigma \vdash (\Delta'.\tau_1 \Rightarrow \tau_2)_l :: \mathbf{E}}\,[\textsc{KIEff}]$$

$\boxed{\Delta \mid \Sigma \vdash \rho \equiv \rho'}$

$$\frac{\Delta, \Delta' \mid \Sigma \vdash \tau_1 \equiv \tau_1' \qquad \Delta, \Delta' \mid \Sigma \vdash \tau_2 \equiv \tau_2' \qquad \Delta \mid \Sigma \vdash l :: \mathbf{L}}{\Delta \mid \Sigma \vdash (\Delta'.\tau_1 \Rightarrow \tau_2)_l \equiv (\Delta'.\tau_1' \Rightarrow \tau_2')_l}\,[\textsc{EIEff}]$$

$\boxed{\Delta \mid \Gamma \mid \Sigma \vdash e : \tau/\rho}$

$$\frac{\begin{array}{c}\Delta \mid \Sigma \vdash (\Delta'.\tau_1 \Rightarrow \tau_2)_l :: \mathbf{E} \\ \Delta \mid \Sigma \vdash \delta :: \Delta' \qquad \Delta \mid \Gamma \mid \Sigma \vdash v : \delta(\tau_1)/\iota\end{array}}{\Delta \mid \Gamma \mid \Sigma \vdash \mathbf{do}\langle l\rangle\ v : \delta(\tau_2)/(\Delta'.\tau_1 \Rightarrow \tau_2)_l \cdot \iota}\,[\textsc{Do}] \qquad \frac{\begin{array}{c}\Delta, \Delta' \mid \Gamma, x : \tau_1, r : \tau_2 \to_\rho \tau_r \mid \Sigma \vdash e_h : \tau_r/\rho \qquad \Delta \mid \Sigma \vdash l :: \mathbf{L} \\ \Delta \mid \Gamma \mid \Sigma \vdash e : \tau/(\Delta'.\tau_1 \Rightarrow \tau_2)_l \cdot \rho \qquad \Delta \mid \Gamma, x : \tau \mid \Sigma \vdash e_r : \tau_r/\rho\end{array}}{\Delta \mid \Gamma \mid \Sigma \vdash \mathbf{handle}\langle l\rangle\ e\ \mathbf{with}\ \{x, r.e_h; x.e_r\} : \tau_r/\rho}\,[\textsc{Handle}]$$

Figure 5: Kinding, Equivalence, and Typing Rules of $\lambda^l_{\mathtt{eff}}$(extensions)

Kind	κ	::=	$\ldots$	
		|	$\mathbf{L}$	(label type)
Type	σ, τ	::=	$\cdots$	
		|	ℓ	(label)
Effect	ϵ	::=	$(\Delta'.\ \tau/\rho)_l$	
Label	ℓ	::=	l	(static label)
Expression	e	::=	$\ldots$	
		|	$\mathbf{shift}_0\langle l\rangle\ k.\ e$	(shift$_0$)
		|	$\langle e \mid x.\ e\rangle_l$	(dollar)

Evaluation Context

$$E\ ::=\ \cdots \mid \langle E \mid x.\ e\rangle_l$$

Reduction Rules

$$\frac{}{\langle v \mid x.\ e_r\rangle_l \mapsto e_r\{v/x\}}\,[\textsc{EReturn}]$$

$$\frac{l \notin \lceil E\rceil \qquad v_c = \lambda z.\langle E[z] \mid x.\ e_r\rangle_l}{\langle E[\mathbf{shift}_0\langle l\rangle\ k.\ e] \mid x.\ e_r\rangle_l \mapsto e\{v_c/x\}}\,[\textsc{EDollar}]$$

Figure 6: Syntax and Semantics of $\lambda^l_{\mathtt{del}}$(extensions)

row ρ (which may refer to type variables in Δ'), and it must be captured by a shift operator indexed by l. As in $\lambda^l_{\mathtt{eff}}$, labels are chosen from a set Σ of statically defined labels. Expressions are extended with shift operators and dollar operators. A shift operator $\mathbf{shift}_0\langle l\rangle\ k.\ e$ captures the continuation k up to the closest dollar operator with prompt tag l and executes the expression e. A dollar operator $\langle e \mid x.\ e_r\rangle_l$ delimits the continuation to be captured in e and executes the return clause $x.e_r$ when e has reduced to a value.

4.2.2 Kinding, Equivalence, and Typing Rules. We define the kinding, equivalence, and typing rules of $\lambda^l_{\mathtt{del}}$ in Figure 7. The KLabel rule is identical to the $\lambda^l_{\mathtt{eff}}$. The KMEff assigns kind $\mathbf{E}$ to effects. The EMEff rule determines whether two effects are equivalent or not. The Shift$_0$ rule introduces an effect $(\Delta'.\ \tau/\rho)_l$ indexed by the label l. The effect row ρ' represents the effects of the body of a shift operator and has to be subsumed by the effect row of the continuation k because a shift operator reduces to the body of itself. The Dollar rule discharges an effect indexed by the label l. It requires that the body of a dollar operator introduces the effect $(\Delta'.\ \tau/\rho)_l$ and that the tail of an effect row ρ is instantiated by type variable substitution δ. The substitution δ is used to instantiate the type of the continuation of a surrounded shift operator as the type of the continuation of a shift operator is generalized by type context Δ.

4.2.3 Operational Semantics. We define the operational semantics of $\lambda^l_{\mathtt{del}}$ in Figure 6. The EReturn rule states that a dollar operator reduces to the return clause $e_r\{v/x\}$ if the body of the dollar operator is a value. The EDollar rule states that the body of the shift operator $\mathbf{shift}_0\langle l\rangle\ k.\ e$ is executed and passed the continuation delimited up to the innermost dollar operator with the same label. As in the EHandler rule in $\lambda^l_{\mathtt{eff}}$, the premise $l \notin \lceil E\rceil$ states that there is no dollar operator indexed by the label l in the evaluation context E. Here, $\lceil\cdot\rceil$ is again a meta-level function for extracting labels, which is defined as follows.

Definition 3 (Label extraction)**.**

$\lceil\square\rceil\ ::=\ \emptyset \quad \lceil E\ e\rceil\ ::=\ \lceil E\rceil \quad \lceil v\ E\rceil\ ::=\ \lceil E\rceil \quad \lceil\langle E \mid x.\ e\rangle_l\rceil ::= l, \lceil E\rceil$
$\lceil\iota\rceil\ ::=\ \bullet, \quad \lceil\epsilon \cdot \rho\rceil\ ::=\ \lceil\epsilon\rceil, \lceil\rho\rceil \quad \lceil(\Delta'.\ \tau/\rho)_\ell\rceil\ ::=\ \ell$

4.2.4 Type Soundness. We prove the type soundness of $\lambda^l_{\mathtt{del}}$ in a similar way to that of $\lambda^l_{\mathtt{eff}}$. As in $\lambda^l_{\mathtt{eff}}$, the theorems can be proved by induction on the typing derivation.

Theorem 3 (Preservation of $\lambda^l_{\mathtt{del}}$)**.**
If $\Delta \mid \Gamma \mid \Sigma \vdash e : \tau/\iota$ and $e \longrightarrow e'$ then $\Delta \mid \Gamma \mid \Sigma \vdash e' : \tau/\iota$.

Theorem 4 (Progress of $\lambda^l_{\mathtt{del}}$)**.**
If $\emptyset \mid \emptyset \mid \Sigma \vdash e : \tau/\iota$ then e is a value or there exists e' such that $e \longrightarrow e'$.

4.3 Translation Between Static Effect Instances and Prompt Tags

4.3.1 Translation from Static Prompt Tags to Effect Instances. In Figure 8, we define $[\![\cdot]\!]^l$, a macro translation from $\lambda^l_{\mathtt{del}}$ to $\lambda^l_{\mathtt{eff}}$. The translation is a straightforward extension of the corresponding translation by Piróg et al. [19].

$$\boxed{\Delta \mid \Sigma \vdash \tau :: \kappa}$$

$$\frac{l \in \Sigma}{\Delta \mid \Sigma \vdash l :: \mathbf{L}}\ [\text{KLabel}] \qquad \frac{\Delta, \Delta' \mid \Sigma \vdash \tau :: \mathbf{T} \qquad \Delta, \Delta' \mid \Sigma \vdash \rho :: \mathbf{R} \qquad \Delta \mid \Sigma \vdash l :: \mathbf{L}}{\Delta \mid \Sigma \vdash (\Delta'.\ \tau/\rho)_l :: \mathbf{E}}\ [\text{KMEff}]$$

$$\boxed{\Delta \mid \Sigma \vdash \rho \equiv \rho'}$$

$$\frac{\Delta, \Delta' \mid \Sigma \vdash \tau \equiv \tau' \qquad \Delta, \Delta' \mid \Sigma \vdash \rho \equiv \rho' \qquad \Delta \mid \Sigma \vdash l :: \mathbf{L}}{\Delta \mid \Sigma \vdash (\Delta'.\ \tau/\rho)_l \equiv (\Delta'.\ \tau'/\rho')_l}\ [\text{EMEff}]$$

$$\boxed{\Delta \mid \Gamma \mid \Sigma \vdash e : \sigma/\rho}$$

$$\frac{\begin{array}{c}\Delta \mid \Sigma \vdash \tau' :: \mathbf{T} \quad \Delta \mid \Sigma \vdash l :: \mathbf{L} \quad \Delta, \Delta' \mid \Sigma \vdash \rho' <: \rho \\ \Delta, \Delta' \mid \Gamma, k : \tau' \to_\rho \tau \mid \Sigma \vdash e : \tau/\rho' \quad \Delta \mid \Sigma \vdash ((\Delta'.\ \tau/\rho)_l) \cdot \rho' :: \mathbf{R}\end{array}}{\Delta \mid \Gamma \mid \Sigma \vdash \mathbf{shift}_0\langle l\rangle\ k.\ e : \tau'/(\Delta'.\ \tau/\rho)_l \cdot \rho'}\ [\text{Shift}_0] \qquad \frac{\begin{array}{c}\Delta \mid \Sigma \vdash \delta :: \Delta' \\ \Delta \mid \Gamma, x : \tau' \mid \Sigma \vdash e_r : \delta(\tau)/\delta(\rho) \\ \Delta \mid \Gamma \mid \Sigma \vdash e : \tau'/(\Delta'.\ \tau/\rho)_l \cdot \delta(\rho)\end{array}}{\Delta \mid \Gamma \mid \Sigma \vdash \langle e \mid x.\ e_r\rangle_l : \delta(\tau)/\delta(\rho)}\ [\text{Dollar}]$$

Figure 7: Kinding, Equivalence, Typing Rules of $\lambda^l_{\mathtt{del}}$(extensions)

$$\llbracket \mathbf{shift}_0\langle l\rangle\ k.\ e \rrbracket^l = \mathbf{do}\langle l\rangle\ \lambda k.\llbracket e \rrbracket^l$$

$$\llbracket \langle e \mid x.\ e_r\rangle_l \rrbracket^l = \mathbf{handle}\langle l\rangle\ \llbracket e \rrbracket^l\ \mathbf{with}\ \{x, r.x\ r; x.\llbracket e_r \rrbracket^l\}$$

$$\llbracket (\Delta'.\ \tau/\rho)_l \rrbracket^l$$
$$= (\alpha :: \mathbf{T}.(\forall \Delta'.(\alpha \to_{\llbracket \rho \rrbracket^l} \llbracket \tau \rrbracket^l) \to_{\llbracket \rho \rrbracket^l} \llbracket \tau \rrbracket^l) \Rightarrow \alpha)_l$$

$$\llfloor \mathbf{do}\langle l\rangle\ v \rrfloor^l = \mathbf{shift}_0\langle l\rangle\ k.\ \lambda h.h\ \llfloor v \rrfloor^l\ (\lambda x.k\ x\ h)$$

$$\llfloor \mathbf{handle}\langle l\rangle\ e\ \mathbf{with}\ \{x, r.e_h; x.e_r\} \rrfloor^l = \langle \llfloor e \rrfloor^l \mid x.\ \lambda h.\llfloor e_r \rrfloor^l\rangle_l\ \lambda x.\lambda r.\llfloor e_h \rrfloor^l$$

$$\llfloor (\Delta'.\tau_1 \Rightarrow \tau_2)_l \rrfloor^l$$
$$= (\alpha :: \mathbf{T}, \beta :: \mathbf{R}.\ ((\forall \Delta'.\llfloor \tau_1 \rrfloor^l \to_\iota (\llfloor \tau_2 \rrfloor^l \to_\beta \alpha) \to_\beta \alpha) \to_\beta \alpha)/\beta)_l$$

Figure 8: Macro Translations Between $\lambda^l_{\mathtt{eff}}$and $\lambda^l_{\mathtt{del}}$

Let us first look at the translation of expressions. A shift operator indexed by the label l is translated to an operation indexed by l. This should intuitively make sense, as the shift operator and an operation call play a similar role, namely to introduce an effect. Observe that the resulting operation receives a function that takes a continuation, which is translated from the body of the shift operator. A dollar operator indexed by the label l is translated to a handler expression indexed by l. Notice that the operation clause $x, r.x\ r$ passes the continuation r to the operation's argument x. And then, the body of a shift operator receives the continuation through the operation clause of the handler. The return clause of a dollar simply becomes the return clause of a handler. Other expressions are simply translated recursively.

We next look at the translation for effect types. An effect type indexed by the label l of $\lambda^l_{\mathtt{del}}$ is translated to an effect type indexed by the same label l of $\lambda^l_{\mathtt{eff}}$. Here, the type $(\alpha \to_{\llbracket \rho \rrbracket^l} \llbracket \tau \rrbracket^l)$ represents the type of the continuation k, where α is the type of the shift expression. Other types are translated recursively.

The translation enjoys two desired properties: type preservation and meaning preservation. The former states that a well-typed expression is translated into a well-typed expression. The latter states that two expressions related by one-step reduction are translated to two expressions related by multi-step reduction.

Theorem 5 (Translation preserves types)**.**
If $\Delta \mid \Gamma \mid \Sigma \vdash e : \tau/\rho$ then $\Delta \mid \llbracket \Gamma \rrbracket^l \mid \Sigma \vdash \llbracket e \rrbracket^l : \llbracket \tau \rrbracket^l/\llbracket \rho \rrbracket^l$.

Theorem 6 (Translation preserves meanings)**.**
If $e \to e'$ then $\llbracket e \rrbracket^l \to^+ \llbracket e' \rrbracket^l$.

4.3.2 Translation from Static Effect Instances to Prompt Tags. In Figure 8, we define $\llfloor \cdot \rrfloor^l$, a macro translation from $\lambda^l_{\mathtt{eff}}$ to $\lambda^l_{\mathtt{del}}$. The translation is again an extension of the corresponding translation by Piróg et al.

We begin with the translation of expressions. An operation indexed by the label l is translated to a shift operator indexed by l. Here, the body of the shift operator is a function that takes in a handler, reflecting the fact that the meaning of an operation is determined externally by a handler. The expression $\lambda x.k\ x\ h$ represents the delimited continuation of the operation call, including the surrounding handler h. A handler expression indexed by the label l is translated to the application of a dollar operator and an operation clause. The application supplies the handler required by the body of the shift operator. The handled expression and return clause of a handler simply become the body and return clause of a dollar operator, respectively. Other expressions are translated recursively.

We next move on to the translation for effect types. An effect type indexed by the label l of $\lambda^l_{\mathtt{eff}}$ is translated to an effect type indexed by l of $\lambda^l_{\mathtt{del}}$. The types $\llfloor \tau_1 \rrfloor^l$ and $\llfloor \tau_2 \rrfloor^l \to_\beta \alpha$ correspond to the types of $\llfloor v \rrfloor^l$ and $\lambda x.k\ x\ h$, respectively. While the effect row of a continuation in $\lambda^l_{\mathtt{eff}}$ is determined by the surrounding handler, the effect row of a continuation in $\lambda^l_{\mathtt{del}}$ is determined by a shift operator. Thus, we generalize the effect row of the effect type by the type variable β in order to preserve typability.

The translation enjoys two desired properties: type preservation and meaning preservation.

Theorem 7 (Translation preserves types)**.**
If $\Delta \mid \Gamma \mid \Sigma \vdash e : \tau/\rho$ then $\Delta \mid \llfloor \Gamma \rrfloor^l \mid \Sigma \vdash \llfloor e \rrfloor^l : \llfloor \tau \rrfloor^l/\llfloor \rho \rrfloor^l$.

Theorem 8 (Translation preserves meaning)**.**
If $e \longrightarrow e'$ then $\llbracket e \rrbracket^{l} \longrightarrow_i^{+} \llbracket e' \rrbracket^{l}$.

The first theorem is almost identical to the corresponding theorem for the opposite translation (Theorem 5), but the second theorem uses a more generous notion of reduction (GStep) that allows us to reduce expressions in any context. The definition of the GStep relation is as follows.

$$\begin{array}{llll} \text{General Context} & C & ::= & \square \mid C\ e \mid e\ C \mid \lambda x.C \\ & & \mid & \langle C \mid x.\ e_r \rangle_l \mid \langle e \mid x.\ C \rangle_l \\ & & \mid & \mathbf{shift}_0 \langle \ell \rangle\ k.\ C \end{array}$$

$$\frac{e_1 \mapsto e_2}{C[e_1] \longrightarrow_i C[e_2]}\ [\text{GStep}]$$

Figure 9: General Context and General Step

Note that the difficulty with meaning preservation of the translation from effect handlers to control operators arises in the unlabeled setting as well [19].

5 GENERATIVE EFFECT INSTANCES AND PROMPT TAGS

In this section, we present two extensions of $\lambda^{l}_{\mathtt{eff}}$ and $\lambda^{l}_{\mathtt{del}}$, and a pair of macro translations between the two extended calculi. One of the extended calculi is called $\lambda^{l+\eta}_{\mathtt{eff}}$, which features algebraic effect handlers and generative effect instances. The other calculus is called $\lambda^{l+\eta}_{\mathtt{del}}$, which features delimited control operators and generative prompt tags. Here, we use the term "generative labels" to mean labels that are generated during evaluation by effect handlers or control delimiters, similar to effect instances in the calculus of Biernacki et al. [2].

5.1 $\lambda^{l+\eta}_{\mathtt{eff}}$: Algebraic Effect Handlers with Generative Effect Instances

5.1.1 Syntax. We define the syntax of $\lambda^{l+\eta}_{\mathtt{eff}}$ in Figure 10. For brevity, we only show the changes to $\lambda^{l}_{\mathtt{eff}}$. Effect labels are extended with type variables that have kind **L**. Expressions are extended with label abstractions $\Lambda\eta.e$, label applications $e[\ell]$ and labeled handlers $\mathbf{handle}\langle\eta\rangle\ e\ \mathbf{with}\ \{x, r.e_h; x.e_r\}$. A label application $e[\ell]$ applies the expression e to the label ℓ. A labeled handler $\mathbf{handle}\langle\eta\rangle\ e$ $\mathbf{with}\ \{x, r.e_h; x.e_r\}$ introduces a label η that can be used in the handled computation e. Values are extended with label abstractions $\Lambda\eta.e$. A label abstraction generalizes the expression e over the type variable η of kind **L**.

5.1.2 Kinding, Equivalence, and Typing Rules. We define the kinding, equivalence, and typing rules of $\lambda^{l+\eta}_{\mathtt{eff}}$ in Figure 11. The KLIEff, ELIEff, LDo rules are almost the same as the KIEff, EIEff, and Do rules, respectively, but derive a conclusion that includes an effect indexed by type variable η. The LAbs rule generalizes the expression e over the type variable η. The LApp rule applies the expression e to the label ℓ. The LHandle rule introduces the label

$$\begin{array}{llll} \text{Label} & \ell & ::= & \dots \\ & & \mid & \eta \qquad \text{(generative label)} \\ \text{Expression} & e & ::= & \dots \\ & & \mid & e[\ell] \qquad \text{(label application)} \\ & & \mid & \mathbf{handle}\langle\eta\rangle\ e \\ & & & \quad \mathbf{with}\ \{x, r.e_h; x.e_r\} \qquad \text{(labeled handler)} \\ \text{Value} & v & ::= & \dots \\ & & \mid & \Lambda\eta.e \qquad \text{(label abstraction)} \end{array}$$

Evaluation Context

$$E\ ::=\ \cdots \mid E[\ell]$$

Reduction Rules

$$\frac{}{\Sigma \vdash\ (\Lambda\eta.e)[\ell] \mapsto e\{\ell/\eta\}\ \dashv \Sigma}\ [\text{ELApp}]$$

$$\frac{l \text{ is fresh} \qquad \Sigma' = \Sigma, l \qquad \delta = \{l/\eta\}}{\begin{array}{l}\Sigma \vdash\ \mathbf{handle}\langle\eta\rangle\ e\ \mathbf{with}\ \{x, r.e_h; x.e_r\} \\ \quad \mapsto \mathbf{handle}\langle l\rangle\ \delta(e)\ \mathbf{with}\ \{x, r.e_h; x.e_r\}\ \dashv \Sigma'\end{array}}\ [\text{EGenLabel}]$$

Figure 10: Syntax and Semantics of $\lambda^{l+\eta}_{\mathtt{eff}}$(extensions)

η into the handled expression e and discharges an effect indexed by the label η. The label η is lexically scoped, hence the operation clause and return clause are typed under the type variable environment Δ that does not have the label $\eta :: \mathbf{L}$. This is necessary for preventing names from escaping their scope. For instance, the expression $\mathbf{handle}\langle\eta\rangle\ \mathbf{do}\langle\eta\rangle\ ()\ \mathbf{with}\ \{x, r.r\ x; x.\lambda x.\mathbf{do}\langle\eta\rangle\ x\}$ must be judged ill-typed because the operation call of the return clause does not have a corresponding handler that has the label η.

5.1.3 Operational Semantics. We define the operational semantics of $\lambda^{l+\eta}_{\mathtt{eff}}$ in Figure 10. We use a reduction judgment of the form $\Sigma \vdash e \mapsto e' \dashv \Sigma'$. The judgment states that under the label environment Σ, expression e reduces to e' while yielding a new label environment Σ'. The semantics is based on Biernacki et al. [2], but it is extended with a label environment that is needed to state the preservation theorem.

The most interesting rule is EGenLabel. The rule states that the type variable η is replaced by the label l that is dynamically generated at runtime and the label environment is extended with the label l. Note that the substitution for η does not apply to the operation clause and return clause as they do not involve η.

5.1.4 Type Soundness. We prove type soundness of $\lambda^{l+\eta}_{\mathtt{eff}}$ by showing the progress and preservation theorems. Both theorems can be proved by induction on the typing derivation.

Theorem 9 (Preservation of $\lambda^{l+\eta}_{\mathtt{eff}}$)**.**
If $\Delta \mid \Gamma \mid \Sigma \vdash e : \tau/\iota$ and $\Sigma \vdash e \longrightarrow e' \dashv \Sigma'$ then $\Delta \mid \Gamma \mid \Sigma' \vdash e' : \tau/\iota$.

Theorem 10 (Progress of $\lambda^{l+\eta}_{\mathtt{eff}}$)**.**
If $\emptyset \mid \emptyset \mid \Sigma \vdash e : \tau/\iota$ then e is a value or there exists e' such that $\Sigma \vdash e \longrightarrow e' \dashv \Sigma'$.

$\boxed{\Delta \mid \Sigma \vdash \tau :: \kappa}$ $\boxed{\Delta \mid \Sigma \vdash \rho \equiv \rho'}$

$$\frac{\Delta, \Delta' \mid \Sigma \vdash \tau_1 :: \mathbf{T} \quad \Delta, \Delta' \mid \Sigma \vdash \tau_2 :: \mathbf{T} \quad \Delta \mid \Sigma \vdash \eta :: \mathbf{L}}{\Delta \mid \Sigma \vdash (\Delta'.\tau_1 \Rightarrow \tau_2)_\eta :: \mathbf{E}} \text{[KLIEff]} \qquad \frac{\Delta, \Delta' \mid \Sigma \vdash \tau_1 \equiv \tau_1' \quad \Delta, \Delta' \mid \Sigma \vdash \tau_2 \equiv \tau_2' \quad \Delta \mid \Sigma \vdash \eta :: \mathbf{L}}{\Delta \mid \Sigma \vdash (\Delta'.\tau_1 \Rightarrow \tau_2)_\eta \equiv (\Delta'.\tau_1' \Rightarrow \tau_2')_\eta} \text{[ELIEff]}$$

$\boxed{\Delta \mid \Gamma \mid \Sigma \vdash e : \tau / \rho}$

$$\frac{\Delta, \eta :: \mathbf{L} \mid \Gamma \mid \Sigma \vdash e : \tau/\iota}{\Delta \mid \Gamma \mid \Sigma \vdash \Lambda\eta.e : \forall \eta :: \mathbf{L}.\tau/\iota} \text{[LAbs]} \qquad \frac{\Delta \mid \Gamma \mid \Sigma \vdash e : \forall \eta :: \mathbf{L}.\tau/\iota \qquad \Delta \mid \Sigma \vdash \ell :: \mathbf{L}}{\Delta \mid \Gamma \mid \Sigma \vdash e[\ell] : \tau\{\ell/\eta\}/\iota} \text{[LApp]}$$

$$\frac{\begin{array}{c}\Delta \mid \Sigma \vdash (\Delta'.\tau_1 \Rightarrow \tau_2)_\eta :: \mathbf{E} \\ \Delta \mid \Sigma \vdash \delta :: \Delta' \quad \Delta \mid \Gamma \mid \Sigma \vdash v : \delta(\tau_1)/\iota\end{array}}{\Delta \mid \Gamma \mid \Sigma \vdash \mathbf{do}\langle\eta\rangle\ v : \delta(\tau_2)/(\Delta'.\tau_1 \Rightarrow \tau_2)_\eta \cdot \iota} \text{[LDo]} \qquad \frac{\begin{array}{c}\Delta \mid \Gamma, x : \tau \mid \Sigma \vdash e_r : \tau_r/\rho \\ \Delta, \Delta' \mid \Gamma, x : \tau_1, r : \tau_2 \to_\rho \tau_r \mid \Sigma \vdash e_h : \tau_r/\rho \\ \Delta, \eta :: \mathbf{L} \mid \Gamma \mid \Sigma \vdash e : \tau/(\Delta'.\tau_1 \Rightarrow \tau_2)_\eta \cdot \rho\end{array}}{\Delta \mid \Gamma \mid \Sigma \vdash \mathbf{handle}\langle\eta\rangle\ e\ \mathbf{with}\ \{x, r.e_h; x.e_r\} : \tau_r/\rho} \text{[LHandle]}$$

Figure 11: Typing Rules of $\lambda^{l+\eta}_{\texttt{eff}}$(extensions)

5.2 $\lambda^{l+\eta}_{\texttt{del}}$: Delimited Control Operators with Generative Prompt Tags

Label	ℓ	::=	$\cdots$	
		$\mid$	η	(generative label)
Expression	e	::=	$\ldots$	
		$\mid$	$e[\ell]$	(label application)
		$\mid$	$\langle e \mid x.\ e_r\rangle_\eta$	(labeled dollar)
Value	v	::=	$\ldots$	
		$\mid$	$\Lambda\eta.e$	(label abstraction)

Evaluation Context

$$E\ ::=\ \cdots \mid E[\ell]$$

General Context

$$C\ ::=\ \cdots \mid C[\ell] \mid \Lambda\eta.C \mid \langle C \mid x.\ e_r\rangle_\eta \mid \langle e \mid x.\ C\rangle_\eta$$

Reduction Rules

$$\frac{}{\Sigma \vdash\ (\Lambda\eta.e)[\ell] \mapsto e\{\ell/\eta\}\ \dashv \Sigma} \text{[ELApp]}$$

$$\frac{l \text{ is fresh} \qquad \Sigma' = \Sigma, l \qquad \delta = \{l/\eta\}}{\Sigma \vdash\ \langle e \mid x.\ e_r\rangle_\eta \mapsto \langle\delta(e) \mid x.\ e_r\rangle_l\ \dashv \Sigma'} \text{[EGenLabel]}$$

Figure 12: Syntax and Semantics of $\lambda^{l+\eta}_{\texttt{del}}$(extensions)

5.2.1 Syntax. We define the syntax of $\lambda^{l+\eta}_{\texttt{del}}$ in Figure 12. For brevity, we only show the changes to $\lambda^{l}_{\texttt{del}}$. Effect labels, label abstractions, and label applications are completely identical to $\lambda^{l}_{\texttt{del}}$. A labeled dollar operator $\langle e \mid x.\ e_r\rangle_\eta$ delimits the continuation to be captured in e and executes the return clause $x.\ e_r$ when e has reduced to a value.

5.2.2 Kinding, Equivalence, Typing Rules. We define the kinding, equivalence, and typing rules of $\lambda^{l+\eta}_{\texttt{del}}$ in Figure 13. The KLMEff, ELMEff, LShift$_0$ rules are almost the same as the KMEff, EMEff, Shift$_0$ rules, respectively, but derive a conclusion that includes an effect indexed by type variable η. The LAbs and LApp rules are completely identical to $\lambda^{l+\eta}_{\texttt{eff}}$. The LDollar rule introduces the label η into the expression e and discharges an effect indexed by the label η. As in $\lambda^{l+\eta}_{\texttt{eff}}$, the label η is lexically scoped, hence the return clause of a dollar operator is typed under the type variable environment Δ (without η), which is necessary for preventing name escaping.

5.2.3 Operational Semantics. We define the operational semantics of $\lambda^{l+\eta}_{\texttt{eff}}$ in Figure 12. We use a reduction judgment of the form $\Sigma \vdash e \mapsto e' \dashv \Sigma'$ as in $\lambda^{l+\eta}_{\texttt{eff}}$.

The most interesting rule is EGenLabel. The rule states that the type variable η is replaced by the label l that is dynamically generated at runtime and the label environment is extended with the label l. Note that the substitution for η does not apply to the return clause as it does not involve η.

5.2.4 Type Soundness. We prove type soundness of $\lambda^{l+\eta}_{\texttt{del}}$ in a similar way to that of $\lambda^{l+\eta}_{\texttt{eff}}$. As in $\lambda^{l+\eta}_{\texttt{eff}}$, the theorems can be proved by induction on the typing derivation.

Theorem 11 (Preservation of $\lambda^{l+\eta}_{\texttt{del}}$).
If $\Delta \mid \Gamma \mid \Sigma \vdash e : \tau/\iota$ and $\Sigma \vdash e \longrightarrow e' \dashv \Sigma'$ then $\Delta \mid \Gamma \mid \Sigma' \vdash e' : \tau/\iota$.

Theorem 12 (Progress of $\lambda^{l+\eta}_{\texttt{del}}$).
If $\emptyset \mid \emptyset \mid \Sigma \vdash e : \tau/\iota$ then e is a value or there exists e' such that $\Sigma \vdash e \longrightarrow e' \dashv \Sigma'$.

5.3 Translation Between Generative Effect Instances and Prompt Tags

5.3.1 Translation from Generative Prompt Tags to Effect Instances. In Figure 14, we define $[\![\cdot]\!]^{l+\eta}$, a macro translation from $\lambda^{l+\eta}_{\texttt{del}}$ to $\lambda^{l+\eta}_{\texttt{eff}}$. The macro translations of label abstractions and applications are identity. The macro translation of a dollar operator with generative labels is almost the same as that of a dollar operator with static labels. The only difference from the macro translation $[\![\cdot]\!]^{l}$ for static labels is that a labeled dollar operator is translated to a handler expression that is labeled with a type variable η instead of a label l. The translation of effects indexed by η is almost the same as effects indexed by l.

$$\boxed{\Delta \mid \Sigma \vdash \tau :: \kappa}$$

$$\frac{\Delta, \Delta' \mid \Sigma \vdash \tau :: \mathbf{T} \quad \Delta, \Delta' \mid \Sigma \vdash \rho :: \mathbf{R} \quad \Delta \mid \Sigma \vdash \eta :: \mathbf{L}}{\Delta \mid \Sigma \vdash (\Delta'.\, \tau/\rho)_\eta :: \mathbf{E}} \,[\text{KLME}\textsc{ff}]$$

$$\boxed{\Delta \mid \Sigma \vdash \rho \equiv \rho'}$$

$$\frac{\Delta, \Delta' \mid \Sigma \vdash \tau \equiv \tau' \quad \Delta, \Delta' \mid \Sigma \vdash \rho \equiv \rho' \quad \Delta \mid \Sigma \vdash \eta :: \mathbf{L}}{\Delta \mid \Sigma \vdash (\Delta'.\, \tau/\rho)_\eta \equiv (\Delta'.\, \tau'/\rho')_\eta} \,[\text{ELME}\textsc{ff}]$$

$$\boxed{\Delta \mid \Gamma \mid \Sigma \vdash e : \sigma/\rho}$$

$$\frac{\Delta, \eta :: \mathbf{L} \mid \Gamma \mid \Sigma \vdash e : \tau/\iota}{\Delta \mid \Gamma \mid \Sigma \vdash \Lambda\eta.e : \forall \eta :: \mathbf{L}.\tau/\iota} \,[\text{L}\textsc{Abs}] \qquad \frac{\Delta \mid \Gamma \mid \Sigma \vdash e : \forall \eta :: \mathbf{L}.\tau/\iota \qquad \Delta \mid \Sigma \vdash \ell :: \mathbf{L}}{\Delta \mid \Gamma \mid \Sigma \vdash e[\ell] : \tau\{\ell/\eta\}/\iota} \,[\text{L}\textsc{App}]$$

$$\frac{\begin{array}{c}\Delta \mid \Sigma \vdash \tau' :: \mathbf{T} \quad \Delta \mid \Sigma \vdash \eta :: \mathbf{L} \quad \Delta, \Delta' \mid \Sigma \vdash \rho' <: \rho \\ \Delta, \Delta' \mid \Gamma, k : \tau' \to_\rho \tau \mid \Sigma \vdash e : \tau/\rho' \quad \Delta \mid \Sigma \vdash ((\Delta'.\, \tau/\rho)_\eta) \cdot \rho' :: \mathbf{R}\end{array}}{\Delta \mid \Gamma \mid \Sigma \vdash \mathbf{shift}_0\langle\eta\rangle\; k.\, e : \tau'/(\Delta'.\, \tau/\rho)_\eta \cdot \rho'} \,[\text{L}\textsc{Shift}_0]$$

$$\frac{\begin{array}{c}\Delta \mid \Sigma \vdash \delta :: \Delta' \\ \Delta \mid \Gamma, x : \tau' \mid \Sigma \vdash e_r : \delta(\tau)/\delta(\rho) \\ \Delta, \eta :: \mathbf{L} \mid \Gamma \mid \Sigma \vdash e : \tau'/(\Delta'.\, \tau/\rho)_\eta \cdot \delta(\rho)\end{array}}{\Delta \mid \Gamma \mid \Sigma \vdash \langle e \mid x.\, e_r\rangle_\eta : \delta(\tau)/\delta(\rho)} \,[\text{L}\textsc{Dollar}]$$

Figure 13: Typing Rules of $\lambda^{l+\eta}_{\mathtt{del}}$(extensions)

$$\llbracket \Lambda\eta.e \rrbracket^{l+\eta} = \Lambda\eta.\llbracket e \rrbracket^{l+\eta}$$

$$\llbracket e\;[\ell] \rrbracket^{l+\eta} = \llbracket e \rrbracket^{l+\eta}\;[\ell]$$

$$\llbracket \langle e \mid x.\, e_r\rangle_\eta \rrbracket^{l+\eta} = \mathbf{handle}\langle\eta\rangle\; \llbracket e \rrbracket^{l+\eta}\; \mathbf{with}\; \{x, r.x\; r; x.\llbracket e_r \rrbracket^{l+\eta}\}$$

$$\llbracket (\Delta'.\, \tau/\rho)_\eta \rrbracket^{l+\eta}$$
$$= (\alpha :: \mathbf{T}.(\forall\Delta'.(\alpha \to_{\llbracket\rho\rrbracket^{l+\eta}} \llbracket\tau\rrbracket^{l+\eta}) \to_{\llbracket\rho\rrbracket^{l+\eta}} \llbracket\tau\rrbracket^{l+\eta}) \Rightarrow \alpha)_\eta$$

$$\lfloor\!\lfloor \Lambda\eta.e \rfloor\!\rfloor^{l+\eta} = \Lambda\eta.\lfloor\!\lfloor e \rfloor\!\rfloor^{l+\eta}$$

$$\lfloor\!\lfloor e\;[\ell] \rfloor\!\rfloor^{l+\eta} = \lfloor\!\lfloor e \rfloor\!\rfloor^{l+\eta}\;[\ell]$$

$$\lfloor\!\lfloor \mathbf{handle}\langle\eta\rangle\; e\; \mathbf{with}\; \{x, r.e_h; x.e_r\} \rfloor\!\rfloor^{l+\eta}$$
$$= \langle \lfloor\!\lfloor e \rfloor\!\rfloor^{l+\eta} \mid x.\, \lambda h.\lfloor\!\lfloor e_r \rfloor\!\rfloor^{l+\eta}\rangle_\eta\; \lambda x.\lambda r.\lfloor\!\lfloor e_h \rfloor\!\rfloor^{l+\eta}$$

$$\lfloor\!\lfloor (\Delta'.\tau_1 \Rightarrow \tau_2)_\eta \rfloor\!\rfloor^{l+\eta}$$
$$= (\alpha :: \mathbf{T}, \beta :: \mathbf{R}.$$
$$((\forall\Delta'.\lfloor\!\lfloor\tau_1\rfloor\!\rfloor^{l+\eta} \to_\iota (\lfloor\!\lfloor\tau_2\rfloor\!\rfloor^{l+\eta} \to_\beta \alpha) \to_\beta \alpha) \to_\beta \alpha)/\beta)_\eta$$

Figure 14: Macro Translations Between $\lambda^{l+\eta}_{\mathtt{eff}}$and $\lambda^{l+\eta}_{\mathtt{del}}$

The translation preserves types and meanings. The first theorem is essentially the same as Theorem 5. The reduction judgment of the second theorem is extended with label environment Σ.

Theorem 13 (Translation preserves types)**.**
If $\Delta \mid \Gamma \mid \Sigma \vdash e : \tau/\rho$
then $\Delta \mid \llbracket\Gamma\rrbracket^{l+\eta} \mid \Sigma \vdash \llbracket e\rrbracket^{l+\eta} : \llbracket\tau\rrbracket^{l+\eta}/\llbracket\rho\rrbracket^{l+\eta}$.

Theorem 14 (Translation preserves meanings)**.**
If $\Sigma \vdash e \longrightarrow e' \dashv \Sigma'$ then $\Sigma \vdash \llbracket e\rrbracket^{l+\eta} \longrightarrow^+ \llbracket e'\rrbracket^{l+\eta} \dashv \Sigma'$.

5.3.2 Translation from Generative Effect Instances to Prompt Tags. In Figure 14, we define $\lfloor\!\lfloor\cdot\rfloor\!\rfloor^{l+\eta}$, a macro translation from $\lambda^{l+\eta}_{\mathtt{eff}}$ to $\lambda^{l+\eta}_{\mathtt{del}}$. The macro translations of label abstractions and applications are completely identical to the $\llbracket\cdot\rrbracket^{l+\eta}$. The macro translation of handlers with generative labels is almost the same as that of handlers with static labels. The only difference from the macro translation $\lfloor\!\lfloor\cdot\rfloor\!\rfloor^{l}$ for static labels is that a labeled handler is translated to a dollar operator that is labeled with a type variable η instead of a label l. The translation of effects indexed by η is almost the same as effects indexed by l.

The translation preserves types and meanings. The first theorem is almost identical to Theorem 7. The reduction judgment of the second theorem is extended with effect label environment Σ.

Theorem 15 (Translation preserves types)**.**
If $\Delta \mid \Gamma \mid \Sigma \vdash e : \tau/\rho$
then $\Delta \mid \lfloor\!\lfloor\Gamma\rfloor\!\rfloor^{l+\eta} \mid \Sigma \vdash \lfloor\!\lfloor e\rfloor\!\rfloor^{l+\eta} : \lfloor\!\lfloor\tau\rfloor\!\rfloor^{l+\eta}/\lfloor\!\lfloor\rho\rfloor\!\rfloor^{l+\eta}$.

Theorem 16 (Translation preserves meaning)**.**
If $\Sigma \vdash e \longrightarrow e' \dashv \Sigma'$ then $\Sigma \vdash \lfloor\!\lfloor e\rfloor\!\rfloor^{l+\eta} \longrightarrow^+_i \lfloor\!\lfloor e'\rfloor\!\rfloor^{l+\eta} \dashv \Sigma'$.

6 RELATED WORK

In this section, we discuss related work on labeled variations of effect facilities and macro translations between effect facilities. For the former, we make a comparison to our calculi from three perspectives: (i) whether labels are first class or second class, (ii) whether labels are static or generative, and (iii) whether the type system is sound or unsound. The results are summarized in Table 1.

6.1 Effect Handlers with Effect Instances

Bauer et al. [1] formalize the core Eff language, which models an old version of the Eff language. They treat effect instances as first-class, static values. To obtain soundness of the type system, they define effect instances globally and thus make it impossible for instances to escape.

Biernacki et al. [2] develop a calculus with effect handlers and effect instances, which we build our work on. They treat effect instances as second-class, generative values, as we do in our calculi. To prevent escaping of effect instances, they enforce lexical scoping of instances, also like we do.

	Label		
	First or second class	**Static or generative**	**Type System**
Effect handlers			
$\lambda^{l}_{\mathtt{eff}}$ (This work)	Second class	Static	Sound
$\lambda^{l+\eta}_{\mathtt{eff}}$ (This work)	Second class	Generative	Sound
Bauer et al. [1]	First class	Static	Sound
Biernacki et al. [2]	Second class	Generative	Sound
Xie et al. [25]	First class	Generative	Sound
Control operators			
$\lambda^{l}_{\mathtt{del}}$ (This work)	Second class	Static	Sound
$\lambda^{l+\eta}_{\mathtt{del}}$ (This work)	Second class	Generative	Sound
Gunter et al. [11]	First class	Generative	Unsound
Kiselyov et al. [14]	Second class	Static	Unsound

Table 1: Comparison Among Labeled Effect Calculi

Xie et al. [25] design a calculus of named handlers, which are equivalent to labeled handlers. They treat handler names as first-class values, and allow them to be statically chosen or dynamically generated. A key novelty is that they prevent escaping of names via rank-2 polymorphism, which is a well-known technique available in, e.g., Haskell. We plan to incorporate their ideas to extend our work to first-class labels.

6.2 Delimited Control Operators with Prompt Tags

Gunter et al. [11] present a calculus with multi-prompt delimited control operators. In their calculus, prompt tags are first-class, generative values, but they are not tracked by the type system. Hence, a program may get stuck by executing a control operator that has no corresponding delimiter.

Kiselyov et al. [14] show a translation from dynamic binding to multi-prompt shift and reset operators. They treat prompt tags as second-class, static values, but do not track them in the type system. In the soundness statement of their calculus, they impose a strong assumption that the program does not have a shift operator with no corresponding reset operator.

6.3 Translations Between Delimited Control Operators and Algebraic Effect Handlers

Forster et al. [10] study the relationship between three calculi with different effect facilities: effect handlers, monads, and control operators. They show the equivalence of expressive power among these calculi by defining macro translations but unlike us, they do this in an untyped setting. They then conjecture that extending their translations to a typed setting would require certain forms of polymorphism.

Piróg et al. [19] partially prove Forster et al.'s conjecture by defining typed macro translations between two calculi with effect handlers and delimited control operators ($\mathtt{shift}_0$/`dollar`). They equip the calculi of effect handlers and $\mathtt{shift}_0$/`dollar` with effect polymorphism, which is the key to type preservation.

6.4 Implementation of Effect Instances Using Prompt Tags

Kislyov et al. [15] embed the Eff language with effect instances into OCaml using the delimcc delimited control operators library [13]. The library has first-class prompt tags that can be generated dynamically. However, prompt tags (and hence effect instances) can escape their scope, because OCaml does not have an effect system that tracks prompt tags.

7 CONCLUSION AND FUTURE WORK

In this paper, we studied the relationship between two facilities for expressing computational effects, namely effect handlers with effect instances and delimited control operators with prompt tags. We first formalized two calculi $\lambda^{l}_{\mathtt{eff}}$ and $\lambda^{l}_{\mathtt{del}}$, featuring static effect instances and prompt tags, and proved the equivalence of their expressibility via a pair of macro translations. We next did the same for calculi $\lambda^{l+\eta}_{\mathtt{eff}}$ and $\lambda^{l+\eta}_{\mathtt{del}}$, which feature effect instances and prompt tags that are dynamically generated at runtime.

Using the established relationship, we can understand and implement one of the two facilities via the other facility. For example, we can derive the CPS translation for effect instances by composing our macro translation and the existing CPS translation for multi-prompt control operators [5], just like Cong and Asai do [3]. As a different example, we can implement labeled handlers using labeled control operators, analogously to what Kammar et al. [12] do.

As future work, we intend to extend our calculi with first-class effect instances and prompt tags. The extension allows us to use effect instances and prompt tags in a more flexible way. Following Xie et al. [25], we avoid escaping of labels using rank-2 polymorphism, which we believe can be integrated into our calculi with no major challenges.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable comments, and the members of the Programming Research Group at Tokyo Institute of Technology for helpful discussions. This work was supported by JSPS KAKENHI grant numbers 18H03219, 19K24339, and 23H03368.

REFERENCES

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

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

[3] Youyou Cong and Kenichi Asai. 2022. Understanding Algebraic Effect Handlers via Delimited Control Operators. In *Trends in Functional Programming - 23rd International Symposium, TFP 2022, Virtual Event, March 17-18, 2022, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 13401)*, Wouter Swierstra and Nicolas Wu (Eds.). Springer, 59–79. https://doi.org/10.1007/978-3-031-21314-4_4

[4] Olivier Danvy and Andrzej Filinski. 1990. Abstracting Control. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming, LFP 1990, Nice, France, 27-29 June 1990*, Gilles Kahn (Ed.). ACM, 151–160. https://doi.org/10.1145/91556.91622

[5] Paul Downen and Zena M. Ariola. 2014. Delimited control and computational effects. *J. Funct. Program.* 24, 1 (2014), 1–55. https://doi.org/10.1017/S0956796813000312

[6] Matthias Felleisen. 1988. The Theory and Practice of First-Class Prompts. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, San Diego, California, USA, January 10-13, 1988*, Jeanne Ferrante and Peter Mager (Eds.). ACM Press, 180–190. https://doi.org/10.1145/73560.73576

[7] Matthias Felleisen. 1991. On the Expressive Power of Programming Languages. *Sci. Comput. Program.* 17, 1-3 (1991), 35–75. https://doi.org/10.1016/0167-6423(91)90036-W

[8] Andrzej Filinski. 1994. Representing Monads. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Portland, Oregon, USA) *(POPL '94)*. Association for Computing Machinery, New York, NY, USA, 446–457. https://doi.org/10.1145/174675.178047

[9] Matthew Flatt, Gang Yu, Robert Bruce Findler, and Matthias Felleisen. 2007. Adding delimited and composable control to a production programming environment. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming, ICFP 2007, Freiburg, Germany, October 1-3, 2007*, Ralf Hinze and Norman Ramsey (Eds.). ACM, 165–176. https://doi.org/10.1145/1291151.1291178

[10] Yannick Forster, Ohad Kammar, Sam Lindley, and Matija Pretnar. 2016. On the Expressive Power of User-Defined Effects: Effect Handlers, Monadic Reflection, Delimited Control. *CoRR* abs/1610.09161 (2016). arXiv:1610.09161 http://arxiv.org/abs/1610.09161

[11] Carl A. Gunter, Didier Rémy, and Jon G. Riecke. 1995. A Generalization of Exceptions and Control in ML-like Languages. In *Proceedings of the seventh international conference on Functional programming languages and computer architecture, FPCA 1995, La Jolla, California, USA, June 25-28, 1995*, John Williams (Ed.). ACM, 12–23. https://doi.org/10.1145/224164.224173

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

[13] Oleg Kiselyov. 2010. Delimited Control in OCaml, Abstractly and Concretely: System Description. In *Functional and Logic Programming, 10th International Symposium, FLOPS 2010, Sendai, Japan, April 19-21, 2010. Proceedings (Lecture Notes in Computer Science, Vol. 6009)*, Matthias Blume, Naoki Kobayashi, and Germán Vidal (Eds.). Springer, 304–320. https://doi.org/10.1007/978-3-642-12251-4_22

[14] Oleg Kiselyov, Chung-chieh Shan, and Amr Sabry. 2006. Delimited dynamic binding. In *Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming, ICFP 2006, Portland, Oregon, USA, September 16-21, 2006*, John H. Reppy and Julia Lawall (Eds.). ACM, 26–37. https://doi.org/10.1145/1159803.1159808

[15] Oleg Kiselyov and K. C. Sivaramakrishnan. 2016. Eff Directly in OCaml. In *Proceedings ML Family Workshop / OCaml Users and Developers workshops, ML/OCAML 2016, Nara, Japan, September 22-23, 2016 (EPTCS, Vol. 285)*, Kenichi Asai and Mark R. Shinwell (Eds.). 23–58. https://doi.org/10.4204/EPTCS.285.2

[16] Daan Leijen. 2014. Koka: Programming with Row Polymorphic Effect Types. In *Proceedings 5th Workshop on Mathematically Structured Functional Programming, MSFP@ETAPS 2014, Grenoble, France, 12 April 2014 (EPTCS, Vol. 153)*, Paul Blain Levy and Neel Krishnaswami (Eds.). 100–126. https://doi.org/10.4204/EPTCS.153.8

[17] Daan Leijen. 2017. Type Directed Compilation of Row-Typed Algebraic Effects. In *Proceedings of Principles of Programming Languages (POPL'17), Paris, France* (proceedings of principles of programming languages (popl'17), paris, france ed.). https://www.microsoft.com/en-us/research/publication/type-directed-compilation-row-typed-algebraic-effects/

[18] Marek Materzok and Dariusz Biernacki. 2012. A Dynamic Interpretation of the CPS Hierarchy. In *Programming Languages and Systems - 10th Asian Symposium, APLAS 2012, Kyoto, Japan, December 11-13, 2012. Proceedings (Lecture Notes in Computer Science, Vol. 7705)*, Ranjit Jhala and Atsushi Igarashi (Eds.). Springer, 296–311. https://doi.org/10.1007/978-3-642-35182-2_21

[19] Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. 2019. Typed Equivalence of Effect Handlers and Delimited Control. In *4th International Conference on Formal Structures for Computation and Deduction, FSCD 2019, June 24-30, 2019, Dortmund, Germany (LIPIcs, Vol. 131)*, Herman Geuvers (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 30:1–30:16. https://doi.org/10.4230/LIPIcs.FSCD.2019.30

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

[21] Matija Pretnar. 2015. An Introduction to Algebraic Effects and Handlers. Invited tutorial paper. *Electronic Notes in Theoretical Computer Science* 319 (2015), 19–35. https://doi.org/10.1016/j.entcs.2015.12.003 The 31st Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXI)..

[22] Asumu Takikawa, T. Stephen Strickland, and Sam Tobin-Hochstadt. 2013. Constraining Delimited Control with Contracts. In *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 7792)*, Matthias Felleisen and Philippa Gardner (Eds.). Springer, 229–248. https://doi.org/10.1007/978-3-642-37036-6_14

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

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

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