Logical Methods in Computer Science
Volume 20, Issue 4, 2024, pp. 17:1–17:50
https://lmcs.episciences.org/

Submitted Apr. 24, 2023
Published Nov. 22, 2024

A CALCULUS FOR SCOPED EFFECTS & HANDLERS

ROGER BOSMAN* [a], BIRTHE VAN DEN BERG* [a], WENHAO TANG* [b], AND TOM SCHRIJVERS [a]

[a] KU Leuven, Celestijnenlaan 200A, Belgium
e-mail address: {roger.bosman/birthe.vandenberg/tom.schrijvers}@kuleuven.be

[b] The University of Edinburgh, 10 Crichton Street, United Kingdom
e-mail address: wenhao.tang@ed.ac.uk

Abstract. Algebraic effects & handlers have become a standard approach for side-effects in functional programming. Their modular composition between different effects and clean separation of syntax and semantics make them attractive to a wide audience. However, not all effects can be classified as algebraic; some need a more sophisticated handling. In particular, effects that have or create a delimited scope need special care, as their continuation consists of two parts—in and out of the scope—and their modular composition introduces additional complexity. These effects are called *scoped* and have gained attention by their growing applicability and adoption in popular libraries. While calculi have been designed with algebraic effects & handlers built in to facilitate their use, a calculus that supports scoped effects & handlers in a similar manner does not yet exist. This work fills this gap: we present λ_{sc}, a calculus with native support for both algebraic and scoped effects & handlers. The most novel part of λ_{sc} is the support for modular composition of different scoped effects & handlers by extending handlers with forwarding clauses, which make λ_{sc} much more expressive than existing calculi with algebraic effects & handlers. Our calculus is based on Eff, an existing calculus for algebraic effects, extended with Koka-style row polymorphism, and consists of a formal grammar, operational semantics, and a (type-safe) type-and-effect system. We give a prototype implementation of λ_{sc} with type inference and demonstrate λ_{sc} on a range of examples.

1. Introduction

While monads [Mog89, Mog91, Wad95] have long been the go-to approach for modelling effects, *algebraic effects & handlers* [PP03, PP09] are steadily gaining more traction. They offer a more structured and modular approach to composing effects, based on an algebraic model. The approach consists of two parts: effects denote the syntax of operations, and handlers interpret them by means of structural recursion. By composing handlers that each interpret only a part of the syntax in the desired order, one can modularly build an interpretation for the entire program. Algebraic effects & handlers have been adopted in

Key words and phrases: algebraic effects, scoped effects, calculus, operational semantics, type- and effect system.

*These authors contributed equally to this work.

LOGICAL METHODS IN COMPUTER SCIENCE
DOI:10.46298/LMCS-20(4:17)2024

several libraries (e.g., fused-effects [RTWS18], extensible-effects [KSSF19], Eff in OCaml [KS18]) and languages (e.g., Eff [Pre15], Links [HL16], Koka [Lei17], Effekt [BSO20]).

Although the modular approach of algebraic effects & handlers is desirable for every effectful program, it is not always applicable. In particular, those effects that have or introduce a delimited scope (e.g., exceptions, concurrency, local state) are not algebraic. Essentially, these so-called *scoped effects* [WSH14] split the program into two: a part *in* the scope of the effect (called the scoped computation) and a part *out* of the scope (called the continuation).

This separation breaks algebraicity, which states that operations commute with sequencing. Modeling scoped effects as handlers [PP03] has been proposed as a way of encoding scoped effects in an algebraic framework. However, this comes at the cost of modularity [YPW+22] (and thus expressiveness). Wu et al. proposes to treat scoped effects as separate, built-in operations, and present their solution in a denotational setting. However, their solution of modular composition of scoped handlers is rather ad-hoc, and not a generalization of the algebraic case. Therefore, a treatment of scoped effects & handlers in an operational setting with full support for modular composition is desirable. The growing interest in scoped effects & handlers, evidenced by their adoption at GitHub [TRWS22] and in Haskell libraries (e.g., eff [Kin19], polysemy [Mag19], fused-effects [RTWS18]), motivates the need for such a calculus, all the more because they use the same ad-hoc approach of modular composition as suggested by Wu et al.

This paper aims to fill this gap in the literature: we cover scoped effects & handlers, which in previous work has only been covered in a denotational setting, in an operational setting by presenting λ_{sc}, a calculus with scoped effects & handlers. Our main source of inspiration is Eff [BP13, BP15, Pre15], a calculus for algebraic effects & handlers, effectively easing programming with those features. Although Eff is an appropriate starting point, the extension to support scoped effects & handlers is non-trivial, for two reasons. First, scoped effects require polymorphic handlers, which we support by let-polymorphism and F_ω-style type operators. Second, we need to be able to forward unknown operations in order to keep the desired modularity. Whereas algebraic effects & handlers have a uniform (and implicit) forwarding mechanism, scoped effects & handlers need explicit forwarding clauses in order to allow sufficient freedom in their implementation.

In what follows, after introducing the appropriate background (Section 2) and informally motivating the challenges and design choices of our calculus (Section 3), we formalize λ_{sc}. We make the following contributions:

- We present our solution for the modular composition of scoped handlers, called forwarding (Section 3.3), and generalize it later (Section 7).
- We design formal syntax for λ_{sc} terms, types and contexts (Section 4).
- We provide an operational semantics (Section 5).
- We define a type-and-effect system for λ_{sc} (Section 6).
- We formulate and prove λ_{sc}'s metatheoretical properties (Section 8).
- We show the usability of our calculus on a range of examples (Section 9).
- We provide an interpreter of our calculus with type inference in which we implement all our examples (supplementary material).

2. Background & Motivation

This section provides the necessary background and motivates our goal. We review *algebraic effects & handlers* as a modular approach to composing side-effects in effectful programs. Next, we present *scoped effects & handlers*: effects that have or create a delimited scope (such as `once` for nondeterminism [PSWJ18, WSH14]), and motivate the need for a calculus with built-in support for these scoped effects.

2.1. Algebraic effects & handlers.

Algebraic effects & handlers consist of *operations*, denoting their syntax, and *handlers*, denoting their semantics. This separation gives us modular composition, which has intrinsic value *and* allows controlling effect interaction.

2.1.1. *Algebraic Operations.*

Effects are denoted by a name (or *label*) and characterized by a *signature* $A \twoheadrightarrow B$, taking a value of type A and producing a value of type B. For example, $\mathtt{choose} : () \twoheadrightarrow \mathsf{Bool}$ takes a unit value and produces a boolean (e.g., nondeterministically). *Operations* invoke effects, combining the **op** keyword, an effect to invoke, a *parameter* passed to the effect, and a *continuation*, containing the rest of the program. For brevity of presentation, we follow the syntax of Eff [Pre15] to have an explicit continuation for every operation.

$$c_{\mathsf{ND1}} = \mathbf{op}\ \mathtt{choose}\ ()\ (b\,.\,\mathbf{if}\ b\ \mathbf{then\ return}\ 1\ \mathbf{else\ return}\ 2)$$

In accordance with its signature, `choose` is passed (), and in the supplied contination b has type Bool. As a result, c_{ND1} is a computation that returns either 1 or 2.

Some operations commute with sequencing. For example:

$$\begin{aligned} &\mathbf{do}\ x \leftarrow \mathbf{op}\ \mathtt{choose}\ ()\ (b\,.\,\mathbf{if}\ b\ \mathbf{then\ return}\ 1\ \mathbf{else\ return}\ 2)\,;\mathbf{return}\ x^2 \\ \equiv\ &\mathbf{op}\ \mathtt{choose}\ ()\ (b\,.\,\mathbf{do}\ x \leftarrow \mathbf{if}\ b\ \mathbf{then\ return}\ 1\ \mathbf{else\ return}\ 2\,;\mathbf{return}\ x^2) \end{aligned}$$

This equivalence is an instance of the *algebraicity property*, and operations are *algebraic* if they satisfy this property. Algebraicity states that the sequencing of a computation c_2 after an operation $\mathbf{op}\ \ell\ v\ (y\,.\,c_1)$ is equivalent to sequencing the same computation after the *continuation* of this operation:

$$\mathbf{do}\ x \leftarrow \mathbf{op}\ \ell\ v\ (y\,.\,c_1)\,;c_2\ \equiv\ \mathbf{op}\ \ell\ v\ (y\,.\,\mathbf{do}\ x \leftarrow c_1\,;c_2$$

2.1.2. *Handlers.*

Handlers give meaning to operations. For example, handler h_{ND} interprets `choose` by collecting all the choices in a list:

$$\begin{aligned} h_{\mathsf{ND}} = \mathbf{handler}\ \{&\mathbf{return}\ x \qquad\quad \mapsto \mathbf{return}\ [x] \\ &, \mathbf{op}\ \mathtt{choose}\ \_\ k \mapsto \mathbf{do}\ xs \leftarrow k\ \mathsf{true}\,;\mathbf{do}\ ys \leftarrow k\ \mathsf{false}\,;xs +\!\!+ ys\,\} \end{aligned}$$

This handler has two clauses. The first clause returns a singleton list in case a value x is returned. The second clause, which interprets `choose`, executes both branches by applying the continuation k to both true and false, and concatenates their resulting lists with the $(+\!\!+)$-operator. We apply h_{ND} to c_{ND1} to obtain both of its results:

$$\begin{aligned} &\mathbf{with}\ h_{\mathsf{ND}}\ \mathbf{handle}\ c_{\mathsf{ND1}} \\ \rightsquigarrow^*\ &[1,2] \end{aligned}$$

The algebraicity property plays a critical role in giving semantics to algebraic effects and handlers. For example, consider the following program which uses h_{ND} to handle a computation that sequences two `choose` operations using the **do** syntax.

$$c_{\mathsf{ND2}} = \mathbf{do}\ p \leftarrow \mathbf{op}\ \texttt{choose}\ ()\ (b\,.\,\mathbf{return}\ b)\,;\mathbf{op}\ \texttt{choose}\ ()\ (q\,.\,\mathbf{return}\ (p,q))$$

In order to handle the first `choose` operation, the handler h_{ND} needs to get access to the continuation of it. Thus, we use the algebraicity property of `choose` to put the second `choose` operation in the explicit continuation of the first `choose` operation.

$$\begin{aligned}
&\quad \mathbf{with}\ h_{\mathsf{ND}}\ \mathbf{handle}\ c_{\mathsf{ND2}}\\
&\rightsquigarrow\ \mathbf{with}\ h_{\mathsf{ND}}\ \mathbf{handle}\ \mathbf{op}\ \texttt{choose}\ ()\ (b\,.\\
&\qquad \mathbf{do}\ p \leftarrow \mathbf{return}\ b\,;\mathbf{op}\ \texttt{choose}\ ()\ (q\,.\,\mathbf{return}\ (p,q)))\\
&\rightsquigarrow^{*}\ [(\mathsf{true},\mathsf{true}),(\mathsf{true},\mathsf{false}),(\mathsf{false},\mathsf{true}),(\mathsf{false},\mathsf{false})]
\end{aligned}$$

Algebraic effects & handlers bring several interesting advantages. Most interestingly, their separation of syntax and semantics allows a modular composition of different effects, which in turn allows for altering the meaning of a program by different effect interactions.

For those familiar with the category-theoretical backings of folds, handler application can be seen as a fold over an abstract syntax tree, where operations are nodes in that tree, and handlers are the fold algebra. The result type of the fold is the carrier of the algebra.

2.1.3. *Modular Composition.* Effects can be composed by combining different primitive operations. For example, computation $c_{\mathsf{c,g}}$ below uses $\texttt{get} : () \rightarrowtriangle \mathsf{String}$ in addition to `choose`.

$$c_{\mathsf{c,g}} = \mathbf{op}\ \texttt{choose}\ ()\ (b\,.\,\mathbf{if}\ b\ \mathbf{then}\ \mathbf{return}\ 1\ \mathbf{else}\ \mathbf{op}\ \texttt{get}\ ()\ (x\,.\,\mathbf{return}\ x))$$

Instead of having to write a handler for each combination of effects, algebraic effects & handlers allow us to write a handler specific for the effect `get`, and to compose it with the existing handler h_{ND}.

$$h_{\mathsf{get}} = \mathbf{handler}\ \{\mathbf{return}\ x \mapsto \mathbf{return}\ x, \mathbf{op}\ \texttt{get}\ \_\ k \mapsto k\ 2\}$$

When composing handlers $\mathbf{with}\ h_{\mathsf{ND}}\ \mathbf{handle}\ (\mathbf{with}\ h_{\mathsf{get}}\ \mathbf{handle}\ c_{\mathsf{c,g}})$, h_{get} is applied first, and handles `get`. Since h_{get} does not contain a clause for `choose`, it leaves (we say *"forwards"*, see below) the `choose` operation to be handled by another handler. This forwarding behavior is key to the modular reuse and composition of handlers. Handler h_{ND} then takes care of the remaining effects.

$$\begin{aligned}
&\quad \mathbf{with}\ h_{\mathsf{ND}}\ \mathbf{handle}\ (\mathbf{with}\ h_{\mathsf{get}}\ \mathbf{handle}\ c_{\mathsf{c,g}})\\
&\rightsquigarrow^{*}\ \mathbf{with}\ h_{\mathsf{ND}}\ \mathbf{handle}\ c_{\mathsf{ND1}}\\
&\rightsquigarrow^{*}\ [1,2]
\end{aligned}$$

2.1.4. *Forwarding.* Critical to modular composition is the possibility to apply partial handlers, i.e. to apply handlers to computations where the handlers may only handle a subset of the effects present in the computations. As discussed in Section 2.1.2, handler application can be seen as a fold, where handlers form the folding algebra. Therefore, the partial algebra formed by partial handlers must be supplemented by a *forwarding* algebra that interprets the effects not handled by the handler such that they can be handled by another handler.

Generally, calculi for algebraic effects require that handler clauses leave computations as their output. The forwarding algebra must "reinterpret" computations into computations,

which is essentially the identity function. Therefore, for these calculi, forwarding can be and is done uniformly, which means handlers do not need to give specialised forwarding clauses for effects they do not handle. For example, `choose` is forwarded by h_{get} by simply leaving the effect with the handled computation.

$$\begin{aligned}&\quad \mathbf{with}\ h_{\mathsf{get}}\ \mathbf{handle}\ \mathbf{op}\ \mathtt{choose}\ ()\ (b\,.\,\mathbf{return}\ b)\\ &\rightsquigarrow \mathbf{op}\ \mathtt{choose}\ ()\ (b\,.\,\mathbf{with}\ h_{\mathsf{get}}\ \mathbf{handle}\ \mathbf{return}\ b)\end{aligned}$$

In the calculus we will present we cannot forward generically, and instead utilize explicit forwarding clauses. This is not novel: it already shows up in the algebraic case when generalizing handler clauses when they may return types that are not computations. Since the return type of handlers corresponds to the carrier of the fold, when generalizing handler clauses like this we get a fold with a carrier type that is not a computation. In this case, we can no longer uniformly forward the unhandled operations. Instead, we must specify, per carrier, how to forward unhandled operations by means of an explicit forwarding clause [SPWJ19]. Requirement of explicit forwarding algebras for algebraic effects and handlers often happens in libraries, but not in languages, because languages with primitive support for algebraic effects and handlers always require the carriers of handlers to be computations.

As stated, in the case of scoped effects, we are required to specify such an explicit forwarding clause even when the carrier is a computation. For now, we would like to observe that this forwarding clause is *not* a consequence of implementing scoped effects, but merely a consequence of moving away from a (very specific) instance of algebraic effects.

2.1.5. *Effect Interaction.* One of the valuable features of the modular composition of algebraic effects & handlers is that effects can interact differently by applying their handlers in a different order. Consider the effect $\mathtt{inc} : () \twoheadrightarrow \mathsf{Int}$, which produces an (incremented) integer. The handler h_{inc} turns computations into state-passing functions.

$$\begin{aligned}h_{\mathsf{inc}} = \mathbf{handler}\ \{&\mathbf{return}\ x \quad \mapsto \mathbf{return}\ (\boldsymbol{\lambda} s\,.\,\mathbf{return}\ (x, s))\\ ,&\mathbf{op}\ \mathtt{inc}\ \_\ k \mapsto \mathbf{return}\ (\boldsymbol{\lambda} s\,.\,\mathbf{do}\ s' \leftarrow s + 1\,;\\ &\qquad\qquad\qquad\qquad\quad \mathbf{do}\ k' \leftarrow k\ s'\,;\\ &\qquad\qquad\qquad\qquad\quad k'\ s')\}\end{aligned}$$

The state s represents the current counter value. On every occurrence of `inc`, the incremented value is passed to the continuation twice: (1) for updating the counter value and (2) for returning the result of the operation. The latter is for the continuation and the former for serving the next `inc` operation. We use the folowing syntactic sugar:

$$\mathit{run}_{\mathsf{inc}}\ s\ c\ \equiv\ \mathbf{do}\ c' \leftarrow \mathbf{with}\ h_{\mathsf{inc}}\ \mathbf{handle}\ c\,;\, c'\ s$$

The computation c_{inc} combines `choose` and `inc`:

$$c_{\mathsf{inc}} = \mathbf{op}\ \mathtt{choose}\ ()\ (b\,.\,\mathbf{if}\ b\ \mathbf{then}\ \mathbf{op}\ \mathtt{inc}\ ()\ (x\,.\,x + 5)\ \mathbf{else}\ \mathbf{op}\ \mathtt{inc}\ ()\ (y\,.\,y + 2))$$

When handling `inc` first, each `choose` branch gets the same initial counter value 0.

$$\begin{aligned}&\quad \mathbf{with}\ h_{\mathsf{ND}}\ \mathbf{handle}\ \mathit{run}_{\mathsf{inc}}\ 0\ c_{\mathsf{inc}}\\ &\rightsquigarrow^* \mathbf{with}\ h_{\mathsf{ND}}\ \mathbf{handle}\ \mathbf{op}\ \mathtt{choose}\ ()\ (b\,.\\ &\qquad \mathbf{do}\ p' \leftarrow \mathbf{if}\ b\ \mathbf{then}\ \mathbf{with}\ h_{\mathsf{inc}}\ \mathbf{handle}\ (\mathbf{op}\ \mathtt{inc}\ ()\ (x\,.\,x + 5))\\ &\qquad\qquad\qquad\quad \mathbf{else}\ \mathbf{with}\ h_{\mathsf{inc}}\ \mathbf{handle}\ (\mathbf{op}\ \mathtt{inc}\ ()\ (y\,.\,y + 2))\,;\\ &\qquad p'\ 0)\\ &\rightsquigarrow^* [(6, 1), (3, 1)]\end{aligned}$$

In contrast, when handling `choose` first, the counter value is threaded through the successive branches, showing that with algebraic effects & handlers, the manner in which effects interact can be controlled by the order in which handlers are applied.

$$
\begin{aligned}
& \mathit{run}_{\mathsf{inc}}\ 0\ (\mathbf{with}\ h_{\mathsf{ND}}\ \mathbf{handle}\ c_{\mathsf{inc}}) \\
\rightsquigarrow^*\ & \mathit{run}_{\mathsf{inc}}\ 0\ (\mathbf{do}\ \mathit{xs} \leftarrow \mathbf{with}\ h_{\mathsf{ND}}\ \mathbf{handle}\ \mathbf{op}\ \mathtt{inc}\ ()\ (x\,.\,x+5)\,; \\
& \qquad\qquad \mathbf{do}\ \mathit{ys} \leftarrow \mathbf{with}\ h_{\mathsf{ND}}\ \mathbf{handle}\ \mathbf{op}\ \mathtt{inc}\ ()\ (y\,.\,y+2)\,; \\
& \qquad\qquad \mathbf{return}\ \mathit{xs} +\!\!+ \mathit{ys}) \\
\rightsquigarrow^*\ & ([6,4],2)
\end{aligned}
$$

2.2. **Scoped effects.** Not all effects are algebraic. There is a wide class of higher-order effects [BS24] which do not satisfy the algebraicity property including scoped effects [WSH14], latent effects [BSPW21], and parallel effects [XJMP21]. In this paper, we focus on scoped effects. Consider extending nondeterminism with an effect that takes a computation that contains nondeterministic choice and returns only its first result. This is known as the `once` operation [PSWJ18]. Since effect operations syntactically already receive the continuation as and argument, which looks like a computation, we might be tempted to use it for this purpose, i.e. to use it to denote the scope of `once`. Subscripting ✗ to indicate an erroneous example, we could attempt to syntactically write this as the algebraic operation $\mathtt{once}_{\text{✗}} : () \twoheadrightarrow ()$ by extending h_{ND} with a clause for $\mathtt{once}_{\text{✗}}$ that executes the continuation, and then takes the head of the result if possible. Otherwise it just returns the empty list. We then try to prune the first `choose` of c_{ND2} by using `once`.

$$
\begin{aligned}
h_{\mathtt{once}_{\text{✗}}} = \mathbf{handler}\ \{\ldots, \mathbf{op}\ \mathtt{once}_{\text{✗}}\ \_\ k \mapsto\ & \mathbf{do}\ \mathit{ts} \leftarrow k\ ()\,; \\
& \mathbf{do}\ b \leftarrow \mathit{ts} = [\,]\,; \\
& \mathbf{if}\ b\ \mathbf{then}\ \mathbf{return}\ [\,]\ \mathbf{else}\ \mathsf{head}\ \mathit{ts}\}
\end{aligned}
$$

$$
\begin{aligned}
c_{\mathtt{once}_{\text{✗}}} = \ & \mathbf{do}\ p \leftarrow \mathbf{op}\ \mathtt{once}_{\text{✗}}\ ()\ (\_\,.\,\mathbf{op}\ \mathtt{choose}\ ()\ (b\,.\,\mathbf{return}\ b))\,; \\
& \mathbf{op}\ \mathtt{choose}\ ()\ (q\,.\,\mathbf{return}\ (p,q))
\end{aligned}
$$

We intend for $\mathbf{with}\ h_{\mathtt{once}_{\text{✗}}}\ \mathbf{handle}\ c_{\mathtt{once}_{\text{✗}}}$ to return $[(\mathsf{true},\mathsf{true}),(\mathsf{true},\mathsf{false})]$ as the first `choose` is pruned by $\mathtt{once}_{\text{✗}}$ to only return the first alternative. The second `choose` is out of scope of $\mathtt{once}_{\text{✗}}$, so should still return both results. However, algebraicity pulls the second `choose` inside the scope of $\mathtt{once}_{\text{✗}}$:

$$
\begin{aligned}
& \mathbf{with}\ h_{\mathtt{once}_{\text{✗}}}\ \mathbf{handle}\ (\mathbf{do}\ p \leftarrow \mathbf{op}\ \mathtt{once}_{\text{✗}}\ ()\ (\_\,.\,\mathbf{op}\ \mathtt{choose}\ ()\ (b\,.\,\mathbf{return}\ b))\,; \\
& \qquad\qquad \mathbf{op}\ \mathtt{choose}\ ()\ (q\,.\,\mathbf{return}\ (p,q))) \\
\rightsquigarrow\ & \mathbf{with}\ h_{\mathtt{once}_{\text{✗}}}\ \mathbf{handle}\ \mathtt{once}_{\text{✗}}\ ()\ (\_\,.\,\mathbf{do}\ \mathbf{op}\ \mathtt{choose}\ ()\ (b\,.\,\mathbf{return}\ b)\,; \\
& \qquad\qquad\qquad\qquad \mathbf{op}\ \mathtt{choose}\ ()\ (q\,.\,\mathbf{return}\ (p,q))) \\
\rightsquigarrow^*\ & [(\mathsf{true},\mathsf{true})]
\end{aligned}
$$

There are many more examples of operations that have a scope; some of which we present in Section 9:

- `catch` for catching exceptions that are raised during program execution;
- `local` for creating local variables (local state);
- `call` for creating a scope in a nondeterministic program, where branches can be cut using the algebraic `cut` operation;
- `depth` for bounding the depth in the depth-bounded search strategy;

Following Wu et al. [WSH14], we call them *scoped operations*. Plotkin and Power [PP03] have already realised that algebraic effects are unable to represent so-called effect deconstructors (e.g. scoped effects) and propose to model them as handlers. Although this solution is used (for example by Thompson et al. [TRWS22]), it merges syntax and semantics at the cost of modularity [WSH14, YPW+22]. In Section 9 we revisit this issue, showing attempts at encoding scoped effects as handlers, their problems, and how λ_{sc} remedies the situation.

The goal of this work is to implement scoped effects while maintaining a separation between syntax and semantics, and thus preserve modular composition and control over effect interaction. It follows a line of research [PSWJ18, WSH14, YPW+22] that has developed denotational semantic domains, backed by categorical models. However, they did not consider the modular composition of scoped effects and handlers in their denotional semantics. What is lacking from the literature is a calculus that allows programming with both algebraic and scoped operations and their handlers as well as supports the modular composition between them.

3. Design Decisions & Challenges

This section informally discusses the design of λ_{sc}, a novel calculus with support for scoped effects & handlers as built-in features. We present the main challenges and design choices, and address how to forward unknown operations.

3.1. Scoped Effects as Built-in Operations.

λ_{sc} supports scoped effects as built-in operations, signalled by the **sc** keyword.

$$\mathbf{sc}\ \mathtt{once}\ ()\ (y\,.\,c_1)\ (z\,.\,c_2)$$

Like algebraic operations, scoped operations feature a label `once` to identify the effect, a parameter, in this case (), and a continuation $(z\,.\,c_2)$. However, unlike algebraic operations, scoped operations feature an additional *scoped computation* $(y\,.\,c_1)$. For `once`, the scoped computation entails the computation to be restricted to the first result (i.e. the computation in scope). The result from the scoped computation c_1 is passed to the continuation c_2 by means of argument z.

Scoped handler clauses feature an additional argument when compared to algebraic handler clauses, corresponding to the scoped computation. For example, below we define a handler for `once`, which extends h_{ND}.

$$\begin{aligned} h_{\mathsf{once}} = \mathbf{handler}\ \{\ \ldots, \mathbf{sc}\ \mathtt{once}\ \_\ p\ k \mapsto\ & \mathbf{do}\ ts \leftarrow p\ ()\,; \\ & \mathbf{do}\ b \leftarrow ts = [\,]\,; \\ & \mathbf{if}\ b\ \mathbf{then\ return}\ [\,]\ \mathbf{else\ do}\ t \leftarrow \mathsf{head}\ ts\,;\, k\ t\ \} \end{aligned}$$

Adding scoped operations gives rise to a variant of the algebraicity property, which models the desired behavior of sequencing for scoped operations: scoped operations commute with sequencing in the continuation, but leave the scoped computation intact.

$$\mathbf{do}\ x \leftarrow \mathbf{sc}\ \ell^{\mathsf{sc}}\ v\ (y\,.\,c_1)\ (z\,.\,c_2)\,;\, c_3\ \equiv\ \mathbf{sc}\ \ell^{\mathsf{sc}}\ v\ (y\,.\,c_1)\ (z\,.\,\mathbf{do}\ x \leftarrow c_2\,;\, c_3)$$

Using `once` as a scoped operation correctly restrict only the first `choose`, as can be seen when applying h_{once} to c_{once}, which is like $c_{\mathsf{once}\boldsymbol{\times}}$ defined in Section 2.2 but with the scoped effect `once` instead of the algebraic effect $\mathtt{once}_{\boldsymbol{\times}}$.

$$c_{\mathsf{once}} = \mathbf{sc}\ \texttt{once}\ ()\ (\_\,.\,\mathbf{op}\ \texttt{choose}\ ()\ (b\,.\,\mathbf{return}\ b))\\ (p\,.\,\mathbf{do}\ q \leftarrow \mathbf{op}\ \texttt{choose}\ ()\ (b\,.\,\mathbf{return}\ b)\,;\mathbf{return}\ (p,q))$$

$$\mathbf{with}\ h_{\mathsf{once}}\ \mathbf{handle}\ c_{\mathsf{once}} \rightsquigarrow^* [(\mathsf{true},\mathsf{true}),(\mathsf{true},\mathsf{false})]$$

Signatures. Scoped effects, like algebraic effects, have signatures $A \twoheadrightarrow B$. Like algebraic effects, the left-hand side of this signature refers to the type of the value passed to the operation (() in the case of `once`). However, where the right-hand side of the signature in the case of algebraic effects refers to the argument of the *continuation*, in the case of scoped effects it refers to the *scoped computation*. Whilst this may seem strange from this point of view, it makes sense from the point of view of writing handler clauses: in both cases, a value of type A is given, and a value of type B is produced by the handler clause. The difference comes from what computation this produced value of type B is passed to: algebraic effect handler clauses pass this value to the continuation, whereas scoped effect handler clauses pass this value to the scoped computation (whose result is passed to the continuation).

Therefore, the signature of `once` is $() \twoheadrightarrow ()$ since the handler clause calls the scoped computation with (). The scoped result type (Section 3.3) varies depending on the computation `once` is applied to, and is therefore not a part of the signature.

3.2. Eff **with Row Typing.** Our calculus is based on Eff [Pre15, BP13, BP15], an existing calculus for algebraic effects & handlers. However, instead of using the subtyping-based effect system of Eff, we use a row-based effect system in the style of Koka [Lei17] for simplicity. Therefore, computations have types of shape $A\,!\,\langle E\rangle$, where A is the type of the value returned by the computation, and E is a collection (row) of effects that *may* occur during its evaluation. For example, $(\mathsf{Bool},\mathsf{Bool})\,!\,\langle\texttt{choose},\texttt{once}\rangle$ is the type of c_{once}.

Handler types, of shape $A\,!\,\langle E\rangle \Rightarrow B\,!\,\langle F\rangle$, reflect that handlers turn one computation (of type $A\,!\,\langle E\rangle$) into another (of type $B\,!\,\langle F\rangle$). For example, h_{once} handles `choose` and `once`.

$$h_{\mathsf{once}} : (\mathsf{Bool},\mathsf{Bool})\,!\,\langle\texttt{choose},\texttt{once}\rangle \Rightarrow \mathsf{List}\,(\mathsf{Bool},\mathsf{Bool})\,!\,\langle\rangle$$

3.3. **Scoped result type.** As we have seen before, the two computations taken by a scoped operation $\mathbf{sc}\ \ell^{\mathsf{sc}}\ v\ (y\,.\,c_1)\ (z\,.\,c_2)$ are not just two irrelevant terms; they are connected in the sense that the result of the scoped computation $y\,.\,c_1$ is passed to the continuation $z\,.\,c_2$ and exactly bound as z in c_2. Therefore, they must agree on the type of this result, which we name the *scoped result type*. For example, consider c_{once} with overall type $(\mathsf{Bool},\mathsf{Bool})\,!\,\langle\texttt{once}\,;\texttt{choose}\rangle$. Its scoped result type is (a singular) Bool: it is the type that is *produced* by the scoped computation, and *consumed* by the continuation.

$$\mathbf{sc}\ \texttt{once}\ ()\ (\underbrace{\_\,.\,\mathbf{op}\ \texttt{choose}\ ()\ (b\,.\,\mathbf{return}\ b)}_{()\to \mathsf{Bool}\,!\,\langle\texttt{once}\,;\texttt{choose}\rangle})\ (\underbrace{p\,.\,\mathbf{do}\ q \leftarrow \mathbf{op}\ldots\ \mathbf{return}\ (p,q)}_{\mathsf{Bool}\to(\mathsf{Bool},\mathsf{Bool})\,!\,\langle\texttt{once}\,;\texttt{choose}\rangle})$$

Dealing with the presence of this scoped result type is the most non-trivial part for designing a calculus with composable scoped effects. There are two main complications. First, since the type does not occur in the computation's overall type, *polymorphic handlers* are required to handle scoped effects. Second, the scoped result type describes a dependency between the scoped computation and continuation: if one changes its type, the other must match this. This makes generic *forwarding* impossible: it alters the type of the scoped computation, but does not make up for it in the continuation.

The rest of this section covers these two complications. While understanding these issues is important for fully understanding the semantics presented in Section 5, it may be hard to fully comprehend the remainder section without having seen the type system presented in Section 6. We recommend the reader to skim over parts of the remainder of this section that are unclear, and revisit them after having seen the type system.

3.3.1. *Polymorphic Handlers.* Applying a handler to a computation involves recursively applying the handler to the computation's subcomputations as well. In the case of algebraic effects, these subcomputations always have the same type as the operation itself, as witnessed by the algebraicity property. This means that calculi that only support algebraic effects & handlers, such as Eff, allow typing handlers monomorphically, without limitations in expressivity.

However, typing scoped effect handlers monomorphically *does* limit their implementation freedom: it only allows scoped operations of which the scoped result type matches the operation's overall type. For example, consider the type $(\mathsf{Bool}, \mathsf{Bool})\,!\,\langle \mathtt{choose}\,;\mathtt{once}\rangle \Rightarrow \mathsf{List}\ (\mathsf{Bool}, \mathsf{Bool})\,!\,\langle\rangle$ we previously assigned to h_{once}. This monomorphic type requires the scoped result type to be $(\mathsf{Bool}, \mathsf{Bool})$ as well, as it is the only type of computation monomorphic h_{once} can handle. This is not the case for c_{once}: as established, its scoped result type is Bool. Therefore, scoped computations such as c_{once}, cannot be handled by monomorphic handlers. The solution is to let handlers abstract over the value type of computations, allowing for the handling of scoped operations with *any* scoped result type. This way, h_{once} can be typed as follows:

$$h_{\mathsf{once}} : \forall\, \alpha\,.\,\alpha\,!\,\langle \mathtt{choose}\,;\mathtt{once}\rangle \Rightarrow \mathsf{List}\ \alpha\,!\,\langle\rangle$$

With this polymorphic typing in place, **with** h_{once} **handle** c_{once} can now be evaluated by *polymorphic recursion*. To support this, λ_{sc} features **let**-polymorphism, F_ω-style type operators, and requires all handlers for scoped effects to be polymorphic.

3.3.2. *Forwarding Unknown Operations.* In order to retain the modularity of composing different effects, as discussed in Section 2.1.4, we allow for partial handlers (i.e. handlers that handle only a subset of the effects they are applied to) to obtain dedicated handlers that interpret only their part of the syntax. This requires that all remaining operations be *forwarded* to other handlers. As established, this forwarding can, in the case for algebraic effects and in some specific situations, be done in a canonical way.

One might hope to forward scoped effects in a similar way, where the inner computations are handled, and the effect is left as-is for future handlers to handle. For example, consider the following computation which uses a scoped operation $\mathtt{catch} : \mathsf{String} \twoheadrightarrow \mathsf{Bool}$ for catching exceptions.

$$c_{\mathsf{catch}} = \mathbf{sc}\ \mathtt{catch}\ \mathtt{"err"}\ (b\,.\,\mathbf{if}\ b\ \mathbf{then\ return}\ 1\ \mathbf{else\ return}\ 2)\ (x\,.\,\mathbf{return}\ x)$$
$$\quad \mathbf{with}\ h_{\mathsf{once}}\ \mathbf{handle}\ c_{\mathsf{catch}}$$

Applying the handler h_{once} to it gives us

$$\rightsquigarrow_{\not\mathsf{x}}\ \mathbf{sc}\ \mathtt{catch}\ \mathtt{"err"}\ (b\,.\,\mathbf{with}\ h_{\mathsf{once}}\ \mathbf{handle\ if}\ b\ldots)\ (x\,.\,\mathbf{with}\ h_{\mathsf{once}}\ \mathbf{handle\ return}\ x)$$

Unfortunately, this does not work: again, the hurdle is in the scoped result type, in combination with the fact that applying a handler to a computation changes its type. In our example, h_{once} applies the type operator List when handling a computation, resulting in a handled scoped computation of type $\mathsf{Bool} \to \mathsf{List\ Int}\,!\,\langle \mathtt{catch} \rangle$, and a handled continuation of type $\mathsf{Int} \to \mathsf{List\ Int}\,!\,\langle \mathtt{catch} \rangle$.

$$\mathbf{sc}\ \mathtt{catch}\ \mathtt{"err"}\ (\underbrace{b\,.\,\mathbf{with}\ h_{\mathsf{once}}\ \mathbf{handle\ if}\ b\ \mathbf{then}\ \ldots}_{\mathsf{Bool} \to \mathsf{List\ Int}\,!\,\langle \mathtt{catch} \rangle})\ (\underbrace{x\,.\,\mathbf{with}\ h_{\mathsf{once}}\ \mathbf{handle\ return}\ x}_{\mathsf{Int} \to \mathsf{List\ Int}\,!\,\langle \mathtt{catch} \rangle})$$

This introduced a type mismatch, as the return type of the scoped computation has changed (List Int), whereas the continuation still expects the original type (Int). In other words: the scoped computation and continuation no longer agree on a scoped result type. Thus, scoped effects cannot be forwarded uniformly. Therefore, we require that every handler is equipped with an explicit forwarding clause for unknown scoped operations, specific to the handler's carrier (i.e. the type constructor is applied to a computation when handled). This forwarding clause is used to resolve the type mismatch between the result type of the handled scoped computation and the parameter type of continuation.

Considering the above example, the goal of applying a function of type $\mathsf{Int} \to \mathsf{List\ Int}\,!\,\langle E \rangle$ to a value of type List Int naturally reminds us of monadic binding. Indeed, for most carriers, this forwarding clause is very similar to that of the monadic bind $\gg\!=$. In fact, for now we will assume this to be the case for all effects, and introduce λ_{sc} based on these bind-like forwarding clauses. In Section 7 we will revisit this firstly by giving examples where bind-like clauses are not expressive enough, secondly by generalizing the forwarding clause, and lastly by showing how to desugar bind-like clauses to generalized forwarding clauses. Therefore, forwarding clauses are—for now—of shape $\mathbf{bind}\ x\ k$, where x represents the result of the scoped computation, and k represents the continuation, yielding a type similar to $\gg\!=$.

What should we write for a forwarding clause? The answer depends on what we expect when we want to connect the results of a handler to outside. For example, consider the handler *h Once*. The x here is the list of results we get from the nondeterministic program in the scope after handled by h_{once}. The k here is what remains to do for each result. The most intuitive way to connect x and k is to run k on every element in x and connect their results together. This is exactly what the concatMap function in Haskell does.

$$h_{\mathsf{once}} = \mathbf{handler}\ \{\ \ldots, \mathbf{bind}\ x\ k \mapsto \mathsf{concatMap}\ x\ k\ \}$$

We define concatMap in λ_{sc} as follows.

$$\begin{aligned}
&\mathsf{concatMap}\ :\forall\ \alpha\ \beta\ \mu\,.\,\mathsf{List}\ \beta \to^{\mu} (\beta \to^{\mu} \mathsf{List}\ \alpha) \to^{\mu} \mathsf{List}\ \alpha\\
&\mathsf{concatMap}\ [\,]\qquad f = \mathbf{return}\ [\,]\\
&\mathsf{concatMap}\ (b : bs)\ f = \mathbf{do}\ as \leftarrow f\ b\,;\, as' \leftarrow \mathsf{concatMap}\ bs\ f\,;\, as +\!\!+ as'
\end{aligned}$$

Note that there is no unique correct forwarding clause for every handler; it is up to the programmers to decide which kind of forwarding behaviours they want. For example, for h_{once} we can also use a reversed version of concatMap which does the same thing except for traversing the list from right to left. Alternatively, we can only pass the first element of x to k, or even invoke other effects like printing some information when forwarding. Comparing these different styles of forwarding, we prefer the original version using a left-to-right concatMap since we usually do not want other scoped effects to unexpectedly change the results of h_{once} by discarding some results or reversing the order of results. From our

$$
\begin{array}{rcll}
\text{values } v & ::= & () \mid (v_1, v_2) \mid x \mid \boldsymbol{\lambda} x \,.\, c \mid h & \\
\text{handlers } h & ::= & \mathbf{handler}\ \{\mathbf{return}\ x \mapsto c_r & \text{return clause} \\
 & & \quad , \mathit{oprs} & \text{effect clauses} \\
 & & \quad , \mathbf{bind}\ x\ k \mapsto c_f\ \} & \text{forwarding clause} \\
\text{operation clauses } \mathit{oprs} & ::= & \cdot & \text{empty } \mathit{oprs} \\
 & \mid & \mathbf{op}\ \ell^{\mathsf{op}}\ x\ k \mapsto c, \mathit{oprs} & \text{algebraic effect clauses} \\
 & \mid & \mathbf{sc}\ \ell^{\mathsf{sc}}\ x\ p\ k \mapsto c, \mathit{oprs} & \text{scoped effect clauses} \\
\text{computations } c & ::= & \mathbf{return}\ v & \text{return value} \\
 & \mid & \mathbf{op}\ \ell^{\mathsf{op}}\ v\ (y \,.\, c) & \text{algebraic operation} \\
 & \mid & \mathbf{sc}\ \ell^{\mathsf{sc}}\ v\ (y \,.\, c_1)\ (z \,.\, c_2) & \text{scoped operation} \\
 & \mid & \mathbf{with}\ v\ \mathbf{handle}\ c & \text{handle} \\
 & \mid & \mathbf{do}\ x \leftarrow c_1\,;\, c_2 & \text{do-statement} \\
 & \mid & v_1\ v_2 & \text{application} \\
 & \mid & \mathbf{let}\ x = v\ \mathbf{in}\ c & \text{let}
\end{array}
$$

Figure 1: Terms of λ_{sc}.

experience of implementing a range of examples of scoped effects in Section 9, we found that a simple and intuitive forwarding clause usually yields the expect semantics.

In what follows, we put our calculus on formal footing, discussing its syntax, operational semantics and type-and-effect system.

4. Term Syntax

As stated, λ_{sc} is based on Eff [BP13, BP15]. Before adding support for scoped effects, we have altered Eff from its presentation in [BP13, BP15] in two ways. Firstly, we have made a number of cosmetic changes that arguably improve the readability of the calculus. Secondly, we adopt row-based typing in the style of Koka [Lei17]. The row types will be introduced in Section 6.

Like Eff we implement fine-grained call-by-value semantics [LPT03]. Therefore, terms are split into inert values and computations that can be reduced.

4.1. Computations.

For computations, **return** can be used to return values. Handlers can be applied to values with the **with** and **handle** keywords. As seen before, computations may be sequenced by means of do-statements (**do** $x \leftarrow c_1\,;\, c_2$). Applications reduce, so they are computations. As discussed in Section 3.3.1, to support polymorphic handlers we support let-polymorphism and thus let-bindings. Finally, a computation may be the invocation of an effect by means of an operation.

To be able to differentiate between algebraic and scoped effects, we add the effect keyword **sc** to model scoped effects. Consequently, **op** now ranges over algebraic effects only. Furthermore, we annotate labels with either op or sc to indicate if they are the label of an algebraic or scoped effect, respectively. We implicitly assume any label ℓ occurs either as an algebraic or scoped effect label. Like their algebraic counterparts, scoped effect operations feature a label ℓ^{sc}, argument v and continuation $(z \,.\, c_2)$. In addition, scoped effect operations feature a scoped computation $(y \,.\, c_1)$. This way, the scope of effect ℓ^{sc} is delimited: (scoped) computation $(y \,.\, c_1)$ is in scope, continuation $(z \,.\, c_2)$ is not.

4.2. **Values.** Values consist of the unit value (), value pairs (v_1, v_2), variables x, functions $\boldsymbol{\lambda} x \,.\, c$ and handlers h. Handlers h have three kinds of clauses: one return clause, zero or more operation clauses, and a forwarding clause.

The return clause **return** $x \mapsto c$ denotes that the result x of a computation is processed by computation c.

Algebraic operation clauses **op** $\ell^{\mathsf{op}}\ x\ k \mapsto c$ specify that handling an effect with label ℓ^{op}, parameter x and continuation k is processed by computation c (e.g., h_{ND}, h_{get}, h_{inc}). For scoped effect clauses the extension is analogous to the operation case: we take the algebraic clause and add support for a scoped computation, which in the case for the clause has the form of parameter p.

Finally, as motivated in Section 3.3.2, we have forwarding clauses of shape **bind** $x\ k \mapsto c$ that deal with forwarding unknown scoped operations with some label ℓ^{sc}. We will generalize these in Section 7.

5. Operational Semantics

Figure 2 displays the small-step operational semantics of λ_{sc}. Here, relation $c \rightsquigarrow c'$ denotes that computation c steps to computation c', with $\rightsquigarrow^*$ its reflexive, transitive closure. The highlighted rules deal with the extensions that support scoped effects. The following discussion of the semantics is exemplified by snippets of derivations of computations[1] used in Section 2. We refer to Appendix A for the full version of these derivations.

Rules E-AppAbs and E-Let deal with function application and let-binding, respectively, and are standard. The rest of the rules consist of two parts: sequencing and handling.

Sequencing. For sequencing computations **do** $x \leftarrow c_1\,;c_2$, we distinguish between the situation where c_1 can take a step (E-Do), and where c_1 is in normal form (**return**, **op**, or **sc**). First, if c_1 returns a value v, we substitute v for x in c_2 (E-DoRet). Second, if c_1 is an algebraic operation, we rewrite the computation using the algebraicity property (E-DoOp), bubbling up the algebraic operation to the front of the computation. Third, the new case, where c_1 is a scoped operation, is analogous: the generalization of the algebraicity property (Section 3.1) is used to rewrite the computation (E-DoSc).

Handling. For handling computations with a handler of the form **with** h **handle** c, we distinguish six situations. First, if possible, c takes a step (E-Hand); in the other cases, c is in normal form. If c returns a value v, we use the handler's return clause **return** $x \mapsto c_r$, switching evaluation to c_r with x replaced by v (E-HandRet).

If computation c is an algebraic operation **op** $\ell^{\mathsf{op}}\ v\ (y \,.\, c_1)$, its label is looked up in the handler h. If the handler contains an algebraic clause with this label, evaluation switches to the clause's computation c (E-HandOp), with v substituted for parameter x and continuation k replaced by a function that, given the original argument y, contains the already-handled[2] continuation. For example, **with** h_{ND} **handle** c_{ND1} (p. 3) reduces as follows.

[1] Following convention, these examples may contain elements not present in our calculus, such as integers and if-then-else statements. These may be viewed as syntactic sugar for their Church encodings.

[2] Like eff, λ_{sc} implements *deep* handlers, as opposed to *shallow* handlers.[HL18a]

$\boxed{c \rightsquigarrow c'}$ Computation reduction

$$\frac{}{(\boldsymbol{\lambda} x \,.\, c)\ v \rightsquigarrow c\ [v\ /\ x]}\ \text{E-AppAbs} \qquad \frac{}{\mathbf{let}\ x = v\ \mathbf{in}\ c \rightsquigarrow c\ [v\ /\ x]}\ \text{E-Let}$$

$$\frac{c_1 \rightsquigarrow c_1'}{\mathbf{do}\ x \leftarrow c_1\,;\, c_2 \rightsquigarrow \mathbf{do}\ x \leftarrow c_1'\,;\, c_2}\ \text{E-Do} \qquad \frac{}{\mathbf{do}\ x \leftarrow \mathbf{return}\ v\,;\, c_2 \rightsquigarrow c_2\ [v\ /\ x]}\ \text{E-DoRet}$$

$$\frac{}{\mathbf{do}\ x \leftarrow \mathbf{op}\ \ell^{\mathsf{op}}\ v\ (y\,.\,c_1)\,;\, c_2 \rightsquigarrow \mathbf{op}\ \ell^{\mathsf{op}}\ v\ (y\,.\,\mathbf{do}\ x \leftarrow c_1\,;\, c_2)}\ \text{E-DoOp}$$

$$\frac{}{\mathbf{do}\ x \leftarrow \mathbf{sc}\ \ell^{\mathsf{sc}}\ v\ (y\,.\,c_1)\ (z\,.\,c_2)\,;\, c_3 \rightsquigarrow \mathbf{sc}\ \ell^{\mathsf{sc}}\ v\ (y\,.\,c_1)\ (z\,.\,\mathbf{do}\ x \leftarrow c_2\,;\, c_3)}\ \text{E-DoSc}$$

$$\frac{c \rightsquigarrow c'}{\mathbf{with}\ h\ \mathbf{handle}\ c \rightsquigarrow \mathbf{with}\ h\ \mathbf{handle}\ c'}\ \text{E-Hand}$$

$$\frac{(\mathbf{return}\ x \mapsto c_r)\ \in\ h}{\mathbf{with}\ h\ \mathbf{handle}\ \mathbf{return}\ v \rightsquigarrow c_r\ [v\ /\ x]}\ \text{E-HandRet}$$

$$\frac{(\mathbf{op}\ \ell^{\mathsf{op}}\ x\ k \mapsto c)\ \in\ h}{\mathbf{with}\ h\ \mathbf{handle}\ \mathbf{op}\ \ell^{\mathsf{op}}\ v\ (y\,.\,c_1) \rightsquigarrow c\ [v\ /\ x, (\boldsymbol{\lambda} y\,.\,\mathbf{with}\ h\ \mathbf{handle}\ c_1)\ /\ k]}\ \text{E-HandOp}$$

$$\frac{(\mathbf{op}\ \ell^{\mathsf{op}}\ \_\ \_)\ \notin\ h}{\mathbf{with}\ h\ \mathbf{handle}\ \mathbf{op}\ \ell^{\mathsf{op}}\ v\ (y\,.\,c_1) \rightsquigarrow \mathbf{op}\ \ell^{\mathsf{op}}\ v\ (y\,.\,\mathbf{with}\ h\ \mathbf{handle}\ c_1)}\ \text{E-FwdOp}$$

$$\frac{(\mathbf{sc}\ \ell^{\mathsf{sc}}\ x\ p\ k \mapsto c)\ \in\ h}{\mathbf{with}\ h\ \mathbf{handle}\ \mathbf{sc}\ \ell^{\mathsf{sc}}\ v\ (y\,.\,c_1)\ (z\,.\,c_2) \rightsquigarrow \begin{array}{l} c\ [v\ /\ x, (\boldsymbol{\lambda} y\,.\,\mathbf{with}\ h\ \mathbf{handle}\ c_1)\ /\ p, \\ \quad (\boldsymbol{\lambda} z\,.\,\mathbf{with}\ h\ \mathbf{handle}\ c_2)\ /\ k] \end{array}}\ \text{E-HandSc}$$

$$\frac{(\mathbf{sc}\ \ell^{\mathsf{sc}}\ \_\ \_\ \_)\ \notin\ h \qquad (\mathbf{bind}\ x\ k \mapsto c)\ \in\ h}{\mathbf{with}\ h\ \mathbf{handle}\ \mathbf{sc}\ \ell^{\mathsf{sc}}\ v\ (y\,.\,c_1)\ (z\,.\,c_2) \rightsquigarrow \begin{array}{l} \mathbf{sc}\ \ell^{\mathsf{sc}}\ v\ (y\,.\,\mathbf{with}\ h\ \mathbf{handle}\ c_1)\ (z\,.\,c\ [z\ /\ x, \\ \quad (\boldsymbol{\lambda} z\,.\,\mathbf{with}\ h\ \mathbf{handle}\ c_2)\ /\ k]) \end{array}}\ \text{E-Bind}$$

Figure 2: Operational semantics of λ_{sc}.

$$\begin{array}{ll} & \mathbf{with}\ h_{\mathsf{ND}}\ \mathbf{handle}\ c_{\mathsf{ND1}} \\ \rightsquigarrow & \mathbf{do}\ xs \leftarrow (\boldsymbol{\lambda} b\,.\,\mathbf{with}\ h_{\mathsf{ND}}\ \mathbf{handle}\ \mathbf{if}\ b\ \mathbf{then}\ \mathbf{return}\ 1\ \mathbf{else}\ \mathbf{return}\ 2)\ \mathsf{true} \\ & \mathbf{do}\ ys \leftarrow (\boldsymbol{\lambda} b\,.\,\mathbf{with}\ h_{\mathsf{ND}}\ \mathbf{handle}\ \mathbf{if}\ b\ \mathbf{then}\ \mathbf{return}\ 1\ \mathbf{else}\ \mathbf{return}\ 2)\ \mathsf{false} \\ & \quad xs +\!\!+ ys \\ \rightsquigarrow^* & \mathbf{return}\ [1, 2] \end{array}$$

In case h does not contain a clause for label ℓ^{op}, the effect is forwarded (E-FwdOp). Algebraic effects can be forwarded generically: we re-invoke the operation and recursively

apply the handler to continuation c_1. For example, during the application of run_{inc} in **with** h_{ND} **handle** $run_{\mathsf{inc}}\ 0\ c_{\mathsf{inc}}$ (p. 5), `choose` is forwarded:

$$
\begin{aligned}
& \textbf{with}\ h_{\mathsf{ND}}\ \textbf{handle}\ run_{\mathsf{inc}}\ 0\ c_{\mathsf{inc}} \\
\rightsquigarrow\ & \textbf{with}\ h_{\mathsf{ND}}\ \textbf{handle do}\ p' \leftarrow \textbf{op}\ \texttt{choose}\ ()\ (b\,.\,\textbf{with}\ h_{\mathsf{inc}}\ \textbf{handle if}\ b\ \textbf{then op}\ \texttt{inc}\ () \\
& \qquad (x\,.\,x+5)\ \textbf{else op}\ \texttt{inc}\ ()\ (y\,.\,y+2))\,;p'\ 0 \\
\rightsquigarrow^*\ & \textbf{return}\ [(6,1),(3,1)]
\end{aligned}
$$

If computation c is a scoped operation $\mathbf{sc}\ \ell^{\mathsf{sc}}\ v\ (y\,.\,c_1)\ (z\,.\,c_2)$, we again distinguish two situations: the case where h contains a clause for ℓ^{sc}, and where it does not. If h contains a clause for label ℓ^{sc}, evaluation switches to the clause's computation c (E-HandSc), with v substituted for parameter x. Both the scoped computation and the continuation are replaced by a function that contains the already-handled computations c_1 and c_2. For example, this happens for the scoped operation `once` in **with** h_{once} **handle** c_{once} (p. 7).

$$
\begin{aligned}
& \textbf{with}\ h_{\mathsf{once}}\ \textbf{handle}\ c_{\mathsf{once}} \\
\rightsquigarrow\ & \textbf{do}\ ts \leftarrow (\boldsymbol{\lambda}\_\,.\,\textbf{with}\ h_{\mathsf{once}}\ \textbf{handle op}\ \texttt{choose}\ (b\,.\,\textbf{return}\ b))\ () \\
& \textbf{do}\ t\ \leftarrow \mathsf{head}\ ts \\
& \quad (\boldsymbol{\lambda}p\,.\,\textbf{with}\ h_{\mathsf{once}}\ \textbf{handle}\ (\textbf{do}\ q \leftarrow \textbf{op}\ \texttt{choose}\ (b\,.\,\textbf{return}\ b)\,;\textbf{return}\ (p,q)))\ t \\
\rightsquigarrow^*\ & \textbf{return}\ [(\mathsf{true},\mathsf{true}),(\mathsf{true},\mathsf{false})]
\end{aligned}
$$

When h does not contain a clause for label ℓ^{sc}, we must forward the effect. As we showed in Section 3.3.2, forwarding scoped effects cannot happen canonically, but (for now) rather proceeds via the handler's forwarding clause **bind** $x\ k \mapsto c$. Similar to the forwarding of algebraic effects, we recursively handle the subcomputations. However, in the case of **bind**, we set the continuation of the to-be-forwarded effect to the forwarding clause's body c, substituted with the relevant arguments.

For a computation, consider again the example described in Section 3.3.2. Even though we have not given any semantics to `catch` yet (we will do so in Section 9.1), we know h_{once} does not contain a clause for `catch`. As described, `catch` must be forwarded, addressing the type mismatch between the handled scoped computation and handled continuation with concatMap.

$$
\begin{aligned}
& \textbf{with}\ h_{\mathsf{once}}\ \textbf{handle}\ c_{\mathsf{catch}} \\
\rightsquigarrow\ & \textbf{sc}\ \texttt{catch}\ \texttt{"err"}\ (b\,.\,\textbf{with}\ h_{\mathsf{once}}\ \textbf{handle if}\ b\ \textbf{then return}\ 1\ \textbf{else return}\ 2) \\
& \qquad\qquad (x\,.\,\mathsf{concatMap}\ x\ (\boldsymbol{\lambda}z\,.\,\textbf{with}\ h_{\mathsf{once}}\ \textbf{handle return}\ z))
\end{aligned}
$$

6. Type-and-Effect System

This section presents the type-and-effect system of λ_{sc}. As before, we distinguish between values, computations and handlers.

6.1. **Grammar.** Figure 3 displays the grammar of the types of λ_{sc}. Like terms, types are split: values have value types A, B, computations have computation types $\underline{C}, \underline{D}$. Value types consist of the unit type $()$, pair types (A, B), function types $A \to \underline{C}$, handler types $\underline{C} \Rightarrow \underline{D}$, type variables α and abstraction over them $\boldsymbol{\lambda}\alpha\,.\,A$, and type application $M\ A$. Following convention and to allow for meaningful examples, we may add base value types to the calculus, such as String, Int and Bool. Functions take a value of type A as argument and

$$
\begin{array}{rcll}
\text{value types } A,\, B,\, M & ::= & () \mid (A, B) \mid A \to \underline{C} \mid \underline{C} \Rightarrow \underline{D} & \\
& \mid & \alpha & \text{type variable} \\
& \mid & \lambda\, \alpha\, .\, A & \text{type operator abstraction} \\
& \mid & M\ A & \text{type application} \\
\text{type schemes } \sigma & ::= & A \mid \forall\, \mu\, .\, \sigma \mid \forall\, \alpha\, .\, \sigma & \\
\text{computation types } \underline{C},\, \underline{D} & ::= & A\,!\langle E \rangle & \\
\text{effect rows } E,\, F & ::= & \cdot \mid \mu \mid \ell\,; E & \\
\text{signature contexts } \Sigma & ::= & \cdot \mid \Sigma, \ell^{\mathsf{op}} : A \twoheadrightarrow B \mid \Sigma, \ell^{\mathsf{sc}} : A \twoheadrightarrow B & \\
\text{type contexts } \Gamma & ::= & \cdot \mid \Gamma, x : \sigma \mid \Gamma, \mu \mid \Gamma, \alpha &
\end{array}
$$

Figure 3: Types of λ_{sc}.

$\boxed{\Gamma \vdash v : \sigma}$ Value Typing

$$\frac{(x : \sigma) \in \Gamma}{\Gamma \vdash x : \sigma}\ \text{T-Var} \qquad \frac{}{\Gamma \vdash () : ()}\ \text{T-Unit} \qquad \frac{\Gamma \vdash v_1 : A \qquad \Gamma \vdash v_2 : B}{\Gamma \vdash (v_1, v_2) : (A, B)}\ \text{T-Pair}$$

$$\frac{\Gamma, x : A \vdash c : \underline{C}}{\Gamma \vdash \boldsymbol{\lambda} x\, .\, c : A \to \underline{C}}\ \text{T-Abs} \qquad \frac{\Gamma \vdash v : A \qquad A \equiv B}{\Gamma \vdash v : B}\ \text{T-EqV}$$

$$\frac{\Gamma \vdash v : \forall\, \alpha\, .\, \sigma \qquad \Gamma \vdash A}{\Gamma \vdash v : [A \,/\, \alpha]\, \sigma}\ \text{T-Inst} \qquad \frac{\Gamma, \alpha \vdash v : \sigma \qquad \alpha \notin \Gamma}{\Gamma \vdash v : \forall\, \alpha\, .\, \sigma}\ \text{T-Gen}$$

$$\frac{\Gamma \vdash v : \forall\, \mu\, .\, \sigma \qquad \Gamma \vdash E}{\Gamma \vdash v : [E \,/\, \mu]\, \sigma}\ \text{T-InstEff} \qquad \frac{\Gamma, \mu \vdash v : \sigma \qquad \mu \notin \Gamma}{\Gamma \vdash v : \forall\, \mu\, .\, \sigma}\ \text{T-GenEff}$$

Figure 4: Value typing.

return a computation of type $\underline{C}$; handlers take a computation of type $\underline{C}$ as argument and return a computation of type $\underline{D}$.

A computation type $A\,!\langle E \rangle$ consists of a value type A, representing the type of the value the computation evaluates to, and an effect type E, representing the effects that *may* be called during this evaluation. Different from Eff, we implement effect types as effect rows using row polymorphism [Lei05] in the style of Koka [Lei17]. Therefore, rows E are represented as collections of the previously discussed atomic labels ℓ^{op} and ℓ^{sc}, optionally terminated by a row variable μ. Finally, we can abstract over both type and row variables, giving rise to type schemes σ.

6.2. **Value typing.** Figure 4 displays the typing rules for values. Rules T-Var, T-Unit, T-Pair and T-Abs type variables, units, pairs and term abstractions, respectively, and are standard.

Rule T-EqV expresses that typing holds up to equivalence of types. The full type equivalence relation ($A \equiv B$), which also uses the equivalence of rows ($E \equiv_{\langle\rangle} F$), is included in Appendix B. However, these relations can be described as the congruence closure

$\boxed{\Gamma \vdash c : \underline{C}}$ Computation Typing

$$\frac{\Gamma \vdash v_1 : A \to \underline{C} \qquad \Gamma \vdash v_2 : A}{\Gamma \vdash v_1\ v_2 : \underline{C}}\ \text{T-App} \qquad \frac{\Gamma \vdash c_1 : A!\langle E\rangle \qquad \Gamma, x : A \vdash c_2 : B!\langle E\rangle}{\Gamma \vdash \mathbf{do}\ x \leftarrow c_1\,;\, c_2 : B!\langle E\rangle}\ \text{T-Do}$$

$$\frac{\Gamma \vdash c : \underline{C} \qquad \underline{C} \equiv \underline{D}}{\Gamma \vdash c : \underline{D}}\ \text{T-EqC} \qquad \frac{\Gamma \vdash v : \sigma \qquad \Gamma, x : \sigma \vdash c : \underline{C}}{\Gamma \vdash \mathbf{let}\ x = v\ \mathbf{in}\ c : \underline{C}}\ \text{T-Let}$$

$$\frac{\Gamma \vdash v : A \qquad \Gamma \vdash E}{\Gamma \vdash \mathbf{return}\ v : A!\langle E\rangle}\ \text{T-Ret} \qquad \frac{\Gamma \vdash v : \underline{C} \Rightarrow \underline{D} \qquad \Gamma \vdash c : \underline{C}}{\Gamma \vdash \mathbf{with}\ v\ \mathbf{handle}\ c : \underline{D}}\ \text{T-Hand}$$

$$\frac{(\ell^{\mathsf{op}} : A_1 \twoheadrightarrow A_2) \in \Sigma \qquad \Gamma \vdash v : A_1 \qquad \Gamma, y : A_2 \vdash c : A_3!\langle E\rangle \qquad \ell^{\mathsf{op}} \in E}{\Gamma \vdash \mathbf{op}\ \ell^{\mathsf{op}}\ v\ (y\,.\,c) : A_3!\langle E\rangle}\ \text{T-Op}$$

$$\frac{\begin{array}{c}(\ell^{\mathsf{sc}} : A_1 \twoheadrightarrow A_2) \in \Sigma \\ \Gamma \vdash v : A_1 \qquad \Gamma, y : A_2 \vdash c_1 : A_3!\langle E\rangle \qquad \Gamma, z : A_3 \vdash c_2 : A_4!\langle E\rangle \qquad \ell^{\mathsf{sc}} \in E\end{array}}{\Gamma \vdash \mathbf{sc}\ \ell^{\mathsf{sc}}\ v\ (y\,.\,c_1)\ (z\,.\,c_2) : A_4!\langle E\rangle}\ \text{T-Sc}$$

Figure 5: Computation typing.

of the following two rules.

$$\frac{}{(\lambda\,\alpha\,.\,A)\ B\ \equiv\ A\,[B\,/\,\alpha]}\ \text{Q-AppAbs} \qquad \frac{\ell_1 \neq \ell_2}{\ell_1\,;\ell_2\,;E\ \equiv_{\langle\rangle}\ \ell_2\,;\ell_1\,;E}\ \text{R-Swap}$$

Rule Q-AppAbs captures type application, following F_ω, and R-Swap captures the insignificance of the order in effect rows, following Koka's row typing approach.

The final four value typing rules deal with generalization and instantiation of type variables and row variables. Rule T-Inst instantiates the type variables α in a type scheme with a value type A. Rule T-Gen is its dual, abstracting over a type variable. The rules for row variables are similar: T-InstEff instantiates row variable with an effect row E; T-GenEff abstracts over a row variable. The definition of well-scopedness for types $\Gamma \vdash \sigma$ and effect rows $\Gamma \vdash E$ is straightforward (Appendix C).

6.3. **Computation typing.** Figure 5 shows the rules for computation typing. Rules T-App and T-Do capture application and sequencing, and are standard. Like value typing, typing of computations holds up to equivalence of types (T-EqC). Rule T-Let is part of our extension of Eff, as scoped effects require introducing let-polymorphism.

Rule T-Ret assigns a computation type to a return statement. This type consists of the value v in the return, together with a effect row E. Notice that, as in Koka, this row can be freely chosen.

Rule T-Hand types handler application. The typing rules for handlers and their clauses are discussed in Section 6.4. A handler of type $C \Rightarrow D$ denotes a handler that transforms computations of type C to a computation of type D.

Rule T-Op types algebraic effects. Looking up label ℓ^{op} in Σ yields a signature $A_1 \twoheadrightarrow A_2$, where A_1 is the type of the operation's parameter v, and A_2 is the type of argument y of

$\boxed{\Gamma \vdash \mathbf{return}\ x \mapsto c_r : \underline{C}}$ $\boxed{\Gamma \vdash \mathit{oprs} : \underline{C}}$ $\boxed{\Gamma \vdash \mathbf{bind}\ x\ k \mapsto c_f : \underline{C}}$
Handler clause typing

$$\frac{\Gamma, x : A \mapsto c_r : \underline{C}}{\Gamma \vdash \mathbf{return}\ x \mapsto c_r : \underline{C}}\ \textsc{T-Return} \qquad \frac{}{\Gamma \vdash \cdot : \underline{C}}\ \textsc{T-Empty}$$

$$\frac{\Gamma \vdash \mathit{oprs} : \underline{C} \qquad (\ell^{\mathsf{op}} : A_1 \twoheadrightarrow A_2) \in \Sigma \qquad \Gamma, x : A_1, k : A_2 \to \underline{C} \vdash c : \underline{C}}{\Gamma \vdash \mathbf{op}\ \ell^{\mathsf{op}}\ x\ k \mapsto c, \mathit{oprs} : \underline{C}}\ \textsc{T-OprOp}$$

$$\frac{\Gamma \vdash \mathit{oprs} : M\ A_1!\langle E\rangle \qquad (\ell^{\mathsf{sc}} : A_2 \twoheadrightarrow A_3) \in \Sigma \qquad \Gamma, \beta, x : A_2, p : A_3 \to M\ \beta!\langle E\rangle, k : \beta \to M\ A_1!\langle E\rangle \vdash c : M\ A_1!\langle E\rangle}{\Gamma \vdash \mathbf{sc}\ \ell^{\mathsf{sc}}\ x\ p\ k \mapsto c, \mathit{oprs} : M\ A_1!\langle E\rangle}\ \textsc{T-OprSc}$$

$$\frac{\Gamma, \alpha, x : M\ \alpha, k : \alpha \to M\ A!\langle E\rangle \vdash c_f : M\ A!\langle E\rangle}{\Gamma \vdash \mathbf{bind}\ x\ k \mapsto c_f : M\ A!\langle E\rangle}\ \textsc{T-Bind}$$

$\boxed{\Gamma \vdash h : \forall\, \alpha\,.\, \alpha!\langle E\rangle \Rightarrow M\ \alpha!\langle F\rangle}$ Handler typing

$$\textsc{T-Handler}_{\textsc{Bind}}\ \frac{\langle F\rangle \equiv_{\langle\rangle} \langle \mathit{labels}\,(\mathit{oprs})\,; E\rangle \qquad \Gamma, \alpha \vdash \mathbf{return}\ x \mapsto c_r : M\ \alpha!\langle E\rangle \qquad \Gamma, \alpha \vdash \mathit{oprs} : M\ \alpha!\langle E\rangle \qquad \Gamma, \alpha \vdash \mathbf{bind}\ x\ k \mapsto c_f : M\ \alpha!\langle E\rangle}{\Gamma \vdash \mathbf{handler}\ \{\mathbf{return}\ x \mapsto c_r, \mathit{oprs}, \mathbf{bind}\ x\ k \mapsto c_f\} : \forall\, \alpha\,.\, \alpha!\langle F\rangle \Rightarrow M\ \alpha!\langle E\rangle}$$

Figure 6: Handler typing.

the continuation. The resulting effect row includes ℓ^{op}. Indeed, $\ell \in E$ means that there is some E' such that $E \equiv_{\langle\rangle} \ell\,; E'$. Finally, the operation's type equals that of continuation c.

Similarly, rule T-Sc types scoped effects. Again, looking up label ℓ^{sc} in Σ yields signature $A_{\mathsf{sc}} \twoheadrightarrow B_{\mathsf{sc}}$ where A_{sc} corresponds to the type of the operation's parameter v. However, where B_{op} in the algebraic case refers to the *continuation's* argument, B_{sc} now describes the *scoped computation's* argument. This leaves the the scoped result type undescribed by the signature, but as discussed in Section 3.3.1, this freedom is exactly what we want. As for the effect rows, T-Sc requires the rows of the scoped computation to match.

6.4. **Handler typing.** The typing rules for handlers and handler clauses are shown in Figure 6. They consist of four judgments. Judgment $\Gamma \vdash \mathbf{return}\ x \mapsto c_r : M\ A!\langle E\rangle$ types return clauses, $\Gamma \vdash \mathit{oprs} : M\ A!\langle E\rangle$ types operation clauses, and $\Gamma \vdash \mathbf{bind}\ x\ k \mapsto c : M\ A!\langle E\rangle$ types forwarding clauses. Finally, $\Gamma \vdash h : \forall\, \alpha\,.\, \alpha!\langle F\rangle \Rightarrow M\ \alpha!\langle E\rangle$ types handlers, using the first three judgments.

Return Clauses. Rule T-Return types return clauses of the form $\mathbf{return}\ x \mapsto c_r$. It binds variable x to type A, adds it to the context, and returns the type $M\ A!\langle E\rangle$ of c_r as the type of the return clause.

Operation Clauses. The judgment $\Gamma \vdash \mathit{oprs} : M\ A!\langle E\rangle$ denotes that all operations in the sequence of operations *oprs* have type $M\ A!\langle E\rangle$. The base case T-Empty types the empty sequence. The other two cases require the head of the sequence (either **op** or **sc**) to have the same type as the tail.

Rule T-OprOp types algebraic operation clauses $\mathbf{op}\ \ell^{\mathsf{op}}\ x\ k \mapsto c$. Looking up label ℓ^{op} in Σ yields signature $A_{\mathsf{op}} \twoheadrightarrow B_{\mathsf{op}}$, where A_{op} describes the type of parameter x, and B_{op} the type of the argument of continuation k. In order for an operation **op** to have type $M\ A!\langle E\rangle$, c should have the same type.

Once again, the case for typing a scoped clause $\mathbf{sc}\ \ell^{\mathsf{sc}}\ x\ p\ k \mapsto c$ (T-OprSc) is similar to its algebraic equivalent, extended to include the scoped computation. Notice the type of p and k when typing c. First, as λ_{sc} allows freedom in the scoped result type, the type variable β is used for this type. Second, as shown in the operational semantics (rule E-HandSc), for a clause $\mathbf{sc}\ \ell^{\mathsf{sc}}\ x\ p\ k \mapsto c$, computation c uses the *already-handled* subcomputations p and k. Therefore, type operator M occurs in the scoped result type as well as in the continuation's result type:

$$p : B_{\mathsf{sc}} \to M\ \beta!\langle E\rangle \qquad\qquad k : \beta \to M\ A!\langle E\rangle$$

This means that, even though our focus on mitigating the type mismatch between scoped computation and continuation so far has been on forwarding *unknown* scoped effects, the same applies when handling *known* scoped effects, where computation c accounts for this discrepancy.

Forwarding Clause. In accordance with Section 3.3.2, when an application of a handler to a scoped effect the handler does not contain a clause for is encountered, the effect must be forwarded. Like the algebraic case, the effect is left in-place with handled sub-computations. However, as described in Section 3.3.2, simply handling the sub-computations would result in a type mismatch. It is clear from the typing rule that $\mathbf{bind}\ x\ k \mapsto c_f$ basically deals with this type mismatch between the type $M\ \alpha$ of x and the parameter type α of k.

Handler. Rule T-Handler types handlers with a polymorphic type of the form $\forall\, \alpha\,.\, \alpha\,!\ \langle F\rangle \Rightarrow M\ \alpha\,!\,\langle E\rangle$. A handler consists of a **return**-clause (T-Return), zero or more operation clauses (T-Empty, T-OprOp and T-OprSc), and a forwarding clause (T-Fwd). All clauses should agree on their result type $M\ \alpha\,!\,\langle E\rangle$. Notice that E denotes a collection with at least the labels of the present algebraic and scoped operation clauses in the handler (computed by the *labels*-function).

7. Generalized Forwarding

In Section 3.3.2 we showed how **bind** can be used to forward certain scoped effects. Here we also noted that not all effects can be forwarded by **bind**, and that some require a more general forwarding construct. In this section we revisit this issue. First, we show how in some situations, we need a more expressive forwarding clause. We will then present a generalized forwarding clause, and show how the **bind**-based forwarding clauses can be straightforwardly desugared into generalized forwarding clauses. Finally, we will amend the calculus with rules for its grammar, semantics and typing.

7.1. *Inc*, **revisited.** We introduced h_{inc} in an algebraic setting, and have not returned to it. We have not yet given a forwarding clause for h_{inc} (with carrier $(\mathsf{Int} \to (\alpha, \mathsf{Int})!\langle E\rangle)!\langle E\rangle$ for some effect type E). We could implement the forwarding clause of it using **bind** as follows.

$$\begin{aligned} h_{\mathsf{inc}\not\mathsf{x}} = \mathbf{handler}\,\{&\mathbf{return}\ x \quad \mapsto \mathbf{return}\ (\boldsymbol{\lambda} s\,.\,\mathbf{return}\ (x, s)) \\ ,&\mathbf{op}\ \mathtt{inc}\ \_\ k \mapsto \mathbf{return}\ (\boldsymbol{\lambda} s\,.\,\mathbf{do}\ s' \leftarrow s + 1\,;\mathbf{do}\ k' \leftarrow k\ s'\,;k'\ s') \\ ,&\mathbf{bind}\ x\ k \quad \mapsto \mathbf{return}\ (\boldsymbol{\lambda} s\,.\,\mathbf{do}\ (y, s') \leftarrow x\ s\,;k' \leftarrow k\ y\,;k'\ s')\,\} \end{aligned}$$

However, this implementation of forwarding would actually result in unexpected behaviours. Consider the derivation displayed in Figure 7. The chosen example has an occurrence of `choose` inside an occurrence of `inc`, both of them inside an occurrence of `once`. Since `choose` is in the scope of `once`, we would expect that the `choose` is pruned by the `once` to only keep the first branch. However, the derivation shows that by using the **bind**-based forwarding clause of $h_{\mathsf{inc}\not\mathsf{x}}$ the `choose` escapes the scope of `once`. As can be seen, when `head` (i.e. the embodiment of the `once` effect) is evaluated, `choose` has not even been handled yet; it is captured by a function in the result list.

For the brevity of presentation, we omitted the unimportant part of terms in the derivation, and we inline the guard that checks for the empty list in the if statement. We underlined parts that will reduce in the next step(s).

The essence of this problem comes from the fact that the carrier of h_{inc} contain computations. As a result, other operations in scope might be captured in these computations and then be carried out of the scope. The **bind**-based forwarding of h_{inc} is not expressive enough to avoid this kind of unexpected behaviour, because it has no control over what is returned by the scoped computation of scoped operations. For the example above, to correctly put `choose` in the scope, we need to evaluate the result function of the scoped computation of `once` that is handled by h_{inc}, instead of directly returning this function which unexpectedly captures `choose`. We will show how to solve this problem using a generalized version of forwarding clauses in the remaining of this section.

In general, this problem shows up for any handler whose carrier suspends computations, such as the carrier of h_{inc} which is a function. The effects captured in the suspend computation can escape their scope without being handled. The generalized forwarding clause fixes this problem by providing us the flexibility to early execute the suspend computation before leaving the scope.

7.2. **Adding generalized forwarding clauses.** Whereas in the case of **bind**-like forwarding clauses the remittal of the unknown, to-be-forwarded scoped effect is dealt with by the semantics (i.e., the reinvocation of scoped effect is embedded in the rule E-Bind), in the case of generalized forwarding clauses, denoted by **fwd**, this responsibility is deferred to the forwarding clause. This allows generalized forwarding clauses to execute logic *before* remitting the forwarded effect, and to alter not only the continuation but also the scoped computation of the forwarded effect. An intuitive implementation of **fwd** may look like this:

$$\frac{(\mathbf{sc}\ \ell^{\mathsf{sc}}\ \_\ \_\ \_) \notin h \qquad (\mathbf{fwd}'\ \ell^{\mathsf{sc}\prime}\ x\ p\ k \mapsto c_f) \in h}{\begin{array}{l}\mathbf{with}\ h\ \mathbf{handle}\ \mathbf{sc}\ \ell^{\mathsf{sc}}\ v\ (y\,.\,c_1)\ (z\,.\,c_2) \rightsquigarrow \\ \qquad\qquad c_f\ [\ell^{\mathsf{sc}}\ /\ \ell^{\mathsf{sc}\prime}, v\ /\ x, \\ \qquad\qquad\ (\boldsymbol{\lambda} y\,.\,\mathbf{with}\ h\ \mathbf{handle}\ c_1)\ /\ p, \\ \qquad\qquad\ (\boldsymbol{\lambda} z\,.\,\mathbf{with}\ h\ \mathbf{handle}\ c_2)\ /\ k\,]\end{array}}\ \text{E-FwdSc'}$$

The forwarding clause has form $\mathbf{fwd}'\ \ell^{\mathsf{sc}\prime}\ x\ p\ k \mapsto c_f$, where x, p, k represents for the parameter, (deeply handled) scoped computation, and (deeply handled) continuation, respectively.

$$
\begin{aligned}
&\textbf{with } h_{\mathsf{once}} \textbf{ handle}\\
&\quad \textbf{do } f \leftarrow \underline{\textbf{with } h_{\mathsf{inc}\text{✗}} \textbf{ handle}}\\
&\qquad \textbf{sc } \underline{\texttt{once}}\ ()\ (\_ .\ \textbf{op } \texttt{inc}\ ()\ (\_ .\ \textbf{op } \texttt{choose}\ ()\ (b .\ \textbf{return } b)))\ (z .\ \textbf{return } z)\,;\\
&\quad f\ 0\\
\rightsquigarrow^* &\ \textbf{with } h_{\mathsf{once}} \textbf{ handle}\\
&\quad \underline{\textbf{do } f \leftarrow}\\
&\qquad \textbf{sc } \texttt{once}\ ()\ (\_ .\ \textbf{with } h_{\mathsf{inc}\text{✗}} \textbf{ handle } (\textbf{op } \texttt{inc}\ ()\ (\_ .\ \textbf{op } \texttt{choose}\ ()\ (b .\ \textbf{return } b))))\\
&\qquad\qquad (z .\ \ldots)\\
&\quad \underline{f\ 0}\\
\rightsquigarrow &\ \underline{\textbf{with } h_{\mathsf{once}} \textbf{ handle}}\\
&\quad \textbf{sc } \underline{\texttt{once}}\ ()\ (\_ .\ \textbf{with } h_{\mathsf{inc}\text{✗}} \textbf{ handle } (\textbf{op } \texttt{inc}\ ()\ (\_ .\ \textbf{op } \texttt{choose}\ ()\ (b .\ \textbf{return } b)))))\\
&\qquad (z .\ \ldots)\\
\rightsquigarrow^* &\ \textbf{do } ts \leftarrow \textbf{with } h_{\mathsf{once}} \textbf{ handle}\\
&\qquad\qquad \underline{\textbf{with } h_{\mathsf{inc}\text{✗}} \textbf{ handle}}\ (\textbf{op } \underline{\texttt{inc}}\ ()\ (\_ .\ \textbf{op } \texttt{choose}\ ()\ (b .\ \textbf{return } b)))\\
&\quad \textbf{if } (ts = [\,]) \textbf{ then return } [\,] \textbf{ else do } t \leftarrow \mathsf{head}\ ts\,; \ldots\\
\rightsquigarrow^* &\ \textbf{do } ts \leftarrow \underline{\textbf{with } h_{\mathsf{once}} \textbf{ handle}}\\
&\qquad\qquad \underline{\textbf{return}}\ (\boldsymbol{\lambda} s .\ \textbf{do } s' \leftarrow s + 1\,;\\
&\qquad\qquad\quad \textbf{do } k' \leftarrow \textbf{with } h_{\mathsf{inc}\text{✗}} \textbf{ handle op } \texttt{choose}\ ()\ (b .\ \textbf{return } b)\,;\\
&\qquad\qquad\quad k'\ s')\\
&\quad \textbf{if } (ts = [\,]) \textbf{ then return } [\,] \textbf{ else do } t \leftarrow \mathsf{head}\ ts\,; \ldots\\
\rightsquigarrow^* &\ \textbf{do } \underline{ts} \leftarrow \underline{\textbf{return}}\ [\boldsymbol{\lambda} s .\ \textbf{do } s' \leftarrow s + 1\,;\\
&\qquad\qquad\qquad\quad \textbf{do } k' \leftarrow \textbf{with } h_{\mathsf{inc}\text{✗}} \textbf{ handle op } \texttt{choose}\ ()\ (b .\ \textbf{return } b)\,;\\
&\qquad\qquad\qquad\quad k'\ s']\,;\\
&\quad \underline{\textbf{if }}\ (ts = [\,])\ \underline{\textbf{ then }}\ \textbf{return } [\,]\ \underline{\textbf{ else }}\ \textbf{do } t \leftarrow \mathsf{head}\ ts\,; \ldots\\
\rightsquigarrow^* &\ \textbf{do } t \leftarrow \mathsf{head}\ [\boldsymbol{\lambda} s .\ \textbf{do } s' \leftarrow s + 1\,;\\
&\qquad\qquad\qquad\quad \textbf{do } k' \leftarrow \textbf{with } h_{\mathsf{inc}\text{✗}} \textbf{ handle op } \texttt{choose}\ ()\ (b .\ \textbf{return } b)\,;\\
&\qquad\qquad\qquad\quad k'\ s']\,;\\
&\quad \ldots\\
\rightsquigarrow^* &\ [(\mathsf{true}, 1), (\mathsf{false}, 1)]
\end{aligned}
$$

Figure 7: `once` escaping its scope when forwarded by **bind**

While this would give us our desired expressivity, it requires us to extend the calculus with the ability to abstract and substitute operation labels.

To avoid unnecessary complication, we opt for another design choice. Because of parametricity, there is only one thing a forwarding clause can do with an unknown label: invoke it with some (possibly altered) scoped computation and continuation. Below we utilize this property by, instead of supplying the forwarding clause with the label, we supply it with what is essentially a partial application of the **sc** operator to the unknown label. This partial application can be used by the forwarding clause in exactly the same way as the original label was. Therefore, this simplifies the type system, but does not affect the expressivity of forwarding clauses.

$$\frac{(\mathbf{sc}\ \ell^{\mathsf{sc}}\ \_\ \_\ \_) \notin h \qquad (\mathbf{fwd}\ f\ p\ k \mapsto c_f) \in h}{\begin{array}{l}\mathbf{with}\ h\ \mathbf{handle} \\ \quad \mathbf{sc}\ \ell^{\mathsf{sc}}\ v\ (y\,.\,c_1)\ (z\,.\,c_2)\end{array} \rightsquigarrow \begin{array}{l} c_f\ [(\boldsymbol{\lambda} y\,.\,\mathbf{with}\ h\ \mathbf{handle}\ c_1)\ /\ p, \\ \quad (\boldsymbol{\lambda} z\,.\,\mathbf{with}\ h\ \mathbf{handle}\ c_2)\ /\ k, \\ \quad (\boldsymbol{\lambda}(p', k')\,.\,\mathbf{sc}\ \ell^{\mathsf{sc}}\ v\ (y\,.\,p'\ y)\ (z\,.\,k'\ z))\ /\ f\,]\end{array}}\ \text{E-FwdSc}$$

7.3. **Syntax & typing.** We replace the **bind** syntax in handlers with **fwd**, and implement **bind** as syntactic sugar for **fwd**.

$$\begin{array}{rcll} \text{handlers } h & ::= & \mathbf{handler}\ \{\ldots & \\ & & \quad , \mathbf{fwd}\ f\ p\ k \mapsto c_f\,\} & \text{generalized forwarding clause} \\ \mathbf{bind}\ x\ k \mapsto c & \equiv & \mathbf{fwd}\ f\ p\ k \mapsto f\ (p, (\boldsymbol{\lambda} x\,.\,c)) & \end{array}$$

Furthermore, we add a typing rule for typing **fwd** clauses, and a new rule for typing handlers containing **fwd** clauses.

$$\frac{\begin{array}{c} A_p = \alpha \to M\ \beta!\langle E\rangle \\ A'_p = \alpha \to \gamma!\langle E\rangle \qquad A_k = \beta \to M\ A!\langle E\rangle \qquad A'_k = \gamma \to \delta!\langle E\rangle \\ \Gamma, \alpha, \beta, p : A_p, k : A_k, f : \forall\ \gamma\ \delta\,.\,(A'_p, A'_k) \to \delta!\langle E\rangle \vdash c_f : M\ A!\langle E\rangle \end{array}}{\Gamma \vdash \mathbf{fwd}\ f\ p\ k \mapsto c_f : M\ A!\langle E\rangle}\ \text{T-Fwd}$$

$$\text{T-Handler}\ \frac{\begin{array}{cc} \langle F\rangle \equiv_{\langle\rangle} \langle \mathit{labels}\,(\mathit{oprs})\,; E\rangle & \Gamma, \alpha \vdash \mathbf{return}\ x \mapsto c_r : M\ \alpha!\langle E\rangle \\ \Gamma, \alpha \vdash \mathit{oprs} : M\ \alpha!\langle E\rangle & \Gamma, \alpha \vdash \mathbf{fwd}\ f\ p\ k \mapsto c_f : M\ \alpha!\langle E\rangle \end{array}}{\Gamma \vdash \mathbf{handler}\ \{\mathbf{return}\ x \mapsto c_r, \mathit{oprs}, \mathbf{fwd}\ f\ p\ k \mapsto c_f\,\} : \forall\ \alpha\,.\,\alpha!\langle F\rangle \Rightarrow M\ \alpha!\langle E\rangle}$$

Rule E-Bind and T-Handler$_{\text{Bind}}$ can be derived (and are therefore superseded by) E-Fwd and T-Handler.

7.4. h_{inc}**, revisited again.** Using generalized forwarding clauses, we can implement the forwarding of h_{inc} correctly as follows.

$$\begin{array}{l} h_{\mathsf{inc}} = \mathbf{handler}\ \{\ldots, \mathbf{fwd}\ f\ p\ k \mapsto \mathbf{return}\ (\boldsymbol{\lambda} s\,.\,f\ (\boldsymbol{\lambda} y\,.\,\mathbf{do}\ p' \leftarrow p\ y\,; p'\ s, \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \boldsymbol{\lambda}(z, s')\,.\,\mathbf{do}\ k' \leftarrow k\ z\,; k'\ s'))\,\} \end{array}$$

Instead of directly returning the function p' like the **bind**-based forwarding clause, the **fwd**-based forwarding clause applies the initial state s to p' to avoid the escaping of the `choose` operation captured in p'. Then, to connect the results inside the scope with outside, we apply the continuation k to the result z and the updated state s'. The derivation is shown in Figure 8. We only get the first result of the `choose` operation.

```
with h_once handle
    do f ← with h_inc handle
      sc once () (_. op inc () (_. op choose () (b . return b))) (z . return z);
    f 0
⇝* with h_once handle
    do f ← return (λs . sc once ()
      (_. do p' ← (with h_inc handle (op inc () (_. op choose () (b . return b))));
            p' s)
      (z . ...))
    f 0
⇝* with h_once handle sc once ()
      (_. do p' ← (with h_inc handle (op inc () (_. op choose () (b . return b))));
            p' 0)
      (z . ...)
⇝* do ts ← with h_once handle (
    do p' ← (with h_inc handle (op inc () (_. op choose () (b . return b))));
    p' 0)
  if (ts = []) then return [] else do t ← head ts; ...
⇝* do ts ← with h_once handle (
    do p' ← return (λs . do s' ← s + 1;
                         do k' ← with h_inc handle op choose () (b . return b);
                         k' s')
    p' 0)
  if (ts = []) then return [] else do t ← head ts; ...
⇝* do ts ← with h_once handle (
    do s' ← 0 + 1;
    do k' ← with h_inc handle op choose () (b . return b);
    k' s')
  if (ts = []) then return [] else do t ← head ts; ...
⇝* do ts ← with h_once handle (
          op choose () (b . do k' ← return (λs . (b, s)); k' 1));
  if (ts = []) then return [] else do t ← head ts; ...
⇝* do ts ← with h_once handle (
          op choose () (b . return (b, 1)));
  if (ts = []) then return [] else do t ← head ts; ...
⇝* do ts ← return [(true, 1), (false, 1)];
  if (ts = []) then return [] else do t ← head ts; ...
⇝* [(true, 1)]
```

Figure 8: `once` staying in its scope forwarded by **fwd**

8. Metatheory

8.1. **Syntax-directed version of λ_{sc}.** Appendix D contains a syntax-directed version of λ_{sc}. We prove the type safety of λ_{sc} by connecting the typing judgement of the declarative version of λ_{sc} to those of the syntax-directed version of λ_{sc} in Appendix E. The type inference algorithm implemented in our interpreter is also built on this syntax-directed version. The syntax-directed version was obtained by the following three transformations.

First we removed rule T-Eqv, which re-types expressions to some equivalent type. This rule is used to make types line up exactly at the site of applications, for example by changing the order of the labels in effect rows. As a consequence, in the syntax directed version we essentially inline T-Eqv wherever it is needed.

Secondly, we removed the rules dealing with generalisation and instantiation (T-Inst, T-InstEff, T-Gen and T-GenEff). Instead, whenever rules insist on some kind of polymorphism on some subderivation, we extend the environment with fresh type variables, and generalize over them locally, instead of via axillary rules.

Finally, as dealing with higher-kinded polymorphism is orthogonal to our work (and real programming languages like Haskell and OCaml already have their solutions for higher-kinded polymorphism [YW14]), we avoid higher-order unification by annotating handlers with the type operator they apply (e.g. $\mathbf{handler}_M\ \{\dots\}$ instead of $\mathbf{handler}\ \{\dots\}$).

8.2. **Safety.** The type-and-effect system of λ_{sc} is type safe, which we show by proving Subject Reduction and Progress. Here we briefly state the theorems; the proofs and used lemmas can be found in Appendix E.

As values are inert, these theorems range over computations only. The formulation of Subject Reduction (Theorem 8.1) is standard. Furthermore, apart from an additional normal form $\mathbf{sc}\ \ell^{\mathsf{sc}}\ v\ (y\,.\,c_1)\ (z\,.\,c_2)$, Progress (Theorem 8.2) is standard as well. Note that Progess implies effect safety, which intuitively means that only operations tracked in the effect type can be invoked.

Theorem 8.1 (Subject Reduction)**.** *If $\Gamma \vdash c : \underline{C}$ and $c \rightsquigarrow c'$, then there exists a $\underline{C}'$ such that $\underline{C} \equiv \underline{C}'$ and $\Gamma \vdash c' : \underline{C}'$.*

Theorem 8.2 (Progress)**.** *If $\cdot \vdash c : A\,!\,\langle E\rangle$, then either:*

- *there exists a computation c' such that $c \rightsquigarrow c'$, or*
- *c is in a* normal form*, which means it is in one of the following forms: (1) $c = \mathbf{return}\ v$, (2) $c = \mathbf{op}\ \ell^{\mathsf{op}}\ v\ (y\,.\,c')$ where $\ell^{\mathsf{op}} \in E$, or (3) $c = \mathbf{sc}\ \ell^{\mathsf{sc}}\ v\ (y\,.\,c_1)\ (z\,.\,c_2)$ where $\ell^{\mathsf{sc}} \in E$.*

9. Examples & Implementation

Now that we have formalized the calculus we can cover some examples. This serves two purposes. First, we will highlight how the conventional encoding of scoped effects as handlers, the solution proposed in Plotkin and Power [PP03], is not expressive enough, even though it is applied in the real world [TRWS22]. We have postponed doing so, because now that we have formally introduced a calculus, we can immediately show how λ_{sc} addresses these issues. The first two examples in this section (exceptions with catch and reader with local) therefore contain both an attempt at encoding them as an handler, as well as a proper encoding as a **sc** in λ_{sc}. Secondly, the examples exemplify the expressivity of λ_{sc}.

Syntactic sugar. To enhance readability, we write the examples in a higher-level syntax following Eff's conventions: we use top-level definitions, coalesce values and computations, implicitly sequence steps and insert **return** where needed. Furthermore, we drop trivial **return** continuations of operations:

$$\begin{aligned} \mathbf{op}\ \ell^{\mathsf{op}}\ x &\equiv \mathbf{op}\ \ell^{\mathsf{op}}\ x\ (y\,.\,\mathbf{return}\ y) \\ \mathbf{sc}\ \ell^{\mathsf{sc}}\ x\ (y\,.\,c_1) &\equiv \mathbf{sc}\ \ell^{\mathsf{sc}}\ x\ (y\,.\,c_1)\ (z\,.\,\mathbf{return}\ z) \end{aligned}$$

Implementation. A prototype implementation of an interpreter for λ_{sc} is available at `https://github.com/thwfhk/lambdaSC`. This implementation contains a Hindley-Milner [Mil78] type inference algorithm for λ_{sc}, which is mostly an extension of the type inference of Koka [Lei14]. There are two main things that we need to additionally deal with for scoped effects. One is that we explicit force all handlers to be polymorphic. Note that there is no requirement for undecidable polymorphic recursion; deep handlers are implicitly recursive and always given polymorphic types. The other is that we require handlers to be annotated with the type operators that they use for their carriers in order to avoid higher-order unification. With explicit annotations for type operators, our unification algorithm just needs to reduce types first before unifying. The details can be found in the code.

Each of the examples covered in the rest of this section have been implemented. See the readme of the implementation's repository for more information.

9.1. **Exceptions.** Wu et al. [WSH14] have shown how to catch exceptions with a scoped operation. Raising an exception is an algebraic operation $\mathtt{raise} : \mathsf{String} \twoheadrightarrow \mathsf{Empty}$, and catching an exception is a scoped operation $\mathtt{catch} : \mathsf{String} \twoheadrightarrow \mathsf{Bool}$. For example, consider that we are dealing with a counter with a maximum value of 10. The following computation increases the counter by 1 and raises an exception when the counter exceeds 10:

$$\begin{aligned} &\mathit{incr} = \mathbf{do}\ x \leftarrow \mathbf{op}\ \mathtt{inc}\ ()\,; \\ &\qquad \mathbf{if}\ x > 10\ \mathbf{then}\ \mathbf{op}\ \mathtt{raise}\ \mathtt{"Overflow"}\ (y\,.\,\mathsf{absurd}\ y)\ \mathbf{else}\ \mathbf{return}\ x \end{aligned}$$

Clearly, if we start with a state of 8 and call `inc` thrice, we end up with an exception. We want to define a `catch` operation that executes an alternative computation when an exception is thrown inside its scope.

9.1.1. *Catch as handler.* One might attempt to write catch as a handler. However, as we will see, this method does not have the same modularity and expressivity as our calculus because it cannot achieve the local update semantics [WSH14].

$$\begin{aligned} &h_{\mathsf{except}\text{✗}} = \mathbf{handler}\ \{\mathbf{return}\ x \mapsto \mathsf{right}\ x, \mathbf{op}\ \mathtt{raise}\ e\ \_ \mapsto \mathsf{left}\ e\} \\ &\mathtt{catch}\text{✗}\ c_1\ c_2 \equiv\ \mathbf{with\ handler}\ \{\mathbf{return}\ x \qquad \mapsto \mathbf{return}\ x \\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad , \mathbf{op}\ \mathtt{raise}\ \_\ \_ \mapsto c_2 \qquad\quad \}\ \mathbf{handle}\ c_1 \\ &c_{\mathtt{catch}\text{✗}} = \mathbf{do}\ \mathit{incr}\,; \mathtt{catch}\text{✗}\ (\mathbf{do}\ \mathit{incr}\,; \mathbf{do}\ \mathit{incr}\,; \mathbf{return}\ \mathtt{"success"}) \\ &\qquad\qquad\qquad\qquad\qquad\qquad (\mathbf{return}\ \mathtt{"fail"}) \end{aligned}$$

The $\mathtt{catch}\text{✗}$ implements the `catch` operation as a handler. The $h_{\mathsf{except}\text{✗}}$ interprets exceptions using a sum type. It is used to handle potential exceptions not captured by $\mathtt{catch}\text{✗}$. By handling exceptions before state we obtain global update semantics where the state updates are not discarded when an error is raised:

$$run_{\mathsf{inc}}\ 8\ (\mathbf{with}\ h_{\mathsf{except}\text{✗}}\ \mathbf{handle}\ c_{\mathsf{catch}\text{✗}}) \rightsquigarrow^* (\mathsf{right}\ \texttt{"fail"}, 11)$$

When handling exceptions *after* state, we would expect *local* update semantics, where we discard the state updates when an error is raised. Thus, the expected result should be $\mathsf{right}\ (\texttt{"fail"}, 9)$. However, we again get the global update semantics:

$$\mathbf{with}\ h_{\mathsf{except}\text{✗}}\ \mathbf{handle}\ (run_{\mathsf{inc}}\ 8\ c_{\mathsf{catch}\text{✗}}) \rightsquigarrow^* (\mathsf{right}\ \texttt{"fail"}, 11)$$

How can this be? By implementing $\texttt{catch}_{\text{✗}}$ as a handler, we have lost the separation between syntax and semantics: $\texttt{catch}_{\text{✗}}$ is supposed to denote syntax, but it contains semantics in the form of a handler. Since we apply $\texttt{catch}_{\text{✗}}$ to a computation ($c_{\mathsf{catch}\text{✗}}$), any containing `raise` will have already been handled by $\texttt{catch}_{\text{✗}}$ before h_{inc} is applied. In other words, we have lost modular composition because the handler of `raise` always come before the handler of `inc`. As a result, we cannot change the semantics of the interactions of effects by swaping the order of their handlers, which is certainly harmful to expressivity.

9.1.2. *Catch as scoped effect.* Let us implement `catch` as a scoped operation in λ_{sc}.

$$\begin{aligned} c_{\mathsf{catch}} = \mathbf{do}\ &incr\,;\mathbf{sc}\ \texttt{catch}\ \texttt{"Overflow"}\ (b\,.\\ &\mathbf{if}\ b\ \mathbf{then}\ (\mathbf{do}\ incr\,;\mathbf{do}\ incr\,;\mathbf{return}\ \texttt{"success"})\\ &\mathbf{else\ return}\ \texttt{"fail"}) \end{aligned}$$

The scoped computation's true branch is the program that may *raise* exceptions, while the false branch *deals with* the exception. Our handler interprets exceptions in terms of a sum type $\mathbf{data}\ \alpha + \beta = \mathsf{left}\ \alpha \mid \mathsf{right}\ \beta$, where $\mathsf{left}\ v$ denotes an exception and $\mathsf{right}\ v$ a result.

$$\begin{aligned} &h_{\mathsf{except}} : \forall\ \alpha\ \mu\,.\,\alpha\,!\,\langle \texttt{raise}\,;\texttt{catch}\,;\mu\rangle \Rightarrow \mathsf{String} + \alpha\,!\,\langle\mu\rangle\\ &h_{\mathsf{except}} = \mathbf{handler}\\ &\quad\{\mathbf{return}\ x \qquad\quad \mapsto \mathsf{right}\ x\\ &\quad,\mathbf{op}\ \texttt{raise}\ e\ \_ \ \mapsto \mathsf{left}\ e\\ &\quad,\mathbf{sc}\ \texttt{catch}\ e\ p\ k \mapsto \mathbf{do}\ x \leftarrow p\ \mathsf{true}\,;\\ &\qquad\qquad\qquad\qquad \mathbf{case}\ x\ \mathbf{of}\ \mathsf{left}\ e' \mid e' = e \rightarrow \mathsf{exceptMap}\ (p\ \mathsf{false})\ k\\ &\qquad\qquad\qquad\qquad\qquad\qquad\quad \_ \qquad\qquad\ \rightarrow \mathsf{exceptMap}\ x\ k\\ &\quad,\mathbf{bind}\ x\ k \qquad\quad \mapsto \mathsf{exceptMap}\ x\ k\} \end{aligned}$$

The return clause and algebraic operation clause for `raise` construct a return value and raise an exception e by calling the right and left constructors, respectively. The scoped operation clause for `catch` catches an exception e. If the scoped computation in $p\ \mathsf{true}$ raises an exception e, it is caught by `catch` and replaced by the scoped computation ($p\ \mathsf{false}$). Otherwise, it continues with $p\ \mathsf{true}$ and its results are passed to the continuation k. The forwarding clause follows the same principle as that of how we write the forwarding clause for h_{inc} in Section 3.3.2. We just need to think about how to continue with k when we have the result x that potentially fails. The most intuitive way is to return the exception if x fails ($\mathsf{left}\ e$), and we run the continuation k with the result if x succeeds ($\mathsf{right}\ y$).

$$\begin{aligned} &\mathsf{exceptMap} : \forall\ \alpha\ \beta\ \mu\,.\,\mathsf{String} + \beta \rightarrow^{\mu} (\beta \rightarrow^{\mu} \mathsf{String} + \alpha) \rightarrow^{\mu} \mathsf{String} + \alpha\\ &\mathsf{exceptMap}\ x\ k = \mathbf{case}\ x\ \mathbf{of}\ \mathsf{left}\ e \ \rightarrow \mathsf{left}\ e\\ &\qquad\qquad\qquad\qquad\qquad\quad \mathsf{right}\ y \rightarrow k\ y \end{aligned}$$

Given an initial counter value 8, we can handle the program c_{catch} with h_{except} and h_{inc}. Different orders of the application of handlers give us different semantics of the interaction of effects [WSH14]. Handling exceptions before increments gives us global updates:

$run_{\mathsf{inc}}\ 8\ (\mathbf{with}\ h_{\mathsf{except}}\ \mathbf{handle}\ c_{\mathsf{catch}}) \rightsquigarrow^* (\mathsf{right}\ \texttt{"fail"}, 11)$

Although an exception is raised and caught, the final value is still updated to 11 by the two `inc` operations and exceeds the maximum value of our counter. When handling exceptions after increments, we obtain the expected local update semantics:

$\mathbf{with}\ h_{\mathsf{except}}\ \mathbf{handle}\ (run_{\mathsf{inc}}\ 8\ c_{\mathsf{catch}}) \rightsquigarrow^* \mathsf{right}\ (\texttt{"fail"}, 9)$

The state updates introduced by the two invocations of `inc` inside the scope of `catch` are discarded. This correctly reflects our intuition when the handler of `inc` comes before the handler of `raise`.

9.2. **Reader with Local.** Reader entails an `ask` operation that lets one read the (integer) state that is passed around. The scoped effect `local` takes a function f which alters the state, and a computation for which the state should be altered, after which the state should be returned to its original state.[3] For example, in $\mathbf{sc}\ \texttt{local}\ (\boldsymbol{\lambda} i\,.\,i * 2)\ (\mathbf{op}\ \texttt{ask}\ ())\ (\mathbf{op}\ \texttt{ask}\ ())$, the first ask receives a state that is doubled, whereas the second ask receives the original state. To exemplify the problems that arise when implementing `local` as a handler, our example uses effect `foo`, which is simply mapped to `ask` by h_{foo}:

$$\begin{aligned} h_{\mathsf{foo}} = \mathbf{handler}\ \{&\mathbf{return}\ x \ \ \mapsto \mathbf{return}\ x \\ ,&\mathbf{op}\ \texttt{foo}\ \_\ k \mapsto \mathbf{do}\ x \leftarrow \mathbf{op}\ \texttt{ask}\ ()\ (y\,.\,k\ y) \\ ,&\mathbf{bind}\ x\ k \ \ \mapsto k\ x\ \} \end{aligned}$$

9.2.1. *Local as a handler.* Whereas the lack of effect interaction control in example of `catch` as a handler could be described as unfortunate, in the case for ask there is arguably only one correct interaction, which is not the one that arises from scoped effects as handlers. Consider c_{local} below, which includes `foo`, which is mapped to `ask` by h_{foo}.

$$\begin{aligned} &\texttt{local}\boldsymbol{\chi}\ f\ c \ \equiv\ \mathbf{with\ handler}\ \{\mathbf{return}\ x \mapsto \mathbf{return}\ x \\ &\qquad\qquad\qquad\qquad\qquad\qquad ,\mathbf{op}\ \texttt{ask}\ \_ \mapsto x \leftarrow \texttt{ask}\,;f\ x\}\ \mathbf{handle}\ c \\ &h_{\mathsf{read}\boldsymbol{\chi}} = \mathbf{handler}\ \{\mathbf{return}\ x \mapsto \boldsymbol{\lambda} s\,.\,x, \mathbf{op}\ \texttt{ask}\ \_\ k \mapsto \boldsymbol{\lambda} s\,.\,k\ s\ s\} \\ &run_{\mathsf{read}\boldsymbol{\chi}}\ s\ c \ \equiv\ \mathbf{do}\ c' \leftarrow \mathbf{with}\ h_{\mathsf{read}\boldsymbol{\chi}}\ \mathbf{handle}\ c\,;c'\ s \\ &c_{\mathsf{local}\boldsymbol{\chi}} = \mathbf{do}\ x \leftarrow \mathbf{op}\ \texttt{ask}\ ()\,;\mathbf{do}\ y \leftarrow \mathbf{op}\ \texttt{foo}\ () \\ &\qquad\quad \texttt{local}\boldsymbol{\chi}\ (\boldsymbol{\lambda} a \to 2 * a)\ (z \leftarrow \mathbf{op}\ \texttt{ask}\ ()\,;u \leftarrow \mathbf{op}\ \texttt{foo}\ ()\,;\mathbf{return}\ (x,y,z,u)) \end{aligned}$$

Since h_{foo} introduces `ask`, we must (re)apply h_{read} after applying h_{foo}. Since `foo` is mapped to `ask`, in $c_{\mathsf{local}\boldsymbol{\chi}}$ we expect x to be equal to y, and z equal to u. Starting with the reader state set to 1, we expect the result $(1,1,2,2)$. Instead, we get:

$run_{\mathsf{read}}\ 1\ (\mathbf{with}\ h_{\mathsf{foo}}\ \mathbf{handle}\ c_{\mathsf{local}\boldsymbol{\chi}}) \rightsquigarrow^* \mathbf{return}\ (1,1,2,1)$

Again, how can this be? The cause is the same as the example with `catch`: since we encode the semantics of $\texttt{local}\boldsymbol{\chi}$ in its definition, we are forced to perform the handling at the moment of application. Notice that `foo` is not caught by $\texttt{local}\boldsymbol{\chi}$! Therefore, f is only applied to `ask`. When `foo` is mapped to `ask` by h_{foo}, $\texttt{local}\boldsymbol{\chi}$'s effect will already have been triggered, which is why f is not applied to it.

[3]The operation signature of `local` requires polymorphic parameter types like $\texttt{local}\!:\!(\forall\,\mu\,.\,\mathsf{Int} \to^{\mu} \mathsf{Int}) \twoheadrightarrow ()$, which we do not support. It is easy to extend operations in λ_{sc} with prenex polymorphic parameter types without any need of other mechanism for higher-rank polymorphism.

9.2.2. *Local as a scoped effect.* Using a scoped effect we can properly encode `local`:

$$
\begin{aligned}
&h_{\mathsf{read}} \;:\; \forall\, \alpha\, \mu\, .\, \alpha\, !\, \langle \mathtt{ask}\,;\mathtt{local}\,;\mu\rangle \Rightarrow (\mathsf{Int} \to^{\mu} \alpha)\, !\, \langle \mu \rangle \\
&h_{\mathsf{read}} = \mathbf{handler}\, \{\mathbf{return}\ x \qquad\quad \mapsto \boldsymbol{\lambda} s\, .\, x \\
&\qquad\qquad\quad ,\mathbf{op}\ \mathtt{ask}\ \_\ k \quad \mapsto \boldsymbol{\lambda} s\, .\, k\ s\ s \\
&\qquad\qquad\quad ,\mathbf{sc}\ \mathtt{local}\ f\ p\ k \mapsto \boldsymbol{\lambda} s\, .\, \mathbf{do}\ x \leftarrow p\ ()\ (f\ s)\,; k\ x\ s \\
&\qquad\qquad\quad ,\mathbf{fwd}\ f\ p\ k \quad \mapsto \boldsymbol{\lambda} s\, .\, f\ (\boldsymbol{\lambda} y\, .\, p\ y\ s, \boldsymbol{\lambda} z\, .\, k\ z\ s)\} \\
&\mathit{run}_{\mathsf{read}}\ s\ c \;\equiv\; \mathbf{do}\ c' \leftarrow \mathbf{with}\ h_{\mathsf{read}}\ \mathbf{handle}\ c\,; c'\ s \\
&c_{\mathsf{local}} = \mathbf{do}\ x \leftarrow \mathbf{op}\ \mathtt{ask}\ ()\,; \mathbf{do}\ y \leftarrow \mathbf{op}\ \mathtt{foo}\ ()\,; \\
&\qquad\quad \mathbf{sc}\ \mathtt{local}\ (\boldsymbol{\lambda} a \to 2 * a) \\
&\qquad\qquad\qquad (\mathbf{do}\ z \leftarrow \mathbf{op}\ \mathtt{ask}\ ()\,; \mathbf{do}\ u \leftarrow \mathbf{op}\ \mathtt{foo}\ ()\,; \mathbf{return}\ (x, y, z, u))
\end{aligned}
$$

Note that the forwarding clause of h_{read} is similar to that of h_{inc} in Section 7.4 except for passing the same state to the scoped computation p and continuation k. This makes natural sense because we definitely do not want the scopes of other irrelevant operations to change what can be read. Since `local` is now purely syntactic, we can apply h_{foo} before h_{read}, and have h_{read} handle the `ask` that h_{foo} outputs:

$$
\begin{aligned}
&\quad \mathit{run}_{\mathsf{read}}\ 1\ (\mathbf{with}\ h_{\mathsf{foo}}\ \mathbf{handle}\ c_{\mathsf{local}}) \\
&\rightsquigarrow \mathit{run}_{\mathsf{read}}\ 1\ (x \leftarrow \mathbf{op}\ \mathtt{ask}\ ()\,; y \leftarrow \mathbf{op}\ \mathtt{ask}\ ()\,; \\
&\qquad\qquad \mathbf{sc}\ \mathtt{local}\ (\boldsymbol{\lambda} a \to 2 * a) \\
&\qquad\qquad\qquad (\mathbf{do}\ z \leftarrow \mathbf{op}\ \mathtt{ask}\ ()\,; \mathbf{do}\ u \leftarrow \mathbf{op}\ \mathtt{ask}\ ()\,; \mathbf{return}\ (x, y, z, u))) \\
&\rightsquigarrow \mathbf{return}\ (1, 1, 2, 2)
\end{aligned}
$$

9.3. **Nondeterminism with Cut.** The algebraic operation $\mathtt{cut} : () \twoheadrightarrow ()$ provides a different flavor of pruning nondeterminism that has its origin as a Prolog primitive. The idea is that `cut` prunes all remaining branches and only allows the current branch to continue. Typically, we want to keep the effect of `cut` local. This is achieved with the scoped operation $\mathtt{call} : () \twoheadrightarrow ()$, as proposed by Wu et al. [WSH14]. To handle `cut` and `call`, we use the CutList datatype [PS17].

$$\mathbf{data}\ \mathsf{CutList}\ \alpha = \mathsf{opened}\ (\mathsf{List}\ \alpha)\ \mid\ \mathsf{closed}\ (\mathsf{List}\ \alpha)$$

We can think of opened v as a list that may be extended and closed v as a list that may not be extended with further elements. This intention is captured in the $\mathsf{append}_{\mathsf{CutList}}$ function, which discards the second list if the constructor of the first list is closed.

$$
\begin{aligned}
&\mathsf{append}_{\mathsf{CutList}} : \forall\, \alpha\, \mu\, .\, \mathsf{CutList}\ \alpha \to^{\mu} \mathsf{CutList}\ \alpha \to^{\mu} \mathsf{CutList}\ \alpha \\
&\mathsf{append}_{\mathsf{CutList}}\ (\mathsf{opened}\ xs)\ (\mathsf{opened}\ ys) = \mathsf{opened}\ (xs +\!\!+ ys) \\
&\mathsf{append}_{\mathsf{CutList}}\ (\mathsf{opened}\ xs)\ (\mathsf{closed}\ ys)\ \ = \mathsf{closed}\ \ (xs +\!\!+ ys) \\
&\mathsf{append}_{\mathsf{CutList}}\ (\mathsf{closed}\ xs)\ \ \_\qquad\qquad\ = \mathsf{closed}\ \ xs
\end{aligned}
$$

The handler for nondeterminism with cut is defined as follows:

$$
\begin{aligned}
&h_{\mathsf{cut}} \;:\; \forall\, \alpha\, \mu\, .\, \alpha\, !\, \langle \mathtt{choose}\,;\mathtt{fail}\,;\mathtt{cut}\,;\mathtt{call}\,;\mu\rangle \Rightarrow \mathsf{CutList}\ \alpha\, !\, \langle \mu \rangle \\
&h_{\mathsf{cut}} = \mathbf{handler}\, \{\mathbf{return}\ x \qquad\quad \mapsto \mathsf{opened}\ [x] \\
&\qquad\qquad\quad ,\mathbf{op}\ \mathtt{fail}\ \_\ \_ \quad \mapsto \mathsf{opened}\ [\,] \\
&\qquad\qquad\quad ,\mathbf{op}\ \mathtt{choose}\ x\ k \mapsto \mathbf{do}\ xs \leftarrow k\ \mathsf{true}\,; \\
&\qquad\qquad\qquad\qquad\qquad\quad \mathbf{if}\ \mathsf{isclose}\ xs\ \mathbf{then}\ xs\ \mathbf{else}\ \mathsf{append}_{\mathsf{CutList}}\ xs\ (k\ \mathsf{false}) \\
&\qquad\qquad\quad ,\mathbf{op}\ \mathtt{cut}\ \_\ k \quad \mapsto \mathsf{close}\ (k\ ())
\end{aligned}
$$

$$
\begin{array}{ll}
, \mathbf{sc}\ \texttt{call}\ \_\ p\ k & \mapsto \mathsf{concatMap}_{\mathsf{CutList}}\ (\mathsf{open}\ (p\ ()))\ k \\
, \mathbf{bind}\ x\ k & \mapsto \mathsf{concatMap}_{\mathsf{CutList}}\ x\ k\,\}
\end{array}
$$

The operation clause for `cut` closes the cutlist and the clause for `call` (re-)opens it when coming out of the scope.

$$
\begin{array}{ll}
\mathsf{close} : \forall\ \alpha\ \mu\,.\,\mathsf{CutList}\ \alpha \to^{\mu} \mathsf{CutList}\ \alpha & \mathsf{open} : \forall\ \alpha\ \mu\,.\,\mathsf{CutList}\ \alpha \to^{\mu} \mathsf{CutList}\ \alpha \\
\mathsf{close}\ (\mathsf{closed}\ as) = \mathsf{closed}\ as & \mathsf{open}\ (\mathsf{closed}\ as) = \mathsf{opened}\ as \\
\mathsf{close}\ (\mathsf{opened}\ as) = \mathsf{closed}\ as & \mathsf{open}\ (\mathsf{opened}\ as) = \mathsf{opened}\ as
\end{array}
$$

The function isclose checks whether a cutlist is closed. It is used in the `choose` clause for efficiency; we do not need to execute k false when k true returns a closed list.

$$
\begin{array}{l}
\mathsf{isclose} : \forall\ \alpha\ \mu\,.\,\mathsf{CutList}\ \alpha \to^{\mu} \mathsf{CutList}\ \alpha \\
\mathsf{isclose}\ (\mathsf{closed}\ \_) = \mathsf{true} \\
\mathsf{isclose}\ (\mathsf{opened}\ \_) = \mathsf{false}
\end{array}
$$

The forwarding of h_{cut} uses the function $\mathsf{concatMap}_{\mathsf{CutList}}$, the cutlist counterpart of concatMap which takes the extensibility of CutList (signalled by opened and closed) into account when concatenating.

$$
\begin{array}{l}
\mathsf{concatMap}_{\mathsf{CutList}} : \forall\ \alpha\ \beta\ \mu\,.\,\mathsf{CutList}\ \beta \to^{\mu} (\beta \to^{\mu} \mathsf{CutList}\ \alpha) \to^{\mu} \mathsf{CutList}\ \alpha \\
\mathsf{concatMap}_{\mathsf{CutList}}\ (\mathsf{opened}\ [\,])\ f = \mathbf{return}\ (\mathsf{opened}\ [\,]) \\
\mathsf{concatMap}_{\mathsf{CutList}}\ (\mathsf{closed}\ [\,])\ f = \mathbf{return}\ (\mathsf{closed}\ [\,]) \\
\mathsf{concatMap}_{\mathsf{CutList}}\ (\mathsf{opened}\ (b : bs))\ f = \mathbf{do}\ as \leftarrow f\ b\,; \\
\qquad\qquad as' \leftarrow \mathsf{concatMap}_{\mathsf{CutList}}\ (\mathsf{opened}\ bs)\ f\,; \\
\qquad\qquad \mathsf{append}_{\mathsf{CutList}}\ as\ as' \\
\mathsf{concatMap}_{\mathsf{CutList}}\ (\mathsf{closed}\ (b : bs))\ f = \mathbf{do}\ as \leftarrow f\ b\,; \\
\qquad\qquad as' \leftarrow \mathsf{concatMap}_{\mathsf{CutList}}\ (\mathsf{closed}\ bs)\ f\,; \\
\qquad\qquad \mathsf{append}_{\mathsf{CutList}}\ as\ as'
\end{array}
$$

In Section 9.5, we give an example usage of `cut` to improve parsers.

9.4. **Depth-Bounded Search.** The handler h_{ND} for nondeterminism shown in Section 2 implements the *depth-first search* (DFS) strategy. However, with scoped effects and handlers we can implement other search strategies, such as *depth-bounded search* (DBS) [YPW+22], which uses the scoped operation $\texttt{depth} : \mathsf{Int} \twoheadrightarrow ()$ to bound the depth of the branches in the scoped computation. The handler uses return type $\mathsf{Int} \to^{\mu} \mathsf{List}\ (\alpha, \mathsf{Int})$. Here, the Int parameter is the current depth bound, and the result is a list of (α, Int) pairs, where α denotes the result and Int reflects the remaining global depth bound.[4]

$$
\begin{array}{ll}
h_{\mathsf{depth}} & : \forall\ \alpha\ \mu\,.\,\alpha\,!\,\langle \texttt{choose}\,;\texttt{fail}\,;\texttt{depth}\,;\mu\rangle \Rightarrow (\mathsf{Int} \to^{\mu} \mathsf{List}\ (\alpha, \mathsf{Int}))\,!\,\langle\mu\rangle \\
h_{\mathsf{depth}} & = \mathbf{handler} \\
& \quad \{\mathbf{return}\ x \qquad \mapsto \boldsymbol{\lambda} d\,.\,[(x, d)] \\
& \quad , \mathbf{op}\ \texttt{fail}\ \_\ \_ \qquad \mapsto \boldsymbol{\lambda}\_\,.\,[\,] \\
& \quad , \mathbf{op}\ \texttt{choose}\ x\ k \mapsto \boldsymbol{\lambda} d\,.\,\mathbf{if}\ d = 0\ \mathbf{then}\ [\,]\ \mathbf{else}\ k\ \mathsf{true}\ (d-1)\ {+\!\!+} k\ \mathsf{false}\ (d-1) \\
& \quad , \mathbf{sc}\ \texttt{depth}\ d'\ p\ k \mapsto \boldsymbol{\lambda} d\,.\,\mathsf{concatMap}\ (p\ ()\ d')\ (\boldsymbol{\lambda}(v, \_)\,.\,k\ v\ d) \\
& \quad , \mathbf{fwd}\ f\ p\ k \qquad \mapsto \boldsymbol{\lambda} d\,.\,f\ (\boldsymbol{\lambda} y\,.\,p\ y\ d, \boldsymbol{\lambda} vs\,.\,\mathsf{concatMap}\ vs\ (\boldsymbol{\lambda}(v, d)\,.\,k\ v\ d))\}
\end{array}
$$

[4]The pair type (α, Int) differs from Yang et al. [YPW+22]'s α in order to enable forwarding.

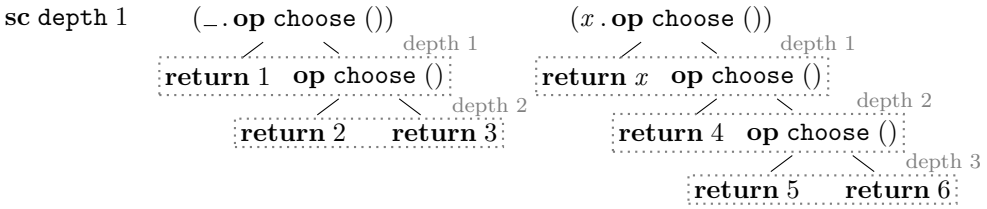

Figure 9: Visual representation of c_{depth}.

For the `depth` operation, we locally use the given depth bound d' for the scoped computation p and go back to using the global depth bound d for the continuation k. In case of an unknown scoped operation, the forwarding clause just threads the depth bound through, first into the scoped computation and from there into the continuation. It is similar to a combination of the forwarding of h_{once} and h_{inc}.

For example, the following program (Figure 9) has a local depth bound of 1 and a global depth bound of 2. It discards the results 2 and 3 in the scoped computation as they appear after the second `choose` operation, and similarly, the results 5 and 6 in the continuation are ignored.

$$
\begin{aligned}
&c_{\mathsf{depth}} = \mathbf{sc}\ \mathtt{depth}\ 1 \\
&\qquad (\_ .\, \mathbf{do}\ b_1 \leftarrow \mathbf{op}\ \mathtt{choose}\ ()\,;\mathbf{if}\ b_1\ \mathbf{then\ return}\ 1\ \mathbf{else} \\
&\qquad\quad \mathbf{do}\ b_2 \leftarrow \mathbf{op}\ \mathtt{choose}\ ()\,;\mathbf{if}\ b_2\ \mathbf{then\ return}\ 2\ \mathbf{else\ return}\ 3) \\
&\qquad (x .\, \mathbf{do}\ b_1 \leftarrow \mathbf{op}\ \mathtt{choose}\ ()\,;\mathbf{if}\ b_1\ \mathbf{then\ return}\ x\ \mathbf{else} \\
&\qquad\quad \mathbf{do}\ b_2 \leftarrow \mathbf{op}\ \mathtt{choose}\ ()\,;\mathbf{if}\ b_2\ \mathbf{then\ return}\ 4\ \mathbf{else} \\
&\qquad\quad \mathbf{do}\ b_3 \leftarrow \mathbf{op}\ \mathtt{choose}\ ()\,;\mathbf{if}\ b_3\ \mathbf{then\ return}\ 5\ \mathbf{else\ return}\ 6) \\
&\texttt{>>>}\ (\mathbf{with}\ h_{\mathsf{depth}}\ \mathbf{handle}\ c_{\mathsf{depth}})\ 2 \\
&[(1,1),(4,0)]
\end{aligned}
$$

The result is $[(1,1),(4,0)]$, where the tuple's second parameter represents the global depth bound. Notice that `choose` operations in the scoped computation `depth` do not consume the global depth bound in the handler. We also show an alternative implementation of the handler clause for `depth` in the Supplementary Material.

9.5. **Parsers.** A parser effect can be achieved by combining the nondeterminism-with-cut effect and a token-consuming effect [WSH14]. The latter features the algebraic operation $\mathtt{token} : \mathsf{Char} \twoheadrightarrow \mathsf{Char}$ where $\mathbf{op}\ \mathtt{token}\ t$ consumes a single character from the implicit input string; if it is t, it is passed on to the continuation; otherwise the operation fails. The token handler has result type $\mathsf{String} \to^{\langle \mathtt{fail};\mu\rangle} (\alpha, \mathsf{String})$: it threads through the remaining part of the input string. Observe that the function type signals it may `fail`, in case the token does not match.

$$
\begin{aligned}
&h_{\mathsf{token}} \ :\ \forall\ \alpha\ \mu .\, \alpha\,!\,\langle \mathtt{token}\,;\mathtt{fail}\,;\mu\rangle \Rightarrow (\mathsf{String} \to^{\langle \mathtt{fail};\mu\rangle} (\alpha, \mathsf{String}))\,!\,\langle \mathtt{fail}\,;\mu\rangle \\
&h_{\mathsf{token}} = \mathbf{handler} \\
&\quad \{\mathbf{return}\ x \qquad \mapsto \boldsymbol{\lambda} s .\, (x, s) \\
&\quad , \mathbf{op}\ \mathtt{token}\ x\ k \mapsto \boldsymbol{\lambda} s .\, \mathbf{case}\ s\ \mathbf{of}\ [\,] \qquad\quad \to \mathsf{failure}\ () \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad (x' : xs) \to \mathbf{if}\ x = x'\ \mathbf{then}\ k\ x\ xs\ \mathbf{else}\ \mathsf{failure}\ () \\
&\quad , \mathbf{fwd}\ f\ p\ k \qquad \mapsto \boldsymbol{\lambda} s .\, f\ (\boldsymbol{\lambda} y .\, p\ y\ s, \boldsymbol{\lambda}(t, s') .\, k\ t\ s')\}
\end{aligned}
$$

The forwarding clause of h_{token} is the same as that of h_{inc} in Section 7.4. We uses the initial state s for the scoped computation p, and the updated state s' for the continuation k. The updated state s' is passed in from the scoped computation p and reflects the consumption of tokens in p. An alternative forwarding semantics is $\mathbf{fwd}\ f\ p\ k \mapsto \boldsymbol{\lambda} s\,.\,f\ (\boldsymbol{\lambda} y\,.\,p\ y\ s, \boldsymbol{\lambda}(t, s')\,.\,k\ t\ s)$, where the continuation k also uses the initial state s. In this way, we actually discard all the changes to the state in the scoped computation, which could potentially be useful to implement parser lookahead. It is up to the programmer to decide which semantics they want.

We give an example parser for a small expression language, in the typical parser combinator style, built on top of the token-consumer and nondeterminism. For convenience, it uses the syntactic sugar $x \diamond y \equiv \mathbf{op}\ \mathtt{choose}\ (b\,.\,\mathbf{if}\ b\ \mathbf{then}\ x\ \mathbf{else}\ y)$.

```
digit    : ∀ μ . () → Char ! ⟨token ; choose ; μ⟩
digit _  = op token '0' ◇ op token '1' ◇ ... ◇ op token '9'
many1    : ∀ α μ . (() →μ α) →μ List α
many1 p  = do a ← p () ; do as ← many1 p ◇ return [] ; return (a : as)
expr'    : ∀ μ . () → Int ! ⟨token ; choose ; μ⟩
expr' _  = (do i ← term' () ; do op token '+' ; do j ← expr' () ; return (i + j))
         ◇ (do i ← term' () ; return i)
term'    : ∀ μ . () → Int ! ⟨token ; choose ; μ⟩
term' _  = (do i ← factor () ; do op token '*' ; do j ← term () ; return (i * j))
         ◇ (do i ← factor () ; return i)
factor   : ∀ μ . () → Int ! ⟨token ; choose ; μ⟩
factor _ = (do ds ← many1 digit ; return (read ds))
         ◇ (do op token '(' ; do i ← expr' () ; do op token ')' ; return i)
```

The expr′ and term′ parsers are naive and can be improved by two steps of refactoring: (1) factoring out the common prefix in the two branches, and (2) pruning the second branch when the first branch successfully consumes a + or ∗, respectively. The optimised versions of expr, term, and factor are shown as follows using the cut operation defined in Section 9.3.

```
expr   : ∀ μ . () → Int ! ⟨token ; choose ; cut ; μ⟩
expr _ = do i ← term () ;
           sc call () (_ . (do op token '+' ; do op cut () ;
                               do j ← expr () ; return (i + j)) ◇ i)
term   : ∀ μ . () → Int ! ⟨token ; choose ; cut ; μ⟩
term _ = do i ← factor () ;
           sc call () (_ . (do op token '*' ; do op cut () ;
                               do j ← term () ; return (i * j)) ◇ i)
```

Here is how we invoke the optimised parser on an example input.

```
>>> with h_cut handle (with h_token handle expr ()) "(2+5)*8"
opened [(56, "")]
```

Note that only the fully parsed result is returned because in expr and term we prune other branches when the first branch succeeds. If we parse the same input using the unoptimised parser expr′, we also get partially parsed results.

>>> **with** h_{cut} **handle** (**with** h_{token} **handle** expr′ ()) "(2+5)*8"
opened $[(56, \texttt{""}), (7, \texttt{"*8"})]$

10. Related Work

In this section, we discuss related work on algebraic effects, scoped effects, and effect systems.

10.1. **Algebraic Effects & Handlers.** Many research languages for algebraic effects and handlers have been proposed, including Eff [BP13, Pre15], Frank [LMM17], Effekt [BSO20], or have been extended to include them, such as Links [HL16] and Koka [Lei17]. OCaml [SDW+21] is the first industrial language supporting algebraic effects and handlers. Although it is possible to use handlers or write algebraic operations with function parameters to simulate the behaviours of scoped effects in these languages, none of them can achieve the expressiveness of real scoped effects & handlers as we have compared in Section 9.1 and Section 9.2.

There are also many packages for writing effect handlers in general purpose languages like Haskell and OCaml [KSSF19, KS18, RTWS18, Mag19, Kin19]. Yet, as far as we know, λ_{sc} is the first *calculus* that supports scoped effects & handlers.

10.2. **Effect Systems.** Most languages with support for algebraic effects are equipped with an effect system to keep track of the effects that are used in the programs. There is already much work on different approaches to effect systems for algebraic effects.

Eff [BP13, Pre15] uses an effect system based on subtyping relations. Each type of computation is decorated with an effect type Δ to represent the set of operations that might be invoked. The subtyping relations are used to extend the effect type Δ with other effects, which makes it possible to compose programs in a modular way. The implementation of such a subtyping system is orthogonal to the implementation of scoped effects, which is why we have opted for Koka-style row polymorphisms, which has yielded a simpler type system. Furthermore, supporting both polymorphism and non-trivial subtyping would complicate the type inference a lot as shown in [Pre14] and [THLM24] which use type inference with constraints and qualified types, respectively. Moreover, since λ_{sc} has type operators, we need to additionally distinguish between the positive and negative positions in type operators and propagate subtyping relations through these positions properly. This would further complicate the formalisation of type system and type inference.

Row polymorphism is another mainstream approach to effect systems. Links [HL16] uses the Rémy-style row polymorphism [Rém94], where the row types are able to represent the absence of labels and each label is restricted to appear at most once. Koka [Lei17] uses row polymorphism based on scoped labels [Lei05], which allows duplicated labels and as a result is easier to implement. We can use row polymorphism to write handlers that handle particular effects and forward other effects represented by a row variable. In λ_{sc}, we opted for an effect system similar to Koka's, mainly because of its brevity. We believe that the Links-style effect system should also work well with scoped effects.

10.3. **Scoped Effects & Handlers.** Wu et al. [WSH14] first introduced the idea of scoped effects & handlers to solve the problem of separating syntax from semantics in programming with effects that delimit the scope. They proposed a syntax based on higher-order functors, which impose fewer restrictions on the shape of the signatures of scoped operations than λ_{sc}. Their work also considers the problem of modular composition of handlers, and presents a solution–called "weaving"–based on threading a handler state through unknown operations. This approach is rather ad-hoc; it is not a generalization of the forwarding approach for algebraic effects. This approach has been adopted by several Haskell packages [RTWS18, Mag19, Kin19].

Piróg et al. [PSWJ18] and Yang et al. [YPW$^+$22] have developed denotational semantic domains of scoped effects, backed by category theoretical models. The key idea is to generalize the denotational approach of algebraic effects & handlers that is based on free monads and their unique homomorphisms. Indeed, the underlying category can be seen as a parameter. Then, by shifting from the base category of types and functions to a different (indexed or functor) category, scoped operations and their handlers turn out to be "just" an instance of the generalized notion of algebraic operations and handlers with the same structure and properties. We focus on a calculus for scoped effects instead of the denotational semantics of scoped effects. We make a simplification with respect to Yang et al. [YPW$^+$22] where we avoid duplication of the base algebra and endoalgebra (for the outer and inner scoped respectively), and thus duplication of the scoped effect clauses in our handlers. With respect to Piróg et al. [PSWJ18], we specialize the generic endofunctor Γ with signatures $A_\ell \twoheadrightarrow B_\ell$ of endofunctors of the form $A \times (B \to -)$. Our λ_{sc} calculus uses a similar idea to the 'explicit substitution' monad of Piróg et al. [PSWJ18], a generalization of Ghani and Uustalu's [GU03] monad of explicit substitutions where each operation is associated with two computations representing the computation in scope and out of the scope (continuation) respectively. Neither Piróg et al. [PSWJ18] nor Yang et al. [YPW$^+$22] considered the modular composition of scoped effects or the forwarding of operations in their categorical models.

Yang and Wu [YW23] develop a framework for (generalized) monoids with operations, of which scope effects are an instance. They study the problem of semantic modularity in this framework, and have some generic results which can be applied to scoped effects. As far as we know, no language design or library implementation of scoped effects has resulted from this yet.

Lindley et al. [LMM$^+$23] developed an equational reasoning framework for scoped effects based on parameterised algebraic theories [Sta13]. They do not consider handlers or forwarding.

11. Conclusion and Future Work

In this work, we have presented λ_{sc}, a novel calculus in which scoped effects & handlers are built-in. We started from the core calculus of Eff, extended it with a row-based effect system in the style of Koka, and added primitive support for scoped operations and their handlers. We introduced novel forwarding clauses as a means to obtain the modular composition of handlers in the presence of scoped effects. Finally, we have demonstrated the usability of λ_{sc} by implementing a range of examples. We believe that the features to support scoped effect in λ_{sc} are orthogonal to other language features and can be added to any programming language with algebraic effects, polymorphism and type operators.

Scoped effects require *every* handler in λ_{sc} to be polymorphic and equipped with an explicit forwarding clause. This breaks backwards compatibility: calculi that support only algebraic effects, such as Eff, miss an explicit forwarding clause for scoped operations and allow monomorphic handlers. This problem can be easily mitigated by kinds and kind polymorphism. The core idea is that we extend λ_{sc} with two kinds op and sc for effect types, such that $\Gamma \vdash E : \mathsf{op}$ means effect type E only contains algebraic operations, and $\Gamma \vdash E : \mathsf{sc}$ means effect type E may contain some scoped operations. Then, for handlers of type $A!\langle E\rangle \Rightarrow M\ A!\langle F\rangle$ which lack forwarding clauses, we can just add the condition $\Gamma \vdash E : \mathsf{op}$ to their typing rules. We leave the full formalisation and implementation of it to future work.

Making scoped effects and handlers into practical languages is a pretty new research area full of potential. There is a lot of future work to do. Directions for future work include: extending λ_{sc} with shallow handlers [KLO13, HL18b], named handlers [BPPS20, XCIL22, BSO20], and other forms of higher-order effects [BS24] including latent effects [BSPW21] and parallel effects [XJMP21]; exploring the notion of named scoped effects where scoped effects themselves generate fresh names for the operations in their scope in a similar style to named handlers; defining a CPS translation for scoped effects and handlers [HLA20, Lei17]; developing control-flow linearity [THLM24] for scoped effects to soundly extend λ_{sc} with linear types; exploring guiding principles and equational theories for writing and reasoning about forwarding clauses.

Acknowledgment

Part of this work was funded by FWO project G0A9423N.

References

[BP13] Andrej Bauer and Matija Pretnar. An effect system for algebraic effects and handlers. In Reiko Heckel and Stefan Milius, editors, *Algebra and Coalgebra in Computer Science - 5th International Conference, CALCO 2013, Warsaw, Poland, September 3-6, 2013. Proceedings*, volume 8089 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2013. doi:10.1007/978-3-642-40206-7_1.

[BP15] Andrej Bauer and Matija Pretnar. Programming with algebraic effects and handlers. *Journal of Logical and Algebraic Methods in Programming*, 84(1):108–123, 2015. Special Issue: The 23rd Nordic Workshop on Programming Theory (NWPT 2011) Special Issue: Domains X, International workshop on Domain Theory and applications, Swansea, 5-7 September, 2011. URL: https://www.sciencedirect.com/science/article/pii/S2352220814000194, doi:10.1016/j.jlamp.2014.02.001.

[BPPS20] Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. Binders by day, labels by night: effect instances via lexically scoped handlers. *Proc. ACM Program. Lang.*, 4(POPL):48:1–48:29, 2020. doi:10.1145/3371116.

[BS24] Birthe van den Berg and Tom Schrijvers. A framework for higher-order effects & handlers. *Science of Computer Programming*, 234:103086, 2024. URL: https://www.sciencedirect.com/science/article/pii/S0167642324000091, doi:10.1016/j.scico.2024.103086.

[BSO20] Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. Effects as capabilities: Effect handlers and lightweight effect polymorphism. *Proc. ACM Program. Lang.*, 4(OOPSLA), November 2020. doi:10.1145/3428194.

[BSPW21] Birthe van den Berg, Tom Schrijvers, Casper Bach Poulsen, and Nicolas Wu. Latent effects for reusable language components. In Hakjoo Oh, editor, *Programming Languages and Systems - 19th Asian Symposium, APLAS 2021, Chicago, IL, USA, October 17-18, 2021, Proceedings*,

volume 13008 of *Lecture Notes in Computer Science*, pages 182–201. Springer, 2021. doi:10.1007/978-3-030-89051-3_11.

[GU03] Neil Ghani and Tarmo Uustalu. Explicit substitutions and higher-order syntax. In *Proceedings of the 2003 ACM SIGPLAN Workshop on Mechanized Reasoning about Languages with Variable Binding*, MERLIN '03, page 1–7, New York, NY, USA, 2003. Association for Computing Machinery. doi:10.1145/976571.976580.

[HL16] Daniel Hillerström and Sam Lindley. Liberating effects with rows and handlers. In *Proceedings of the 1st International Workshop on Type-Driven Development*, TyDe 2016, page 15–27, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2976022.2976033.

[HL18a] Daniel Hillerström and Sam Lindley. Shallow effect handlers. In Sukyoung Ryu, editor, *Programming Languages and Systems - 16th Asian Symposium, APLAS 2018, Wellington, New Zealand, December 2-6, 2018, Proceedings*, volume 11275 of *Lecture Notes in Computer Science*, pages 415–435. Springer, 2018. doi:10.1007/978-3-030-02768-1_22.

[HL18b] Daniel Hillerström and Sam Lindley. Shallow effect handlers. In Sukyoung Ryu, editor, *Programming Languages and Systems - 16th Asian Symposium, APLAS 2018, Wellington, New Zealand, December 2-6, 2018, Proceedings*, volume 11275 of *Lecture Notes in Computer Science*, pages 415–435. Springer, 2018. doi:10.1007/978-3-030-02768-1_22.

[HLA20] Daniel Hillerström, Sam Lindley, and Robert Atkey. Effect handlers via generalised continuations. *J. Funct. Program.*, 30:e5, 2020. doi:10.1017/S0956796820000040.

[Kin19] Alexis King. eff – screaming fast extensible effects for less, 2019. https://github.com/hasura/eff.

[KLO13] Ohad Kammar, Sam Lindley, and Nicolas Oury. Handlers in action. In Greg Morrisett and Tarmo Uustalu, editors, *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, pages 145–158. ACM, 2013. doi:10.1145/2500365.2500590.

[KS18] Oleg Kiselyov and KC Sivaramakrishnan. Eff directly in OCaml. *Electronic Proceedings in Theoretical Computer Science*, 285:23–58, Dec 2018. URL: http://dx.doi.org/10.4204/EPTCS.285.2, doi:10.4204/eptcs.285.2.

[KSSF19] Oleg Kiselyov, Amr Sabry, Cameron Swords, and Ben Foppa. extensible-effects: An alternative to monad transformers, 2019. https://hackage.haskell.org/package/extensible-effects.

[Lei05] Daan Leijen. Extensible records with scoped labels. In Marko C. J. D. van Eekelen, editor, *Revised Selected Papers from the Sixth Symposium on Trends in Functional Programming, TFP 2005, Tallinn, Estonia, 23-24 September 2005*, volume 6 of *Trends in Functional Programming*, pages 179–194. Intellect, 2005.

[Lei14] Daan Leijen. Koka: Programming with row polymorphic effect types. *Electronic Proceedings in Theoretical Computer Science*, 153, 06 2014. doi:10.4204/EPTCS.153.8.

[Lei17] Daan Leijen. Type directed compilation of row-typed algebraic effects. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, page 486–499, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3009837.3009872.

[LMM17] Sam Lindley, Conor McBride, and Craig McLaughlin. Do be do be do. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, page 500–514, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3009837.3009897.

[LMM+23] Sam Lindley, Cristina Matache, Sean Moss, Sam Staton, Nicolas Wu, and Zhixuan Yang. Scoped effects as parameterized algebraic theories. In *European Symposium on Programming 2024*, Lecture Notes in Computer Science. Springer, December 2023. URL: https://etaps.org/2024/conferences/esop/.

[LPT03] Paul Blain Levy, John Power, and Hayo Thielecke. Modelling environments in call-by-value programming languages. *Inf. Comput.*, 185(2):182–210, 2003. doi:10.1016/S0890-5401(03)00088-9.

[Mag19] Sandy Maguire. polysemy: Higher-order, low-boilerplate free monads, 2019. https://hackage.haskell.org/package/polysemy.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.*, 17(3):348–375, 1978. doi:10.1016/0022-0000(78)90014-4.

[Mog89] Eugenio Moggi. An abstract view of programming languages. Technical Report ECS-LFCS-90-113, Edinburgh University, Department of Computer Science, June 1989.

[Mog91] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55 – 92, 1991. Selections from 1989 IEEE Symposium on Logic in Computer Science. doi:10.1016/0890-5401(91)90052-4.

[PP03] Gordon D. Plotkin and John Power. Algebraic operations and generic effects. *Appl. Categorical Struct.*, 11(1):69–94, 2003. doi:10.1023/A:1023064908962.

[PP09] Gordon Plotkin and Matija Pretnar. Handlers of algebraic effects. In Giuseppe Castagna, editor, *Programming Languages and Systems*, pages 80–94, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. doi:10.1007/978-3-642-00590-9_7.

[Pre14] Matija Pretnar. Inferring algebraic effects. *Log. Methods Comput. Sci.*, 10(3), 2014. doi:10.2168/LMCS-10(3:21)2014.

[Pre15] Matija Pretnar. An introduction to algebraic effects and handlers. invited tutorial paper. *Electronic Notes in Theoretical Computer Science*, 319:19–35, 2015. The 31st Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXI). URL: https://www.sciencedirect.com/science/article/pii/S1571066115000705, doi:10.1016/j.entcs.2015.12.003.

[PS17] Maciej Piróg and Sam Staton. Backtracking with cut via a distributive law and left-zero monoids. *J. Funct. Program.*, 27:e17, 2017. doi:10.1017/S0956796817000077.

[PSWJ18] Maciej Piróg, Tom Schrijvers, Nicolas Wu, and Mauro Jaskelioff. Syntax and semantics for operations with scopes. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '18, page 809–818, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3209108.3209166.

[Rém94] Didier Rémy. Type inference for records in a natural extension of ml. In *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*. Citeseer, 1994.

[RTWS18] Rob Rix, Patrick Thomson, Nicolas Wu, and Tom Schrijvers. fused-effects: A fast, flexible, fused effect system, 2018. https://hackage.haskell.org/package/fused-effects.

[SDW+21] KC Sivaramakrishnan, Stephen Dolan, Leo White, Tom Kelly, Sadiq Jaffer, and Anil Madhavapeddy. Retrofitting effect handlers onto OCaml. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, page 206–221, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3453483.3454039.

[SPWJ19] Tom Schrijvers, Maciej Piróg, Nicolas Wu, and Mauro Jaskelioff. Monad transformers and modular algebraic effects: what binds them together. In Richard A. Eisenberg, editor, *Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell, Haskell@ICFP 2019, Berlin, Germany, August 18-23, 2019*, pages 98–113. ACM, 2019. doi:10.1145/3331545.3342595.

[Sta13] Sam Staton. Instances of computational effects: An algebraic perspective. In *28th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2013, New Orleans, LA, USA, June 25-28, 2013*, page 519. IEEE Computer Society, 2013. doi:10.1109/LICS.2013.58.

[THLM24] Wenhao Tang, Daniel Hillerström, Sam Lindley, and J. Garrett Morris. Soundly handling linearity. *Proc. ACM Program. Lang.*, 8(POPL):1600–1628, 2024. doi:10.1145/3632896.

[TRWS22] Patrick Thomson, Rob Rix, Nicolas Wu, and Tom Schrijvers. Fusing industry and academia at GitHub (experience report). *Proc. ACM Program. Lang.*, 6(ICFP):496–511, 2022. doi:10.1145/3547639.

[Wad95] Philip Wadler. Monads for functional programming. In Johan Jeuring and Erik Meijer, editors, *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques, Båstad, Sweden, May 24-30, 1995, Tutorial Text*, volume 925 of *Lecture Notes in Computer Science*, pages 24–52. Springer, 1995. doi:10.1007/3-540-59451-5_2.

[WSH14] Nicolas Wu, Tom Schrijvers, and Ralf Hinze. Effect handlers in scope. In *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell*, Haskell '14, page 1–12, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2633357.2633358.

[XCIL22] Ningning Xie, Youyou Cong, Kazuki Ikemori, and Daan Leijen. First-class names for effect handlers. *Proc. ACM Program. Lang.*, 6(OOPSLA2):30–59, 2022. doi:10.1145/3563289.

[XJMP21] Ningning Xie, Daniel D. Johnson, Dougal Maclaurin, and Adam Paszke. Parallel algebraic effect handlers. *CoRR*, abs/2110.07493, 2021. URL: https://arxiv.org/abs/2110.07493, arXiv:2110.07493.

[YPW+22] Zhixuan Yang, Marco Paviotti, Nicolas Wu, Birthe van den Berg, and Tom Schrijvers. Structured handling of scoped effects. In Ilya Sergey, editor, *Programming Languages and Systems - 31st European Symposium on Programming, ESOP 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings*, volume 13240 of *Lecture Notes in Computer Science*, pages 462–491. Springer, 2022. doi:10.1007/978-3-030-99336-8_17.

[YW14] Jeremy Yallop and Leo White. Lightweight higher-kinded polymorphism. In Michael Codish and Eijiro Sumii, editors, *Functional and Logic Programming*, pages 119–135, Cham, 2014. Springer International Publishing.

[YW23] Zhixuan Yang and Nicolas Wu. Modular models of monoids with operations. *Proc. ACM Program. Lang.*, 7(ICFP):566–603, 2023. doi:10.1145/3607850.

Appendix Contents

Appendix A. Semantic Derivations

This Appendix contains semantic derivations of different handler applications that are used in the examples throughout this paper.

A.1. **Nondeterminism.**

$$
\begin{aligned}
& \textbf{with } h_{\mathsf{ND}} \textbf{ handle } c_{\mathsf{ND1}} \\
\equiv\ & \textbf{with } h_{\mathsf{ND}} \textbf{ handle op } \mathtt{choose}\ ()\ (b \,.\, \textbf{if } b \textbf{ then return } 1 \textbf{ else return } 2) \\
\rightsquigarrow\ & \{\text{- E-HandOp -}\} \\
& \quad \textbf{do } xs \leftarrow (\boldsymbol{\lambda} y \,.\, \textbf{with } h_{\mathsf{ND}} \textbf{ handle if } b \textbf{ then return } 1 \textbf{ else return } 2)\ \mathsf{true} \\
& \quad \textbf{do } ys \leftarrow (\boldsymbol{\lambda} y \,.\, \textbf{with } h_{\mathsf{ND}} \textbf{ handle if } b \textbf{ then return } 1 \textbf{ else return } 2)\ \mathsf{false} \\
& \qquad xs +\!\!+ ys \\
\rightsquigarrow\ & \{\text{- E-AppAbs -}\} \\
& \quad \textbf{do } xs \leftarrow \textbf{with } h_{\mathsf{ND}} \textbf{ handle if } \mathsf{true} \textbf{ then return } 1 \textbf{ else return } 2 \\
& \quad \textbf{do } ys \leftarrow (\boldsymbol{\lambda} y \,.\, \textbf{with } h_{\mathsf{ND}} \textbf{ handle if } b \textbf{ then return } 1 \textbf{ else return } 2)\ \mathsf{false} \\
& \qquad xs +\!\!+ ys \\
\rightsquigarrow\ & \{\text{- reducing } \textbf{if} \text{ -}\} \\
& \quad \textbf{do } xs \leftarrow \textbf{with } h_{\mathsf{ND}} \textbf{ handle return } 1 \\
& \quad \textbf{do } ys \leftarrow (\boldsymbol{\lambda} y \,.\, \textbf{with } h_{\mathsf{ND}} \textbf{ handle if } b \textbf{ then return } 1 \textbf{ else return } 2)\ \mathsf{false} \\
& \qquad xs +\!\!+ ys \\
\rightsquigarrow\ & \{\text{- E-HandRet -}\} \\
& \quad \textbf{do } ys \leftarrow (\boldsymbol{\lambda} y \,.\, \textbf{with } h_{\mathsf{ND}} \textbf{ handle if } b \textbf{ then return } 1 \textbf{ else return } 2)\ \mathsf{false} \\
& \qquad [1] +\!\!+ ys \\
\rightsquigarrow\ & \{\text{- similar to above (the first branch of } \textbf{if}\text{) -}\} \\
& \qquad [1] +\!\!+ [2] \\
\rightsquigarrow\ & \{\text{- reducing } +\!\!+ \text{ -}\} \\
& \qquad \textbf{return } [1, 2]
\end{aligned}
$$

A.2. **Increment.**

$$
\begin{aligned}
& \textbf{with } h_{\mathsf{ND}} \textbf{ handle } run_{\mathsf{inc}}\ 0\ c_{\mathsf{inc}} \ \equiv\ \textbf{with } h_{\mathsf{ND}} \textbf{ handle } (\boldsymbol{\lambda} c\ p \,.\, \textbf{do } p' \leftarrow \\
& \quad \textbf{with } h_{\mathsf{inc}} \textbf{ handle } p\,;\, p'\ c)\ 0\ c_{\mathsf{inc}} \\
\rightsquigarrow\ & \{\text{- E-AppAbs -}\} \\
& \textbf{with } h_{\mathsf{ND}} \textbf{ handle do } p' \leftarrow \textbf{with } h_{\mathsf{inc}} \textbf{ handle } c_{\mathsf{inc}}\,;\, p'\ 0 \\
\equiv\ & \{\text{- definition of } c_{\mathsf{inc}} \text{ -}\} \\
& \textbf{with } h_{\mathsf{ND}} \textbf{ handle do } p' \leftarrow \textbf{with } h_{\mathsf{inc}} \textbf{ handle op } \mathtt{choose}\ ()\ (b \,.\, \textbf{if } b \textbf{ then} \\
& \quad \textbf{op } \mathtt{inc}\ ()\ (x \,.\, x + 5) \textbf{ else op } \mathtt{inc}\ ()\ (y \,.\, y + 2))\,;\, p'\ 0
\end{aligned}
$$

$\rightsquigarrow$ {- E-FwdOp -}
with h_{ND} **handle do** $p' \leftarrow$ **op** `choose` $()$ $(b \,.\, $ **with** h_{inc} **handle if** b **then**
op `inc` $()$ $(x \,.\, x + 5)$ **else op** `inc` $()$ $(y \,.\, y + 2))\,;\, p'\ 0$
$\rightsquigarrow$ {- E-Hand and E-DoOp -}
with h_{ND} **handle op** `choose` $()$ $(b \,.\,$ **do** $p' \leftarrow$ **with** h_{inc} **handle if** b **then**
op `inc` $()$ $(x \,.\, x + 5)$ **else op** `inc` $()$ $(y \,.\, y + 2)\,;\, p'\ 0)$
$\rightsquigarrow$ {- E-HandOp -}
do $xs \leftarrow (\boldsymbol{\lambda} b \,.\,$ **with** h_{ND} **handle do** $p' \leftarrow$ **with** h_{inc} **handle if** b **then op**
`inc` $()$ $(x \,.\, x + 5)$ **else op** `inc` $()$ $(y \,.\, y + 2)\,;\, p'\ 0)$ $\mathsf{true}\,;$
do $ys \leftarrow (\boldsymbol{\lambda} b \,.\,$ **with** h_{ND} **handle do** $p' \leftarrow$ **with** h_{inc} **handle if** b **then op**
`inc` $()$ $(x \,.\, x + 5)$ **else op** `inc` $()$ $(y \,.\, y + 2)\,;\, p'\ 0)$ $\mathsf{false}\,;\, xs +\!\!+ ys$
$\rightsquigarrow$ {- E-AppAbs -}
do $xs \leftarrow$ **with** h_{ND} **handle do** $p' \leftarrow$ **with** h_{inc} **handle if** true **then op**
`inc` $()$ $(x \,.\, x + 5)$ **else op** `inc` $()$ $(y \,.\, y + 2)\,;\, p'\ 0$
do $ys \leftarrow (\boldsymbol{\lambda} b \,.\,$ **with** h_{ND} **handle do** $p' \leftarrow$ **with** h_{inc} **handle if** b **then op**
`inc` $()$ $(x \,.\, x + 5)$ **else op** `inc` $()$ $(y \,.\, y + 2)\,;\, p'\ 0)$ $\mathsf{false}\,;\, xs +\!\!+ ys$
$\rightsquigarrow$ {- reducing **if** -}
do $xs \leftarrow$ **with** h_{ND} **handle** (**do** $p' \leftarrow$ **with** h_{inc} **handle op** `inc` $()$
$(x \,.\, x + 5)\,;\, p'\ 0)$
do $ys \leftarrow (\boldsymbol{\lambda} b \,.\,$ **with** h_{ND} **handle do** $p' \leftarrow$ **with** h_{inc} **handle if** b **then op**
`inc` $()$ $(x \,.\, x + 5)$ **else op** `inc` $()$ $(y \,.\, y + 2)\,;\, p'\ 0)$ $\mathsf{false}\,;\, xs +\!\!+ ys$
$\rightsquigarrow$ {- E-HandOp -}
do $xs \leftarrow$ **with** h_{ND} **handle** (**do** $p' \leftarrow$ **return** $(\boldsymbol{\lambda} s \,.\,$ **do** $s' \leftarrow s + 1\,;$ **do** $k' \leftarrow$
$(\boldsymbol{\lambda} x \,.\,$ **with** h_{inc} **handle** $(x + 5))\ s'\,;\, k'\ s')\,;\, p'\ 0)$
do $ys \leftarrow (\boldsymbol{\lambda} b \,.\,$ **with** h_{ND} **handle do** $p' \leftarrow$ **with** h_{inc} **handle if** b **then op**
`inc` $()$ $(x \,.\, x + 5)$ **else op** `inc` $()$ $(y \,.\, y + 2)\,;\, p'\ 0)$ $\mathsf{false}\,;\, xs +\!\!+ ys$
$\rightsquigarrow$ {- E-DoRet -}
do $xs \leftarrow$ **with** h_{ND} **handle** $(\boldsymbol{\lambda} s \,.\,$ **do** $s' \leftarrow s + 1\,;\, k' \leftarrow (\boldsymbol{\lambda} x \,.\,$ **with** h_{inc}
handle $(x + 5))s'\,;\, k'\ s')\ 0$
do $ys \leftarrow (\boldsymbol{\lambda} b \,.\,$ **with** h_{ND} **handle do** $p' \leftarrow$ **with** h_{inc} **handle if** b **then op**
`inc` $()$ $(x \,.\, x + 5)$ **else op** `inc` $()$ $(y \,.\, y + 2)\,;\, p'\ 0)$ $\mathsf{false}\,;\, xs +\!\!+ ys$
$\rightsquigarrow^*$ {- E-AppAbs and reducing + -}
do $xs \leftarrow$ **with** h_{ND} **handle** (**with** h_{inc} **handle** (**return** 6)) 1
do $ys \leftarrow (\boldsymbol{\lambda} b \,.\,$ **with** h_{ND} **handle do** $p' \leftarrow$ **with** h_{inc} **handle if** b **then op**
`inc` $()$ $(x \,.\, x + 5)$ **else op** `inc` $()$ $(y \,.\, y + 2)\,;\, p'\ 0)$ $\mathsf{false}\,;\, xs +\!\!+ ys$
$\rightsquigarrow$ {- E-HandRet -}
do $xs \leftarrow (\boldsymbol{\lambda} s \,.\,$ **return** $(6, s))\ 1$
do $ys \leftarrow (\boldsymbol{\lambda} b \,.\,$ **with** h_{ND} **handle do** $p' \leftarrow$ **with** h_{inc} **handle if** b **then op**
`inc` $()$ $(x \,.\, x + 5)$ **else op** `inc` $()$ $(y \,.\, y + 2)\,;\, p'\ 0)$ $\mathsf{false}\,;\, xs +\!\!+ ys$
$\rightsquigarrow$ {- E-AppAbs -}
do $xs \leftarrow$ **with** h_{ND} **handle return** $(6, 1)$
do $ys \leftarrow (\boldsymbol{\lambda} b \,.\,$ **with** h_{ND} **handle do** $p' \leftarrow$ **with** h_{inc} **handle if** b **then op**
`inc` $()$ $(x \,.\, x + 5)$ **else op** `inc` $()$ $(y \,.\, y + 2)\,;\, p'\ 0)$ $\mathsf{false}\,;\, xs +\!\!+ ys$
$\rightsquigarrow$ {- E-HandOp -}

$\quad$ **do** $xs \leftarrow$ **return** $[(6,1)]$
$\quad$ **do** $ys \leftarrow (\boldsymbol{\lambda} b \,.\, \textbf{with}\ h_{\mathsf{ND}}\ \textbf{handle do}\ p' \leftarrow \textbf{with}\ h_{\mathsf{inc}}\ \textbf{handle if}\ b\ \textbf{then op}$
$\quad\quad$ `inc` $()\ (x \,.\, x+5)\ \textbf{else op}\ \texttt{inc}\ ()\ (y \,.\, y+2)\,;\, p'\ 0)\ \mathsf{false}\,;\, xs +\!\!+ ys$
$\rightsquigarrow$ {- E-DoRet -}
$\quad$ **do** $ys \leftarrow (\boldsymbol{\lambda} b \,.\, \textbf{with}\ h_{\mathsf{ND}}\ \textbf{handle do}\ p' \leftarrow \textbf{if}\ b\ \textbf{then with}\ h_{\mathsf{inc}}\ \textbf{handle op}$
$\quad\quad$ `inc` $()\ (x \,.\, x+5)\ \textbf{else with}\ h_{\mathsf{inc}}\ \textbf{handle op}\ \texttt{inc}\ ()\ (y \,.\, y+2)\,;\, p'\ 0)\ \mathsf{false}\,;$
$\quad\quad\quad [(6,1)] +\!\!+ ys$
$\rightsquigarrow^*$ {- similar to above (the first branch of **if**) -}
$\quad$ **do** $ys \leftarrow$ **return** $[(3,1)]\ [(6,1)] +\!\!+ ys$
$\rightsquigarrow^*$ {- E-DoRet -}
$\quad [(6,1)] +\!\!+ [(3,1)]$
$\rightsquigarrow^*$ {- reducing $+\!\!+$ -}
$\quad$ **return** $[(6,1),(3,1)]$

A.3. **Once.**

with h_{once} **handle** c_{once}
$\rightsquigarrow$ {- E-HandSc -}
$\quad$ **do** $ts \leftarrow (\boldsymbol{\lambda} y \,.\, \textbf{with}\ h_{\mathsf{once}}\ \textbf{handle op}\ \texttt{choose}\ ()\ (x \,.\, \textbf{return}\ x))\ ()\,;$
$\quad$ **do** $b\ \leftarrow ts = [\,]\,;\, \textbf{if}\ b\ \textbf{then return}\ [\,]\ \textbf{else do}\ t \leftarrow \mathsf{head}\ ts\,;\, (\boldsymbol{\lambda} p \,.\, \textbf{with}$
$\quad\quad h_{\mathsf{once}}\ \textbf{handle}\ (\textbf{do}\ q \leftarrow \textbf{op}\ \texttt{choose}\ (b \,.\, \textbf{return}\ b)\,;\, \textbf{return}\ (p,q)))\ t$
$\rightsquigarrow$ {- E-Do and E-AppAbs -}
$\quad$ **do** $ts \leftarrow \textbf{with}\ h_{\mathsf{once}}\ \textbf{handle op}\ \texttt{choose}\ ()\ (x \,.\, \textbf{return}\ x)\,;$
$\quad$ **do** $b\ \leftarrow ts = [\,]\,;\, \textbf{if}\ b\ \textbf{then return}\ [\,]\ \textbf{else do}\ t \leftarrow \mathsf{head}\ ts\,;\, (\boldsymbol{\lambda} p \,.\, \textbf{with}$
$\quad\quad h_{\mathsf{once}}\ \textbf{handle}\ (\textbf{do}\ q \leftarrow \textbf{op}\ \texttt{choose}\ (b \,.\, \textbf{return}\ b)\,;\, \textbf{return}\ (p,q)))\ t$
$\rightsquigarrow$ {- E-Do and E-HandOp -}
$\quad$ **do** $ts \leftarrow$ **do** $xs \leftarrow (\boldsymbol{\lambda} x \,.\, \textbf{with}\ h_{\mathsf{once}}\ \textbf{handle return}\ x)\ \mathsf{true}\,;$
$\quad\quad\quad$ **do** $ys \leftarrow (\boldsymbol{\lambda} x \,.\, \textbf{with}\ h_{\mathsf{once}}\ \textbf{handle return}\ x)\ \mathsf{false}\,;\, xs +\!\!+ ys\,;$
$\quad$ **do** $b\ \leftarrow ts = [\,]\,;\, \textbf{if}\ b\ \textbf{then return}\ [\,]\ \textbf{else do}\ t \leftarrow \mathsf{head}\ ts\,;\, (\boldsymbol{\lambda} p \,.\, \textbf{with}$
$\quad\quad h_{\mathsf{once}}\ \textbf{handle}\ (\textbf{do}\ q \leftarrow \textbf{op}\ \texttt{choose}\ (b \,.\, \textbf{return}\ b)\,;\, \textbf{return}\ (p,q)))\ t$
$\rightsquigarrow$ {- E-Do and E-AppAbs -}
$\quad$ **do** $ts \leftarrow$ **do** $xs \leftarrow \textbf{with}\ h_{\mathsf{once}}\ \textbf{handle return}\ \mathsf{true}\,;$
$\quad\quad\quad$ **do** $ys \leftarrow \textbf{with}\ h_{\mathsf{once}}\ \textbf{handle return}\ \mathsf{false}\,;\, xs +\!\!+ ys\,;$
$\quad$ **do** $b\ \leftarrow ts = [\,]\,;\, \textbf{if}\ b\ \textbf{then return}\ [\,]\ \textbf{else do}\ t \leftarrow \mathsf{head}\ ts\,;\, (\boldsymbol{\lambda} p \,.\, \textbf{with}$
$\quad\quad h_{\mathsf{once}}\ \textbf{handle}\ (\textbf{do}\ q \leftarrow \textbf{op}\ \texttt{choose}\ (b \,.\, \textbf{return}\ b)\,;\, \textbf{return}\ (p,q)))\ t$
$\rightsquigarrow$ {- E-Do and E-HandRet -}
$\quad$ **do** $ts \leftarrow$ **do** $xs \leftarrow$ **return** $[\mathsf{true}]\,;$ **do** $ys \leftarrow$ **return** $[\mathsf{false}]\,;\, xs +\!\!+ ys\,;$
$\quad$ **do** $b\ \leftarrow ts = [\,]\,;\, \textbf{if}\ b\ \textbf{then return}\ [\,]\ \textbf{else do}\ t \leftarrow \mathsf{head}\ ts\,;\, (\boldsymbol{\lambda} p \,.\, \textbf{with}$
$\quad\quad h_{\mathsf{once}}\ \textbf{handle}\ (\textbf{do}\ q \leftarrow \textbf{op}\ \texttt{choose}\ (b \,.\, \textbf{return}\ b)\,;\, \textbf{return}\ (p,q)))\ t$
$\rightsquigarrow^*$ {- E-DoRet -}
$\quad$ **do** $ts \leftarrow [\mathsf{true}, \mathsf{false}]\,;$
$\quad$ **do** $b\ \leftarrow ts = [\,]\,;\, \textbf{if}\ b\ \textbf{then return}\ [\,]\ \textbf{else do}\ t \leftarrow \mathsf{head}\ ts\,;\, (\boldsymbol{\lambda} p \,.\, \textbf{with}$
$\quad\quad h_{\mathsf{once}}\ \textbf{handle}\ (\textbf{do}\ q \leftarrow \textbf{op}\ \texttt{choose}\ (b \,.\, \textbf{return}\ b)\,;\, \textbf{return}\ (p,q)))\ t$
$\rightsquigarrow^*$ {- E-DoRet -}

$(\boldsymbol{\lambda} p \,.\, \textbf{with}\ h_{\mathsf{once}}\ \textbf{handle}\ (\textbf{do}\ q \leftarrow \textbf{op}\ \texttt{choose}\ (b \,.\, \textbf{return}\ b)\,;\textbf{return}\ (p, q)))$
  true
$\rightsquigarrow$ {- E-AppAbs -}
  $\textbf{with}\ h_{\mathsf{once}}\ \textbf{handle}\ (\textbf{do}\ q \leftarrow \textbf{op}\ \texttt{choose}\ (b \,.\, \textbf{return}\ b)\,;\textbf{return}\ (\mathsf{true}, q))$
$\rightsquigarrow^*$ {- similar to A.1 (handling of `choose`) -}
  $\textbf{return}\ [(\mathsf{true}, \mathsf{true}), (\mathsf{true}, \mathsf{false})]$

Appendix B. Type Equivalence Rules

This appendix shows the type equivalence rules of λ_{sc}. Figures 10 and 11 contains the rules. Rules Q-AppAbs and Q-Swap deserve special attention. The other rules are straightforward.

$\boxed{\sigma_1 \equiv \sigma_2}$ Type equivalence

$$\frac{}{\sigma \equiv \sigma}\ \text{Q-Refl} \qquad \frac{\sigma_1 \equiv \sigma_2}{\sigma_2 \equiv \sigma_1}\ \text{Q-Symm} \qquad \frac{\sigma_1 \equiv \sigma_2 \qquad \sigma_2 \equiv \sigma_3}{\sigma_1 \equiv \sigma_3}\ \text{Q-Trans}$$

$$\frac{A_1 \equiv A_2 \qquad B_1 \equiv B_2}{(A_1, B_1) \equiv (A_2, B_2)}\ \text{Q-Pair} \qquad \frac{A \equiv B \qquad \underline{C} \equiv \underline{D}}{A \to \underline{C} \equiv B \to \underline{D}}\ \text{Q-Fun}$$

$$\frac{\underline{C_1} \equiv \underline{D_1} \qquad \underline{C_2} \equiv \underline{D_2}}{\underline{C_1} \Rightarrow \underline{C_2} \equiv \underline{D_1} \Rightarrow \underline{D_2}}\ \text{Q-Hand} \qquad \frac{\sigma_1 \equiv \sigma_2}{\forall\, \alpha\,.\, \sigma_1 \equiv \forall\, \alpha\,.\, \sigma_2}\ \text{Q-AllTy}$$

$$\frac{\sigma_1 \equiv \sigma_2}{\forall\, \mu\,.\, \sigma_1 \equiv \forall\, \mu\,.\, \sigma_2}\ \text{Q-AllRow} \qquad \frac{A \equiv B}{\lambda\, \alpha\,.\, A \equiv \lambda\, \alpha\,.\, B}\ \text{Q-Abs}$$

$$\frac{M_1 \equiv M_2 \qquad A \equiv B}{M_1\ A \equiv M_2\ B}\ \text{Q-App} \qquad \frac{}{(\lambda\, \alpha\,.\, A)\ B \equiv A\,[B \,/\, \alpha]}\ \text{Q-AppAbs}$$

$$\frac{A \equiv B \qquad E \equiv_{\langle\rangle} F}{A\,!\,\langle E\rangle \equiv B\,!\,\langle F\rangle}\ \text{Q-Comp}$$

Figure 10: Type equivalence of λ_{sc}.

$\boxed{E \equiv_{\langle\rangle} F}$ Row equivalence

$$\frac{}{E \equiv_{\langle\rangle} E}\ \text{R-Refl} \qquad \frac{E \equiv_{\langle\rangle} F}{F \equiv_{\langle\rangle} E}\ \text{R-Symm} \qquad \frac{E_1 \equiv_{\langle\rangle} E_2 \qquad E_2 \equiv_{\langle\rangle} E_3}{E_1 \equiv_{\langle\rangle} E_3}\ \text{R-Trans}$$

$$\frac{E \equiv_{\langle\rangle} F}{\ell\,;E \equiv_{\langle\rangle} \ell\,;F}\ \text{R-Head} \qquad \frac{\ell_1 \neq \ell_2}{\ell_1\,;\ell_2\,;E \equiv_{\langle\rangle} \ell_2\,;\ell_1\,;E}\ \text{R-Swap}$$

Figure 11: Row equivalence of λ_{sc}.

Appendix C. Well-scopedness Rules

This appendix shows the well-scopedness rules of λ_{sc}. Figure 12 contains the rules.

$\boxed{\Gamma \vdash \sigma}$ $\boxed{\Gamma \vdash M}$ $\boxed{\Gamma \vdash E}$ $\boxed{\Gamma \vdash \underline{C}}$ Type well-scopedness

$$\frac{}{\Gamma \vdash ()}\ \text{W-Unit} \qquad \frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash (A, B)}\ \text{W-Pair} \qquad \frac{\alpha \in \Gamma}{\Gamma \vdash \alpha}\ \text{W-Var}$$

$$\frac{\Gamma, \alpha \vdash A}{\Gamma \vdash \forall\, \alpha\,.\, A}\ \text{W-All} \qquad \frac{\Gamma \vdash A \qquad \Gamma \vdash E}{\Gamma \vdash A\,!\,\langle E \rangle}\ \text{W-Comp} \qquad \frac{\Gamma, \alpha \vdash A}{\Gamma \vdash \lambda\, \alpha\,.\, A}\ \text{W-Abs}$$

$$\frac{\Gamma \vdash M \qquad \Gamma \vdash A}{\Gamma \vdash M\ A}\ \text{W-App} \qquad \frac{\Gamma \vdash A \qquad \Gamma \vdash \underline{C}}{\Gamma \vdash A \to \underline{C}}\ \text{W-Fun} \qquad \frac{\Gamma \vdash \underline{C} \qquad \Gamma \vdash \underline{D}}{\Gamma \vdash \underline{C} \Rightarrow \underline{D}}\ \text{W-Hand}$$

$$\frac{\mu \in \Gamma}{\Gamma \vdash \mu}\ \text{W-RowVar} \qquad \frac{}{\Gamma \vdash \cdot}\ \text{W-EmptyRow} \qquad \frac{\Gamma \vdash E}{\Gamma \vdash \ell\,;\, E}\ \text{W-Extension}$$

Figure 12: Well-scopedness rules of λ_{sc}.

Appendix D. Syntax-directed version of λ_{sc}

This section describes the syntax-direction version of λ_{sc}.

The syntax-directed rules can be found in Figure 13 for value typing, Figure 14 for computation and Figure 15 for handler typing.

$\boxed{\Gamma \vdash_{\mathsf{SD}} v : A}$ Value Typing

$$\frac{(x : \sigma) \in \Gamma \qquad \sigma \leqslant A \qquad \Gamma \vdash_{\mathsf{SD}} A}{\Gamma \vdash_{\mathsf{SD}} x : A}\ \text{SD-Var} \qquad \frac{}{\Gamma \vdash_{\mathsf{SD}} () : ()}\ \text{SD-Unit}$$

$$\frac{\Gamma \vdash_{\mathsf{SD}} v_1 : A \qquad \Gamma \vdash_{\mathsf{SD}} v_2 : B}{\Gamma \vdash_{\mathsf{SD}} (v_1, v_2) : (A, B)}\ \text{SD-Pair} \qquad \frac{\Gamma, x : A \vdash_{\mathsf{SD}} c : \underline{C}}{\Gamma \vdash_{\mathsf{SD}} \boldsymbol{\lambda} x\,.\,c : A \to \underline{C}}\ \text{SD-Abs}$$

SD-Handler

$$\frac{\begin{array}{lll} F \equiv_{\langle\rangle} \mathit{labels}(\mathit{oprs}) ; E & \alpha \notin \Gamma & \Gamma, \alpha \vdash_M \mathbf{return}\ x \mapsto c_r : M\ \alpha!\langle E \rangle \\ \Gamma, \alpha \vdash_M \mathit{oprs} : M\ \alpha!\langle E \rangle & \Gamma, \alpha \vdash_M \mathbf{fwd}\ f\ p\ k \mapsto c_f : M\ \alpha!\langle E \rangle & \Gamma \vdash_{\mathsf{SD}} A \end{array}}{\Gamma \vdash_{\mathsf{SD}} \mathbf{handler}_M\ \{\mathbf{return}\ x \mapsto c_r, \mathit{oprs}, \mathbf{fwd}\ f\ p\ k \mapsto c_f\} : A!\langle F \rangle \Rightarrow M\ A!\langle E \rangle}$$

Figure 13: Syntax-directed value typing.

The syntax-directed system is obtained by incorporating the non-syntax-directed rules into the syntax-directed-ones where needed. In particular, we inline the non-syntax-directed rules for equivalence (T-EqV and T-EqC) into the syntax-directed rules that mention the same type or row twice in their assumptions (e.g., SD-App, SD-Do). Similarly, we inline the rules T-Inst, T-InstEff, T-Gen and T-GenEff for instantiating and generalizing type and row variables. The generalization is incorporated into the rule for let-bindings (T-Let). Instantiation is incorporated into the variable rule (T-Var) using $\sigma \leqslant A$ defined in Figure 16.

Instantiation is also incorporated into the handler rule: we implicitly instantiate α with an arbitrary type A, which results in a monomorphically typed handler. However, since SD-Handler insists on sufficiency polymorphic handler clauses, we can still handle scoped effects by polymorphic recursion.

Figure 17 displays declarative and syntax-directed typing derivations for both inline handler application (**with** h **handle** c) as well as let-bound handlers. As can be seen in the first derivation, in the case of inline handler application, the declarative system derives a polymorphically typed handler, which is instantiated. The syntax-directed system essentially combines these steps, as can be seen in the second derivation. In the case of a let-bound handler, the declarative system keeps the polymorphic handler type as-is (third derivation). The syntax-directed system however instantiates and then immediately generalizes handlers, as can be seen in the fourth derivation.

The other rules of the declarative system are syntax-directed and remain unchanged.

$\boxed{\Gamma \vdash_{\mathsf{SD}} c : \underline{C}}$ Computation Typing

$$\frac{\Gamma \vdash_{\mathsf{SD}} v_1 : A_1 \to \underline{C} \qquad \Gamma \vdash_{\mathsf{SD}} v_2 : A_2 \qquad A_1 \equiv A_2}{\Gamma \vdash_{\mathsf{SD}} v_1\ v_2 : \underline{C}}\ \text{SD-App}$$

$$\frac{\Gamma \vdash_{\mathsf{SD}} c_1 : A!\langle E_1 \rangle \qquad \Gamma, x : A \vdash_{\mathsf{SD}} c_2 : B!\langle E_2 \rangle \qquad E_1 \equiv E_2}{\Gamma \vdash_{\mathsf{SD}} \mathbf{do}\ x \leftarrow c_1\,;\, c_2 : B!\langle E_2 \rangle}\ \text{SD-Do}$$

$$\frac{\Gamma, \overline{\alpha}, \overline{\mu} \vdash_{\mathsf{SD}} v : A \qquad (\overline{\alpha} \notin \Gamma) \qquad (\overline{\mu} \notin \Gamma) \qquad \Gamma, x : \overline{\forall\, \alpha}\,.\,\overline{\forall\, \mu}\,.\, A \vdash_{\mathsf{SD}} c : \underline{C}}{\Gamma \vdash_{\mathsf{SD}} \mathbf{let}\ x = v\ \mathbf{in}\ c : \underline{C}}\ \text{SD-Let}$$

$$\frac{\Gamma \vdash_{\mathsf{SD}} v : A}{\Gamma \vdash_{\mathsf{SD}} \mathbf{return}\ v : A!\langle E \rangle}\ \text{SD-Ret}$$

$$\frac{\Gamma \vdash_{\mathsf{SD}} v : \underline{C}_1 \Rightarrow \underline{D}_1 \qquad \Gamma \vdash_{\mathsf{SD}} c : \underline{C}_2 \qquad \underline{C}_1 \equiv \underline{C}_2 \qquad \underline{D}_1 \equiv \underline{D}_2}{\Gamma \vdash_{\mathsf{SD}} \mathbf{with}\ v\ \mathbf{handle}\ c : \underline{D}_2}\ \text{SD-Hand}$$

$$\frac{\begin{array}{c}(\ell^{\mathsf{op}} : A_{\mathsf{op}} \twoheadrightarrow B_{\mathsf{op}}) \in \Sigma \\ \Gamma \vdash_{\mathsf{SD}} v : A_1 \qquad A_{\mathsf{op}} \equiv A_1 \qquad \Gamma, y : B_{\mathsf{op}} \vdash_{\mathsf{SD}} c : A!\langle E \rangle \qquad \ell^{\mathsf{op}} \in E\end{array}}{\Gamma \vdash_{\mathsf{SD}} \mathbf{op}\ \ell^{\mathsf{op}}\ v\ (y\,.\,c) : A!\langle E \rangle}\ \text{SD-Op}$$

$$\frac{\begin{array}{c}(\ell^{\mathsf{sc}} : A_{\mathsf{sc}} \twoheadrightarrow B_{\mathsf{sc}}) \in \Sigma \qquad \Gamma \vdash_{\mathsf{SD}} v : A_1 \qquad A_{\mathsf{sc}} \equiv A_1 \\ \Gamma, y : B_{\mathsf{sc}} \vdash_{\mathsf{SD}} c_1 : B!\langle E_1 \rangle \qquad \Gamma, z : B \vdash_{\mathsf{SD}} c_2 : A!\langle E_2 \rangle \qquad E_1 \equiv E_2 \qquad \ell^{\mathsf{sc}} \in E_2\end{array}}{\Gamma \vdash_{\mathsf{SD}} \mathbf{sc}\ \ell^{\mathsf{sc}}\ v\ (y\,.\,c_1)\ (z\,.\,c_2) : A!\langle E_2 \rangle}\ \text{SD-Sc}$$

Figure 14: Syntax-directed computation typing.

$\boxed{\Gamma \vdash_M \ \mathbf{return}\ x \mapsto c_r : M\ A!\langle E\rangle}$ $\boxed{\Gamma \vdash_M \ \mathit{oprs} : M\ A!\langle E\rangle}$
$\boxed{\Gamma \vdash_M \ \mathbf{fwd}\ f\ p\ k \mapsto c_f : M\ A!\langle E\rangle}$
Return-, operation-, and forwarding-clause typing

$$\frac{\Gamma, x : A_1 \vdash_{\mathsf{SD}} c_r : M\ A_2!\langle E\rangle \qquad A_1 \ \equiv\ A_2}{\Gamma \vdash_M \ \mathbf{return}\ x \mapsto c_r : M\ A!\langle E_2\rangle}\ \text{SD-Return} \qquad \frac{}{\Gamma \vdash_M \ \cdot : M\ A!\langle E\rangle}\ \text{SD-Empty}$$

$$\frac{\begin{array}{c}\Gamma \vdash_M \ \mathit{oprs} : M\ A_1!\langle E_1\rangle \qquad (\ell^{\mathsf{op}} : A_{\mathsf{op}} \twoheadrightarrow B_{\mathsf{op}}) \ \in\ \Sigma \\ \Gamma, x : A_{\mathsf{op}}, k : B_{\mathsf{op}} \to M\ A_1!\langle E_1\rangle \vdash_{\mathsf{SD}} c : M\ A_2!\langle E_2\rangle \\ M\ A_1!\langle E_1\rangle \ \equiv\ M\ A_2!\langle E_2\rangle\end{array}}{\Gamma \vdash_M \ \mathbf{op}\ \ell^{\mathsf{op}}\ x\ k \mapsto c, \mathit{oprs} : M\ A_2!\langle E_2\rangle}\ \text{SD-OprOp}$$

$$\frac{\begin{array}{c}\Gamma \vdash_M \ \mathit{oprs} : M\ A_1!\langle E_1\rangle \qquad (\ell^{\mathsf{sc}} : A_{\mathsf{sc}} \twoheadrightarrow B_{\mathsf{sc}}) \ \in\ \Sigma \qquad \beta \notin \Gamma \\ \Gamma, \beta, x : A_{\mathsf{sc}}, p : B_{\mathsf{sc}} \to M\ \beta!\langle E_1\rangle, k : \beta \to M\ A_1!\langle E_1\rangle \vdash_{\mathsf{SD}} c : M\ A_2!\langle E_2\rangle \\ M\ A_1!\langle E_1\rangle \ \equiv\ M\ A_2!\langle E_2\rangle\end{array}}{\Gamma \vdash_M \ \mathbf{sc}\ \ell^{\mathsf{sc}}\ x\ p\ k \mapsto c, \mathit{oprs} : M\ A_2!\langle E_2\rangle}\ \text{SD-OprSc}$$

$$\frac{\begin{array}{c}\alpha, \beta, \gamma, \delta \notin \Gamma \qquad A_p = \alpha \to M\ \beta!\langle E_1\rangle \\ A'_p = \alpha \to \gamma!\langle E_1\rangle \qquad A_k = \beta \to M\ A_1!\langle E_1\rangle \qquad A'_k = \gamma \to \delta!\langle E_1\rangle \\ \Gamma, \alpha, \beta, p : A_p, k : A_k, f : \forall\ \gamma\ \delta\ .\ (A'_p, A'_k) \to \delta!\langle E_1\rangle \vdash_{\mathsf{SD}} c_f : M\ A_2!\langle E_2\rangle \\ M\ A_1!\langle E_1\rangle \ \equiv\ M\ A_2!\langle E_2\rangle \qquad \Gamma \vdash_{\mathsf{SD}} A_2\end{array}}{\Gamma \vdash_M \ \mathbf{fwd}\ f\ p\ k \mapsto c_f : M\ A_2!\langle E_2\rangle}\ \text{SD-Fwd}$$

Figure 15: Syntax-directed handler typing.

$\boxed{\sigma \leqslant A}$ σ-instantiation

$$\frac{}{A \leqslant A}\ \sigma\text{-Inst-Base} \qquad \frac{[B\ /\ \alpha]\sigma \leqslant A}{\forall\ \alpha\ .\ \sigma \leqslant A}\ \sigma\text{-Inst-}\alpha \qquad \frac{[E\ /\ \mu]\ \sigma \leqslant A}{\forall\ \mu\ .\ \sigma \leqslant A}\ \sigma\text{-Inst-}\mu$$

Figure 16: σ-instantiation.

$$\dfrac{\dfrac{\dfrac{\Gamma, \alpha \vdash_{\mathsf{SD}} \mathit{oprs} : M\ \alpha!\langle E\rangle}{\Gamma \vdash_{\mathsf{SD}} h : \forall\, \alpha\,.\, \alpha!\langle F\rangle \Rightarrow M\ \alpha!\langle E\rangle}\ \text{T-Handler} \quad \Gamma \vdash_{\mathsf{SD}} A}{\Gamma \vdash_{\mathsf{SD}} h : A!\langle F\rangle \Rightarrow M\ A!\langle E\rangle}\ \text{T-Inst} \quad \Gamma \vdash_{\mathsf{SD}} c : A!\langle F\rangle}{\Gamma \vdash_{\mathsf{SD}} \mathbf{with}\ h\ \mathbf{handle}\ c : M\ A!\langle E\rangle}\ \text{T-Hand}$$

$$\dfrac{\dfrac{\Gamma, \alpha \vdash_M \mathit{oprs} : M\ \alpha!\langle E\rangle \qquad \Gamma \vdash_{\mathsf{SD}} A}{\Gamma \vdash_{\mathsf{SD}} h_M : A!\langle F\rangle \Rightarrow M\ A!\langle E\rangle}\ \text{SD-Handler} \qquad \Gamma \vdash_{\mathsf{SD}} c : A!\langle F\rangle}{\Gamma \vdash_{\mathsf{SD}} \mathbf{with}\ h_M\ \mathbf{handle}\ c : M\ A!\langle E\rangle}\ \text{SD-Hand}$$

$$\dfrac{\dfrac{\Gamma, \alpha \vdash_{\mathsf{SD}} \mathit{oprs} : M\ \alpha!\langle E\rangle}{\Gamma \vdash_{\mathsf{SD}} h : \forall\, \alpha\,.\, \alpha!\langle F\rangle \Rightarrow M\ \alpha!\langle E\rangle}\ \text{T-Handler} \quad \Gamma, x : \forall\, \alpha\,.\, \alpha!\langle F\rangle \Rightarrow M\ \alpha!\langle E\rangle \vdash_{\mathsf{SD}} c : \underline{C}}{\Gamma \vdash_{\mathsf{SD}} \mathbf{let}\ x = h\ \mathbf{in}\ c : \underline{C}}\ \text{T-Let}$$

$$\dfrac{\dfrac{\Gamma, \alpha, \beta \vdash_M \mathit{oprs} : M\ \beta!\langle E\rangle \qquad \Gamma, \alpha \vdash_{\mathsf{SD}} \alpha}{\Gamma \vdash_{\mathsf{SD}} h_M : \alpha!\langle F\rangle \Rightarrow M\ \alpha!\langle E\rangle}\ \text{SD-Handler} \quad \Gamma, x : \forall\, \alpha\,.\, \alpha!\langle F\rangle \Rightarrow M\ \alpha!\langle E\rangle \vdash_{\mathsf{SD}} c : \underline{C}}{\Gamma \vdash_{\mathsf{SD}} \mathbf{let}\ x = h_M\ \mathbf{in}\ c : \underline{C}}\ \text{SD-Let}$$

Figure 17: Handler generalisation and instantiation

Appendix E. Metatheory

E.1. **Lemmas.**

Lemma E.1 (Canonical forms)**.**
- *If* $\cdot \vdash_{\mathsf{SD}} v : A \to \underline{C}$ *then* v *is of shape* $\boldsymbol{\lambda} x \,.\, c$.
- *If* $\cdot \vdash_{\mathsf{SD}} v : \underline{C} \Rightarrow \underline{D}$ *then* v *is of shape* h.

Lemma E.2 (Generalisation-equivalence)**.** *If* $\sigma_1 \leqslant A_1$ *and* $\sigma_1 \equiv \sigma_2$*, then there exists a* A_2 *such that* $A_1 \equiv A_2$ *and* $\sigma_2 \leqslant A_2$.

Lemma E.3 (Generalisation-instantiation)**.** *If* $\Gamma, \overline{\alpha}, \overline{\mu} \vdash_{\mathsf{SD}} v : A$ *and* $\overline{\forall\, \alpha}\, \overline{\forall\, \mu} \,.\, A \leqslant B$*, then* $\Gamma \vdash_{\mathsf{SD}} v : B$.

Lemma E.4 (Preservation of types under term substitution)**.** *Given* $\Gamma_1, \overline{\alpha}, \overline{\mu} \vdash_{\mathsf{SD}} v : A_1$ *and* $A_1 \equiv A_2$ *we have that:*
- *If* $\Gamma_1, x : \overline{\forall\, \alpha}\, \overline{\forall\, \mu} \,.\, A_2, \Gamma_2 \vdash_{\mathsf{SD}} c : \underline{C}_1$*, then there exists a* $\underline{C}_2$ *such that* $\underline{C}_1 \equiv \underline{C}_2$ *and* $\Gamma_1, \Gamma_2 \vdash_{\mathsf{SD}} [v \,/\, x]\, c : \underline{C}_2$.
- *If* $\Gamma_1, x : \overline{\forall\, \alpha}\, \overline{\forall\, \mu} \,.\, A_2, \Gamma_2 \vdash_{\mathsf{SD}} v : B_1$*, then there exists a* B_2 *such that* $B_1 \equiv B_2$ *and* $\Gamma_1, \Gamma_2 \vdash_{\mathsf{SD}} [v \,/\, x]\, v : B_2$.

Proof. By mutual induction on the typing derivations. The only interesting case, SD-Var, requires us to show that, given $\Gamma_1, x : \overline{\forall\, \alpha}\, \overline{\forall\, \mu} \,.\, A_2, \Gamma_2 \vdash_{\mathsf{SD}} y : B_1$, there exists a B_2 such that $B_1 \equiv B_2$ and $\Gamma_1, \Gamma_2 \vdash_{\mathsf{SD}} [v \,/\, x]\, y : B_2$. If $x \neq y$, it is trivial. If $x = y$, then $\overline{\forall\, \alpha}\, \overline{\forall\, \mu} \,.\, A_2 \leqslant B_1$, which means by Lemma E.2 there exists a B_2 such that $B_1 \equiv B_2$ and $\overline{\forall\, \alpha}\, \overline{\forall\, \mu} \,.\, A_1 \leqslant B_2$, which means the result follows from Lemma E.3. □

Lemma E.5 (Preservation of types under type substitution)**.** *If* $\Gamma_1, \alpha, \Gamma_2 \vdash_{\mathsf{SD}} c : \underline{C}$ *and* $\Gamma_1 \vdash_{\mathsf{SD}} B$*, then* $\Gamma_1, [B \,/\, \alpha]\, \Gamma_2 \vdash_{\mathsf{SD}} c : [B \,/\, \alpha]\, \underline{C}$.

Lemma E.6 (Unused binding insertion)**.** *If* $\Gamma_1, \Gamma_2 \vdash_{\mathsf{SD}} c : \underline{C}$ *and* $x \notin c$ *then* $\Gamma_1, x : A, \Gamma_2 \vdash_{\mathsf{SD}} c : \underline{C}$.

Lemma E.7 (Handlers are polymorphic)**.** *If* $\Gamma \vdash_{\mathsf{SD}} h : A\,!\,\langle F \rangle \Rightarrow M\ A\,!\,\langle E \rangle$ *and* $\Gamma \vdash_{\mathsf{SD}} B$*, then* $\Gamma \vdash_{\mathsf{SD}} h : B\,!\,\langle F \rangle \Rightarrow M\ B\,!\,\langle E \rangle$.

Lemma E.8 (Op membership)**.** *If* $\Gamma \vdash_{\mathsf{SD}} oprs : \underline{C}$ *and* $\mathbf{op}\ \ell^{op}\ x\ k \mapsto c \in oprs$*, then there exists* $oprs_1$ *and* $oprs_2$ *such that* $oprs = oprs_1, \mathbf{op}\ \ell^{op}\ k \vdash_{\mathsf{SD}} c, oprs_2$ *and* $\Gamma \vdash_{\mathsf{SD}} \mathbf{op}\ \ell^{op}\ x\ k \mapsto c, oprs_2 : \underline{C}$.

Lemma E.9 (Sc membership)**.** *If* $\Gamma \vdash_{\mathsf{SD}} oprs : \underline{C}$ *and* $(\mathbf{sc}\ \ell^{sc}\ x\ p\ k \mapsto c) \in h$*, then there exists* $oprs_1$ *and* $oprs_2$ *such that* $oprs = oprs_1, \mathbf{sc}\ \ell^{sc}\ x\ p\ k \mapsto c, oprs_2$ *and* $\Gamma \vdash_{\mathsf{SD}} \mathbf{sc}\ \ell^{sc}\ x\ p\ k \mapsto c, oprs_2 : \underline{C}$.

Lemma E.10 (Syntax-directed to Non-syntax-directed value typing)**.** *If* $\Gamma \vdash v : \overline{\forall\, \alpha} \,.\, \overline{\forall\, \mu} \,.\, A$ *then* $\Gamma, \overline{\alpha}, \overline{\mu} \vdash_{\mathsf{SD}} v : A$.

Lemma E.11 (Non-syntax-directed iff Syntax-directed)**.** $\Gamma \vdash c : \underline{C} \iff \Gamma \vdash_{\mathsf{SD}} c : \underline{C}$.

E.2. **Subject reduction.**

Theorem 8.1 (Subject Reduction)**.** *If* $\Gamma \vdash c : \underline{C}$ *and* $c \rightsquigarrow c'$*, then there exists a* $\underline{C}'$ *such that* $\underline{C} \equiv \underline{C}'$ *and* $\Gamma \vdash c' : \underline{C}'$*.*

Proof. By Lemma E.11 (above) and Theorem E.12 (below). □

Theorem E.12 (Syntax-directed Subject Reduction)**.** *If* $\Gamma \vdash_{\mathsf{SD}} c : \underline{C}$ *and* $c \rightsquigarrow c'$*, then there exists a* $\underline{C}'$ *such that* $\underline{C} \equiv \underline{C}'$ *and* $\Gamma \vdash_{\mathsf{SD}} c' : \underline{C}'$*.*

Proof. Assume, without loss of generality, that $\underline{C} = B\,!\langle F\rangle$ for some B, F. Proceed by induction on the derivation $c \rightsquigarrow c'$.

- E-AppAbs: Inversion on $\Gamma \vdash_{\mathsf{SD}} (\boldsymbol{\lambda} x\,.\,c)\ v : B\,!\langle F\rangle$ (SD-App) gives $\Gamma \vdash_{\mathsf{SD}} \boldsymbol{\lambda} x\,.\,c : A_1 \to B\,!\langle F\rangle$ (1), $\Gamma \vdash_{\mathsf{SD}} v : A_2$ (2), and $A_1 \equiv A_2$ (3). Inversion on fact 1 (SD-Abs) gives $\Gamma, x : A_1 \vdash_{\mathsf{SD}} c : B\,!\langle F\rangle$ (4), which means the goal follows from facts 2 and 4 and Lemma E.4.
- E-Let: Inversion on $\Gamma \vdash_{\mathsf{SD}} \mathbf{let}\ x = v\ \mathbf{in}\ c : B\,!\langle F\rangle$ (SD-Let) gives $\Gamma \vdash_{\mathsf{SD}} v : A$ (1), $\sigma = \mathit{gen}\ (A, \Gamma)$ (2), and $\Gamma, x : \sigma \vdash_{\mathsf{SD}} c : B\,!\langle F\rangle$ (3), which means the goal follows from facts 1 and 3 and Lemma E.4.
- E-Do: Follows from the IH.
- E-DoRet: Inversion on $\Gamma \vdash_{\mathsf{SD}} \mathbf{do}\ x \leftarrow \mathbf{return}\ v\ \mathbf{in}\ c : B\,!\langle F_2\rangle$ (SD-Do) gives $\Gamma \vdash_{\mathsf{SD}} \mathbf{return}\ v : A\,!\langle F_1\rangle$ (1) and $\Gamma, x : A \vdash_{\mathsf{SD}} c : B\,!\langle F_2\rangle$ (2). Inversion on (1) (SD-Ret) gives $\Gamma \vdash_{\mathsf{SD}} v : A$ (3). The case follows from facts 2 and 4 and Lemma E.4.
- E-DoOp: Similar to E-DoSc. By inversion on $\Gamma \vdash_{\mathsf{SD}} \mathbf{do}\ x \leftarrow \mathbf{op}\ \ell^{\mathsf{op}}\ v\ (y\,.\,c_1)\ \mathbf{in}\ c_2 : B\,!\langle F_2\rangle$ (SD-Do) we have that $\Gamma \vdash_{\mathsf{SD}} \mathbf{op}\ \ell^{\mathsf{op}}\ v\ (y\,.\,c_1) : A\,!\langle F_1\rangle$ (1), $\Gamma, x : A \vdash_{\mathsf{SD}} c_2 : B\,!\langle F_2\rangle$ (2), and $F_1 \equiv F_2$ (3). From inversion on fact 1 (SD-Op) it follows that $\ell^{\mathsf{op}} : A_{\mathsf{op}} \twoheadrightarrow B_{\mathsf{op}} \in \Sigma$ (4), $\Gamma \vdash_{\mathsf{SD}} v : A_1$ (5), $A_{\mathsf{op}} \equiv A_1$ (6), $\Gamma, y : B_{\mathsf{op}} \vdash_{\mathsf{SD}} c_1 : A\,!\langle F_1\rangle$ (7), and $\ell^{\mathsf{op}} \in F_1$ (8). Lemma E.6 on (2) gives us $\Gamma, y : B_{\mathsf{op}}, x : A \vdash_{\mathsf{SD}} c_2 : B\,!\langle F_2\rangle$ (9). Facts 3, 7 and 9 and rule SD-Do give us $\Gamma, y : B_{\mathsf{op}} \vdash_{\mathsf{SD}} \mathbf{do}\ x \leftarrow c_1\ \mathbf{in}\ c_2 : B\,!\langle F_2\rangle$ (10). Our goal then follows from facts 4, 5, 6, 8, and 10 and rule SD-Op.
- E-DoSc: Similar to E-DoOp. By inversion on $\Gamma \vdash_{\mathsf{SD}} \mathbf{do}\ x \leftarrow \mathbf{sc}\ \ell^{\mathsf{sc}}\ v\ (y\,.\,c_1)\ (z\,.\,c_2)\ \mathbf{in}\ c_3 : B\,!\langle F_3\rangle$ (SD-Do) we have that $\Gamma \vdash_{\mathsf{SD}} \mathbf{sc}\ \ell^{\mathsf{sc}}\ v\ (y\,.\,c_1)\ (z\,.\,c_2) : A\,!\langle F_2\rangle$ (1), $\Gamma, x : A \vdash_{\mathsf{SD}} c_3 : B\,!\langle F_3\rangle$ (2), and $F_2 \equiv F_3$ (2.1). From inversion on fact 1 (SD-Sc) it follows that $\ell^{\mathsf{sc}} : A_{\mathsf{sc}} \twoheadrightarrow B_{\mathsf{sc}} \in \Sigma$ (3), $\Gamma \vdash_{\mathsf{SD}} v : A_1$ (4), $A_{\mathsf{sc}} \equiv A_1$ (5), $\Gamma, y : B_{\mathsf{sc}} \vdash_{\mathsf{SD}} c_1 : B'\,!\langle F_1\rangle$ (6), $\Gamma, z : B' \vdash_{\mathsf{SD}} c_2 : A\,!\langle F_2\rangle$ (7), $F_1 \equiv F_2$ (8), and $\ell^{\mathsf{sc}} \in F_2$ (9). Lemma E.6 on (2) gives us $\Gamma, z : B', x : A \vdash_{\mathsf{SD}} c_3 : B\,!\langle F_3\rangle$ (10), which means facts 2.1, 7 and 10 and rule SD-Do give us $\Gamma, z : B' \vdash_{\mathsf{SD}} \mathbf{do}\ x \leftarrow c_2\ \mathbf{in}\ c_3 : B\,!\langle F_3\rangle$ (11). Our goal then follows from facts 3, 4, 5, 6, 8 9, and 11 and rule SD-Sc.
- E-Hand: Follows from the IH.
- E-HandRet: By inversion on $\Gamma \vdash_{\mathsf{SD}} \mathbf{with}\ h\ \mathbf{handle\ return}\ v : B\,!\langle F_2\rangle$ (SD-Hand) we have that $\Gamma \vdash_{\mathsf{SD}} h : \underline{C}_1 \Rightarrow B\,!\langle F_2\rangle$ (1), $\Gamma \vdash_{\mathsf{SD}} \mathbf{return}\ v : \underline{C}_2$ (2), and $\underline{C}_1 \equiv \underline{C}_2$ (3). Inversion on fact 1 (SD-Handler) gives $B = M\ A_2$, $\underline{C}_1 = A_2\,!\langle E\rangle$, and $\Gamma, \alpha \vdash_M \mathbf{return}\ x \mapsto c_r : M\ \alpha\,!\langle F_2\rangle$ (4). Based on fact (3) we get that $\underline{C}_2 = A2'\,!\langle E'\rangle$, $A_2 \equiv A2'$ (4), and $E \equiv E'$ (5). Inversion on fact 4 (SD-Return) gives $\Gamma, \alpha, x : A_1 \vdash_{\mathsf{SD}} c_r : M\ \alpha\,!\langle F_2\rangle$ (5) and $A_1 \equiv A_2$ (6). Inversion on fact 2 (SD-Ret) gives $\Gamma \vdash_{\mathsf{SD}} v : A2'$ (7). From facts 4-8 and Lemma E.4, we get that $\Gamma, \alpha \vdash_{\mathsf{SD}} [\,v\ /\ x\,]\ c_r : M\ \alpha\,!\langle F_2\rangle$ (8). We obtain our goal from fact 8 and Lemma E.5.
- E-HandOp: By inversion on $\Gamma \vdash_{\mathsf{SD}} \mathbf{with}\ h\ \mathbf{handle\ op}\ \ell^{\mathsf{op}}\ v\ (y\,.\,c_1) : B\,!\langle F_2\rangle$ (SD-Hand) we have that $\Gamma \vdash_{\mathsf{SD}} h : \underline{C}_1 \Rightarrow B\,!\langle F_2\rangle$ (1), $\Gamma \vdash_{\mathsf{SD}} \mathbf{op}\ \ell^{\mathsf{op}}\ v\ (y\,.\,c_1) : \underline{C}_2$ (2), and $\underline{C}_1 \equiv \underline{C}_2$ (3). Inversion on fact 1 (SD-Handler) gives $B = M\ A_2$, $\underline{C}_1 = A_2\,!\langle E\rangle$, and

$\Gamma, \alpha \vdash_M \mathit{oprs} : M\ \alpha!\langle F_2\rangle$ (4). Based on fact (3) we get that $\underline{C}_2 = A2'!\langle E'\rangle$, $A_2 \equiv A2'$ (4), and $E \equiv E'$ (5). Inversion on fact 2 (SD-Op) gives us $\ell^{\mathsf{op}} : A_{\mathsf{op}} \twoheadrightarrow B_{\mathsf{op}} \in \Sigma$ (6), $\Gamma \vdash_{\mathsf{SD}} v : A_1$ (7), $A_{\mathsf{op}} \equiv A_1$ (8), $\Gamma, y : B_{\mathsf{op}} \vdash_{\mathsf{SD}} c_1 : A2'!\langle E'\rangle$ (9), and $\ell^{\mathsf{op}} \in E'$ (10). By Lemma E.8 we get that $\Gamma, \alpha \vdash_M \mathbf{op}\ \ell^{\mathsf{op}}\ x\ k \mapsto c, \mathit{oprs}_2 : M\ \alpha!\langle F_2\rangle$ (11). Inversion on fact 11 (SD-OprOp) gives that $\Gamma, \alpha \vdash_M \mathit{oprs} : M\ \alpha!\langle F_1\rangle$ (12), $(\ell^{\mathsf{op}} : A_{\mathsf{op}} \twoheadrightarrow B_{\mathsf{op}}) \in \Sigma$ (13), $\Gamma, \alpha, x : A_{\mathsf{op}}, k : B_{\mathsf{op}} \to M\ \alpha!\langle F_1\rangle \vdash_{\mathsf{SD}} c : M\ \alpha!\langle F_2\rangle$ (14), and $F_1 \equiv F_2$ (15). Facts 1, 4 and 9 in combination with constructors SD-Abs and ST-Hand gives us that $\Gamma \vdash_{\mathsf{SD}} \boldsymbol{\lambda} y\,.\,\mathbf{with}\ h\ \mathbf{handle}\ c_1 : B_{\mathsf{op}} \to M\ A_2!\langle F_2\rangle$ (16). The goal follows from facts 7, 8, 14 and 16 and lemmas Lemmas E.4 and E.5.

- E-FwdOp By inversion on $\Gamma \vdash_{\mathsf{SD}} \mathbf{with}\ h\ \mathbf{handle\ op}\ \ell^{\mathsf{op}}\ v\ (y\,.\,c_1) : B!\langle F_2\rangle$ (SD-Hand) we have that $\Gamma \vdash_{\mathsf{SD}} h : \underline{C}_1 \Rightarrow B!\langle F_2\rangle$ (1), $\Gamma \vdash_{\mathsf{SD}} \mathbf{op}\ \ell^{\mathsf{op}}\ v\ (y\,.\,c_1) : \underline{C}_2$ (2), and $\underline{C}_1 \equiv \underline{C}_2$ (3). Inversion on fact 1 (SD-Handler) gives $B = M\ A_2$, and $\underline{C}_1 = A_2!\langle E\rangle$. Based on fact (3) we get that $\underline{C}_2 = A2'!\langle E'\rangle$, $A_2 \equiv A2'$ (4), and $E \equiv E'$ (5). Inversion on fact 2 (SD-Op) gives us $\ell^{\mathsf{op}} : A_{\mathsf{op}} \twoheadrightarrow B_{\mathsf{op}} \in \Sigma$ (6), $\Gamma \vdash_{\mathsf{SD}} v : A_1$ (7), $A_{\mathsf{op}} \equiv A_1$ (8), $\Gamma, y : B_{\mathsf{op}} \vdash_{\mathsf{SD}} c_1 : A2'!\langle E'\rangle$ (9), and $\ell^{\mathsf{op}} \in E'$ (10). The goal follows from facts 1, 4, 5, 6, 7, 8, 10, constructors SD-Hand and SD-Op, and Lemma E.6.
- E-HandSc: By inversion on $\Gamma \vdash_{\mathsf{SD}} \mathbf{with}\ h\ \mathbf{handle\ op}\ \ell^{\mathsf{op}}\ v\ (y\,.\,c_1) : B!\langle F_2\rangle$ (SD-Hand) we have that $\Gamma \vdash_{\mathsf{SD}} h : \underline{C}_1 \Rightarrow B!\langle F_2\rangle$ (1), $\Gamma \vdash_{\mathsf{SD}} \ell^{\mathsf{sc}}\ v\ (y\,.\,c_1)\ (z\,.\,c_2) : \underline{C}_2$ (2), and $\underline{C}_1 \equiv \underline{C}_2$ (3). Inversion on fact 1 (SD-Handler) gives $B = M\ A_2$, $\underline{C}_1 = A_2!\langle E_1\rangle$, and $\Gamma, \alpha \vdash_M \mathit{oprs} : M\ \alpha!\langle F_2\rangle$ (4). Based on fact (3) we get that $\underline{C}_2 = A2'!\langle E_2\rangle$, $A_2 \equiv A2'$ (5), and $E_1 \equiv E_2$ (6). Inversion on fact 2 (SD-Sc) gives us $\ell^{\mathsf{sc}} : A_{\mathsf{sc}} \twoheadrightarrow B_{\mathsf{sc}} \in \Sigma$ (7), $\Gamma \vdash_{\mathsf{SD}} v : A_1$ (8), $A_{\mathsf{sc}} \equiv A_1$ (9), $\Gamma, y : B_{\mathsf{sc}} \vdash_{\mathsf{SD}} c_1 : A_3!\langle E_3\rangle$ (10), $\Gamma, z : A_3 \vdash_{\mathsf{SD}} c_2 : A2'!\langle E_2\rangle$ (11), $E_3 \equiv E_2$ (12), and $\ell^{\mathsf{sc}} \in E_2$ (13). Lemma E.8 we get that $\Gamma, \alpha \vdash_M \mathbf{op}\ \ell^{\mathsf{op}}\ x\ k \mapsto c, \mathit{oprs}_2 : M\ \alpha!\langle F_2\rangle$ (13.1). Inversion on fact 13.1 (SD-OprSc gives $\ell^{\mathsf{sc}} \in \Sigma$ (14), β *fresh* (15), $\Gamma, \alpha, \beta, x : A_{\mathsf{sc}}, p : B_{\mathsf{sc}} \to M\ \beta!\langle F_3\rangle, k : \beta \to M\ \alpha!\langle F_3\rangle \vdash_{\mathsf{SD}} c : M\ \alpha!\langle F_2\rangle$ (16), and $F_2 \equiv F_3$ (17). Facts 1, 5, 6, 10 and 12, constructors SD-Abs and SD-Hand and Lemmas E.6 and E.7 give us that $\Gamma, \beta \vdash_{\mathsf{SD}} \boldsymbol{\lambda} y\,.\,\mathbf{with}\ h\ \mathbf{handle}\ c_1 : p : B_{\mathsf{sc}} \to M\ \beta!\langle F_2\rangle$ (18). Facts 1, 5, 6 and 11 and constructors SD-Abs and SD-Hand and Lemma E.6 give us that $\Gamma, \beta \vdash_{\mathsf{SD}} \boldsymbol{\lambda} z\,.\,\mathbf{with}\ h\ \mathbf{handle}\ c_2 : \beta \to M\ A_2!\langle F_2\rangle$ (19). The goal now follows from facts 8, 9, 16, 17 and 18 and Lemma E.4.
- E-FwdSc By inversion on $\Gamma \vdash_{\mathsf{SD}} \mathbf{with}\ h\ \mathbf{handle\ op}\ \ell^{\mathsf{op}}\ v\ (y\,.\,c_1) : B!\langle F_2\rangle$ (SD-Hand) we have that $\Gamma \vdash_{\mathsf{SD}} h : \underline{C}_1 \Rightarrow B!\langle F_2\rangle$ (1), $\Gamma \vdash_{\mathsf{SD}} \ell^{\mathsf{sc}}\ v\ (y\,.\,c_1)\ (z\,.\,c_2) : \underline{C}_2$ (2), and $\underline{C}_1 \equiv \underline{C}_2$ (3). Inversion on fact 1 (SD-Handler) gives $B = M\ A_2$, $\underline{C}_1 = A_2!\langle E_1\rangle$, and $\Gamma, \alpha \vdash_M \mathbf{fwd}\ f\ p\ k \mapsto c_f : M\ \alpha!\langle F_2\rangle$ (4). Based on fact (3) we get that $\underline{C}_2 = A2'!\langle E_2\rangle$, $A_2 \equiv A2'$ (5), and $E_1 \equiv E_2$ (6). Inversion on fact 2 (SD-Sc) gives us $\ell^{\mathsf{sc}} : A_{\mathsf{sc}} \twoheadrightarrow B_{\mathsf{sc}} \in \Sigma$ (7), $\Gamma \vdash_{\mathsf{SD}} v : A_1$ (8), $A_{\mathsf{sc}} \equiv A_1$ (9), $\Gamma, y : B_{\mathsf{sc}} \vdash_{\mathsf{SD}} c_1 : A_3!\langle E_3\rangle$ (10), $\Gamma, z : A_3 \vdash_{\mathsf{SD}} c_2 : A2'!\langle E_2\rangle$ (11), $E_3 \equiv E_2$ (12), and $\ell^{\mathsf{sc}} \in E_2$ (13). Inversion on fact 4 (SD-Fwd) gives $A_p = \alpha' \to M\ \beta!\langle F_1\rangle$, $A'_p = \alpha' \to \gamma!\langle F_1\rangle$, $A_k = \beta \to M\ A_4!\langle F_1\rangle$, $A'_k = \gamma \to \delta!\langle F_1\rangle, \Gamma, \alpha, \alpha', \beta, p{:}A_p, k{:}A_k, f{:}\forall\ \gamma\ \delta\,.\,(A'_p, A'_k) \to \delta!\langle F_1\rangle \vdash_{\mathsf{SD}} c_f{:}M\ \alpha!\langle F_2\rangle$ (14) and $M\ A_1!\langle F_1\rangle \equiv M\ \alpha!\langle F_2\rangle$ (15). Facts 1, 6, 10 and 12, constructors SD-Abs and SD-Hand and Lemmas E.6 and E.7 give us that $\Gamma, \alpha, y{:}B_{\mathsf{sc}} \vdash_{\mathsf{SD}} \mathbf{with}\ h\ \mathbf{handle}\ c_1{:}M\ A_3!\langle F_2\rangle$ (16) Facts 1, 6, 10 and 12, constructors SD-Abs and SD-Hand and Lemma E.6 give us that $\Gamma, \alpha \vdash_{\mathsf{SD}} \boldsymbol{\lambda} z\,.\,\mathbf{with}\ h\ \mathbf{handle}\ c_2 : A_3 \to M\ A_4!\langle F_2\rangle$ (17) Facts 7, 8, 9, 13, the fact that $\ell^{\mathsf{sc}} \notin \mathit{labels}(\mathit{oprs})$, constructors SD-Abs, SD-App and SD-Var and Lemma E.6 give us that $\Gamma, \alpha, (p', k'){:}\forall\ \gamma\ \delta\,.\,(B_{\mathsf{sc}} \to \gamma!\langle F_1\rangle, \gamma \to \delta!\langle F_1\rangle) \vdash_{\mathsf{SD}} \mathbf{sc}\ \ell^{\mathsf{sc}}\ v\ (y\,.\,p'\ y)\ (z\,.\,k'\ z){:}\delta!\langle F_1\rangle$ (18). Our goal then follows from facts 14, 15, 16, 17, 18 and Lemma E.4.

□

E.3. **Progress.**

Theorem 8.2 (Progress)**.** *If* $\cdot \vdash c : A!\langle E\rangle$*, then either:*

- *there exists a computation* c' *such that* $c \rightsquigarrow c'$*, or*
- c *is in a* normal form*, which means it is in one of the following forms: (1)* $c = \mathbf{return}\ v$*, (2)* $c = \mathbf{op}\ \ell^{op}\ v\ (y\,.\,c')$ *where* $\ell^{op} \in E$*, or (3)* $c = \mathbf{sc}\ \ell^{sc}\ v\ (y\,.\,c_1)\ (z\,.\,c_2)$ *where* $\ell^{sc} \in E$*.*

Proof. By Lemma E.11 (above) and Theorem E.13 (below). □

Theorem E.13 (Syntax-directed Progress)**.** *If* $\cdot \vdash_{\mathsf{SD}} c : A!\langle E\rangle$*, then either:*

- *there exists a computation* c' *such that* $c \rightsquigarrow c'$*, or*
- c *is in a* normal form*, which means it is in one of the following forms: (1)* $c = \mathbf{return}\ v$*, (2)* $c = \mathbf{op}\ \ell^{op}\ v\ (y\,.\,c')$ *where* $\ell^{op} \in E$*, or (3)* $c = \mathbf{sc}\ \ell^{sc}\ v\ (y\,.\,c_1)\ (z\,.\,c_2)$ *where* $\ell^{sc} \in E$*.*

Proof. By induction on the typing derivation $\cdot \vdash_{\mathsf{SD}} c : \underline{C}$.

- SD-App: Here, $\cdot \vdash_{\mathsf{SD}} v_1\ v_2$. Since v_1 has type $A \to B!\langle F\rangle$, by Lemma E.1 it must be of shape $\boldsymbol{\lambda} x\,.\,c$, which means we can step by rule E-AppAbs.
- SD-Do: Here, $\cdot \vdash_{\mathsf{SD}} \mathbf{do}\ x \leftarrow c_1\ \mathbf{in}\ c_2 : \underline{C}$. By the induction hypothesis, c_1 can either step (in which case we can step by E-Do), or it is a computation result. Every possible form has a corresponding reduction: if $c_1 = \mathbf{return}\ v$ we can step by E-DoRet, if $c_1 = \mathbf{op}\ \ell^{\mathsf{op}}\ v\ (y\,.\,c)$ we can step by E-DoOp, and if $\mathbf{sc}\ \ell^{\mathsf{op}}\ v\ (y\,.\,c_1')\ (z\,.\,c_2')$ we can step by E-DoSc.
- SD-Let: Here, $\cdot \vdash_{\mathsf{SD}} \mathbf{let}\ x = v\ \mathbf{in}\ c : \underline{C}$, which means we can step by E-Let.
- SD-Ret, SD-Op, and SD-Sc: all of these are computation results (forms (1), (2), and (3), resp.). Since they are well-typed, the scoping condition of ℓ^{op} or ℓ^{sc} must be satisfied.
- SD-Hand: Here $\cdot \vdash_{\mathsf{SD}} \mathbf{with}\ v\ \mathbf{handle}\ c : M\ A!\langle F\rangle$. By Lemma E.1, v is of shape h. By the induction hypothesis, c can either step (in which case we can step by E-Hand), or it is in a normal form. Proceed by case split on the three forms.
  (1) Case $c = \mathbf{return}\ v$. Since $\cdot \vdash_{\mathsf{SD}} h : \underline{C} \Rightarrow \underline{D}$, there must be some $(\mathbf{return}\ x \mapsto c_r) \in h$ which means we can step by rule E-HandRet.
  (2) Case $c = \mathbf{op}\ \ell^{\mathsf{op}}\ v\ (y\,.\,c')$. Depending on $(\mathbf{op}\ \ell^{\mathsf{op}}\ x\ k \mapsto c) \in h$ we can step by E-HandOp or E-FwdOp.
  (3) Case $c = \mathbf{sc}\ \ell^{\mathsf{sc}}\ v\ (y\,.\,c_1)\ (z\,.\,c_2)$. If $(\mathbf{sc}\ \ell^{\mathsf{sc}}\ x\ p\ k \mapsto c) \in h$, we can step by E-HandSc. If not, since $\cdot \vdash_{\mathsf{SD}} h : \underline{C} \Rightarrow \underline{D}$, there must be some $(\mathbf{fwd}\ f\ p\ k \mapsto c_f) \in h$ which means we can step by rule E-FwdSc. □