A Calculus for Scoped Effects & Handlers

ANONYMOUS AUTHOR(S)

Algebraic effects & handlers have become a standard approach for working with side-effects in functional programming languages. Their modular composition with other effects and clean separation of syntax and semantics make them attractive to a wide audience of programmers. 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 with other effects brings in additional complexity. This class of effects is called scoped effects and has gained attention by their growing applicability and adoption in popular libraries. Although calculi have been designed with algebraic effects & handlers as first-class operations, effectively easing programming with those features, a calculus to support scoped effects & handlers in a similar matter is missing from the literature. In this work we fill this gap: we present a novel calculus, named λ_{sc}, in which both algebraic and scoped effects & handlers are first-class. This involves the need for polymorphic handlers and explicit forwarding clauses to forward unknown scoped operations to other 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 (type-safe) type-and-effect system. We demonstrate the usability of our work by discussing a range of scoped effect examples. In summary, our novel calculus allows modular composition of different algebraic and scoped effects & handlers and eases programming with these features.

1 INTRODUCTION

Whereas monads [Moggi 1989, 1991; Wadler 1995] have long been the standard approach for modelling effects, algebraic effects & handlers [Plotkin and Pretnar 2009; Plotkin and Power 2003] are gaining steadily more traction. They offer a more structured and *modular* approach to composing effects that is based on an algebraic model. Algebraic effects & handlers consist of two parts: effects denote the syntax of operations and handlers interpret them by means of structural recursion. This clean separation is a coveted property for programming with effects. Each handler is appointed to interpret its part of the syntax and forwards the unknown parts to other handlers. This allows a programmer to write dedicated handlers for each effect that occurs in the program (e.g., state, interactive input/output, nondeterminism). Different effect handlers can give a different interpretation to the same effectful operation. By means of composing the handlers in the desired order, one can modularly build an interpretation for the entire program. The idea of algebraic effects & handlers and their modular composition has been adopted by many libraries in functional programming, e.g., fused-effects [Rix et al. 2018], extensible-effects [Kiselyov et al. 2019] and Eff in OCaml [Kiselyov and Sivaramakrishnan 2018]. Additionally, various languages support programming with algebraic effects & handlers, e.g., Links [Hillerström and Lindley 2016], Koka [Leijen 2017], and Effekt [Brachthäuser et al. 2020].

Although the modular approach of algebraic effects & handlers is desirable for every effectful program, it is not applicable to all kinds of effects. In particular, those effects that have or introduce a delimited scope (e.g., exceptions, concurrency, local state) are not algebraic. By this scope, these so-called *scoped effects* [Wu et al. 2014] essentially split the program in two parts: one scoped

0004-5411/2022/1-ART1 $15.00
https://doi.org/XXXXXXX.XXXXXXX

computation where the effect is in scope, and a continuation where it is out of scope. This separation is broken by the algebraicity property, which means we cannot "just" include scoped effects in the algebraic framework. Other workarounds, such as modeling scoped effects as handlers, or encoding the scoped computation in the parameter of the algebraic operation, come at the cost of modularity. Instead, a calculus that considers scoped effects & handlers first-class is required. The growing interest in scoped effects & handlers, evidenced by their adoption in several Haskell libraries (e.g., eff [King 2019], polysemy [Maguire 2019], fused-effects [Rix et al. 2018]), motivates the imminent need for such a calculus.

This paper aims to fill this gap in the literature: we present a novel calculus, called λ_{sc}, that puts scoped effects & handlers on formal footing. Our main source of inspiration is Eff [Bauer and Pretnar 2013, 2015; Pretnar 2015], which introduces a calculus for first-class 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. First, we use Koka-style row polymorphism to keep track of effects. Second, scoped effects require the inclusion of polymorphic handlers, which we support by adding let-polymorphism and F_ω-style type operators to λ_{sc}. Finally, in order to keep the desired modularity, and similar to algebraic handlers, we need to be able to forward unknown operations. Whereas algebraic effects & handlers have a generic (and implicit) forwarding mechanism, scoped effects & handlers need more sophisticated machinery, in order to allow sufficient freedom in their implementation. We address this by adding an explicit forwarding clause for scoped effects to all handlers.

In what follows, we formalize λ_{sc} by means of its syntax, semantics, type-and-effect system and metatheoretical properties. In particular, after introducing the reader to the appropriate background and informally motivating the need for our calculus (Section 2), we formally present the syntax of a variant of Eff (Section 3), the starting point for our calculus. From there, we make the following contributions:

- We design a formal syntax for λ_{sc} terms, types and contexts (Section 4).
- We provide an operational semantics (Section 5)[1].
- We formulate the type-and-effect system of λ_{sc}, and its metatheoretical properties (Section 6).
- We show the usability of our calculus on a range of examples (Section 7).
- We provide a Haskell implementation of our examples and an interpreter of our calculus.

Finally, we discuss related work (Section 8) and conclude (Section 9).

2 BACKGROUND, MOTIVATION & CHALLENGES

This section provides the necessary background and motivates the goal of this paper. First, we briefly summarize *algebraic effects & handlers*, a modular approach to composing side-effects in effectful programming. Next, we discuss *scoped effects*: effects that have or create a delimited scope (such as `once` for nondeterminism). We show how these cannot be modelled as algebraic effects. We argue that the current workarounds to implement these scoped effects are insufficient, and offer our solution: a calculus that allows programming with first-class scoped effects & handlers. We conclude this section with an overview of the challenges that arise from adding support for scoped effects: our calculus requires polymorphic handlers and explicit forwarding clauses.

[1]Note that, in order to smoothly guide the reader through the complex notion of forwarding for scoped effects, we first introduce the operational semantics, before unraveling the type-and-effect system.

2.1 Algebraic Effects & Handlers

Algebraic effects & handlers allow overloading, modularity and interactions between different effects. As the name suggests, they consist of two parts: algebraic effects (*operations*) and *handlers*.

Algebraic Operations. Firstly, *operations* represent effectful primitives. The signature $A \twoheadrightarrow B$ denotes an effect taking a value of type A and producing a value of type B. For example, the operation $\texttt{choose} : () \twoheadrightarrow \mathsf{Bool}$ takes a unit value and produces a boolean (e.g., randomly or nondeterministically). Syntactically, an operation is invoked as follows:

$$\textbf{op}\ \texttt{choose}\ ()\ (y\,.\,c)$$

Here, we have the **op** keyword, followed by the *label* `choose` that identifies the particular operation. Then, we have a *parameter*—in this case ()—, and a *continuation* $(y\,.\,c)$. This continuation c contains the remainder of the program, taking the result of the operation as argument y. For `choose`, the type of y is Bool, and c uses this boolean for its further computation. For example, one can use `choose` to write a computation c_{ND} that returns "heads" or "tails", depending on the result:

$$c_{ND} = \textbf{op}\ \texttt{choose}\ ()\ (b\,.\,\textbf{if}\ b\ \textbf{then return}\ \texttt{"heads"}\ \textbf{else return}\ \texttt{"tails"})$$

Using a row-based typing approach like that of Koka [Leijen 2017], c_{ND} can be given the type $\mathsf{String}\,!\,\langle\texttt{choose}\rangle$. This *computation type* consists of a *value type* String, indicating that c_{ND} produces a string, and an *effect row* $\langle\texttt{choose}\rangle$, indicating that c_{ND} may invoke the `choose` operation.

The "algebraic" adjective in algebraic operations is due to an equivalence they share: algebraic operations commute with sequencing. For example, observe the following equivalence for `choose`.

$$\begin{aligned}&\textbf{do}\ x \leftarrow \textbf{op}\ \texttt{choose}\ ()\ (b\,.\,\textbf{if}\ b\ \textbf{then return}\ \texttt{"heads"}\ \textbf{else return}\ \texttt{"tails"})\,;\textbf{return}\ x + x\\ \equiv\ &\textbf{op}\ \texttt{choose}\ ()\ (b\,.\,\textbf{do}\ x \leftarrow \textbf{if}\ b\ \textbf{then return}\ \texttt{"heads"}\ \textbf{else return}\ \texttt{"tails"}\ \,;\textbf{return}\ x + x)\end{aligned}$$

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

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

Handlers. Next, *handlers* give meaning to operations, by stating how to interpret operations that occur in a computation. For example, handler h_{ND} interprets `choose` nondeterministically:

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

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_{ND} with the $\star$-operator to obtain both of its results:

```
>>> h_ND ⋆ c_ND
["heads","tails"]
```

A possible type for the handler is $h_{ND} : \mathsf{String}\,!\,\langle\texttt{choose}\rangle \Rightarrow \mathsf{List}\ \mathsf{String}\,!\,\langle\rangle$, which expresses that the handler interprets a computation of type $\mathsf{String}\,!\,\langle\texttt{choose}\rangle$ into a computation of type $\mathsf{List}\ \mathsf{String}\,!\,\langle\rangle$. However, for the sake of reusability, handlers often abstract over the value type (in this case String), so that they can handle computations of any type, as long as the effect rows match.

$h_{ND} : \forall\ \alpha\ .\ \alpha\,!\,\langle \texttt{choose} \rangle \Rightarrow \mathsf{List}\ \alpha\,!\,\langle\rangle$

Algebraic effects & handlers bring several interesting advantages and features. The clear separation between the syntax of effects and their semantics allows for (1) overloading the interpretation of operations, (2) modularly composing different effects and (3) altering the meaning of a program by different effect interactions. In what follows, we discuss these features in more detail.

Overloading. Decoupling operations from their interpretation by handlers has as advantage that different contexts can provide different interpretations, giving a mechanism for overloading. For example, $h_{1st} : \forall\ \alpha\ .\ \alpha\,!\,\langle \texttt{choose} \rangle \Rightarrow \alpha\,!\,\langle\rangle$ is a variant of h_{ND} that, instead of a list of all possible results, retains only a single result:

$$\begin{aligned} h_{1st} = \mathbf{handler}\ \{&\mathbf{return}\ x \quad\ \mapsto \mathbf{return}\ x \\ ,\ &\mathbf{op}\ \texttt{choose}\ \_\ k \mapsto k\ \mathsf{true}\} \end{aligned}$$

We apply h_{1st} to c_1 to obtain only the first result:

```
>>> h1st ★ cND
"heads"
```

Modular Composition. Different effects can be composed by combining different primitive operations in the same computation. For example, computation $c_{c,g} : \mathsf{String}\,!\,\langle \texttt{choose}\,;\texttt{get} \rangle$ below uses $\texttt{get} : () \rightarrowtriangle \mathsf{String}$ in addition to `choose`.

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

If we want to handle `get` while maintaining the semantics for `choose`, we can write a handler $h_{c,g} : \forall\ \alpha\ .\ \alpha\,!\,\langle \texttt{choose}\,;\texttt{get} \rangle \Rightarrow \mathsf{List}\ \alpha\,!\,\langle\rangle$ that repeats the clauses of h_{ND} and adds a clause for `get`:

$$\begin{aligned} h_{c,g} = \mathbf{handler}\ \{&\mathbf{return}\ x \quad\ \mapsto \mathbf{return}\ [x] \\ ,\ &\mathbf{op}\ \texttt{choose}\ \_\ k \mapsto \mathbf{do}\ xs \leftarrow k\ \mathsf{true}\,;\mathbf{do}\ ys \leftarrow k\ \mathsf{false}\,;xs \mathbin{+\!\!+} ys \\ ,\ &\mathbf{op}\ \texttt{get}\ \_\ k \quad\ \mapsto k\ \texttt{"surprise"}\} \end{aligned}$$

However, it is more modular and reusable to write separate handlers for both effects, and compose their applications. For example, we can separately define h_{get}.

$$\begin{aligned} h_{get} = \mathbf{handler}\ \{&\mathbf{return}\ x \quad\ \mapsto \mathbf{return}\ x \\ ,\ &\mathbf{op}\ \texttt{get}\ \_\ k \mapsto k\ \texttt{"surprise"}\} \end{aligned}$$

Operationally, $h_{get} \star c_{c,g}$ evaluates into the following.

```
>>> hget ★ cc,g
op choose () (b . if b then return "heads" else return "surprise")
```

We see that h_{get} leaves (we say *"forwards"*) the `choose` operation to be handled by some other handler. This forwarding behavior is key to the modular reuse and composition of handlers. We can compose h_{get} and h_{ND} in order to interpret $c_{c,g}$.

```
>>> hND ★ (hget ★ cc,g)
["heads","surprise"]
```

This modular composition is enabled by the row-polymorphic type of $h_{get} : \forall\ \alpha\ \mu\ .\ \alpha\,!\,\langle \texttt{get}\,;\mu \rangle \Rightarrow \mathsf{List}\ \alpha\,!\,\langle \mu \rangle$, where μ is a row variable standing for any row, such as the empty row or `choose` in this example.[2]

[2] In fact, h_{ND} can similarly be given a more general type $h_{ND} : \forall\ \alpha\ \mu\ .\ \alpha\,!\,\langle \texttt{choose}\,;\mu \rangle \Rightarrow \mathsf{List}\ \alpha\,!\,\langle \mu \rangle$.

Effect Interactions. One of the valuable features of modular composition is that effects can be made to interact differently by applying their handlers in a different order. Consider the $\texttt{inc} : () \twoheadrightarrow \mathsf{Int}$ operation, which produces an (incremented) integer. The handler h_{inc} turns computations into state-passing functions.

$$\begin{aligned} &h_{inc} \;:\; \forall\, \alpha\; \mu\,.\, \alpha\,!\,\langle \texttt{inc} ; \mu\rangle \Rightarrow (\mathsf{Int} \to (\alpha, \mathsf{Int})\,!\,\langle\mu\rangle)\,!\,\langle\mu\rangle \\ &h_{inc} = \mathbf{handler}\,\{\mathbf{return}\; x \quad\;\; \mapsto \mathbf{return}\; (\boldsymbol{\lambda} s\,.\,\mathbf{return}\; (x, s)) \\ &\qquad\qquad\quad , \mathbf{op}\; \texttt{inc}\; \_\; k \mapsto \mathbf{return}\; (\boldsymbol{\lambda} s\,.\, k\; s\; (s+1))\,\} \end{aligned}$$

The state s represents the current counter value. On every occurrence of operation `inc`, both the current counter value and its increment are given to the continuation. The former is for the continuation and the latter for serving the next `inc` operation. We use syntactic sugar to apply the initial counter value to the results of the handler h_{inc}.

$$run_{inc}\; s\; c \;\equiv\; \mathbf{do}\; c' \leftarrow h_{inc} \star c\,;\, c'\; s$$

Computation $c_{inc} : \mathsf{Int}\,!\,\langle \texttt{choose} ; \texttt{inc}\rangle$ combines `choose` and `inc`:

$$c_{inc} = \mathbf{op}\; \texttt{choose}\; ()\; (b\,.\,\mathbf{if}\; b\; \mathbf{then}\; \mathbf{op}\; \texttt{inc}\; ()\; (x\,.\,\mathbf{return}\; x)\; \mathbf{else}\; \mathbf{op}\; \texttt{inc}\; ()\; (y\,.\,\mathbf{return}\; y))$$

Operationally, $run_{inc}\; 0\; c_{inc}$ starts with a counter with value zero and evaluates into the following.

$$\begin{aligned} &>\!>\!> run_{inc}\; 0\; c_{inc} \\ &\mathbf{op}\; \texttt{choose}\; ()\; (b\,.\,\mathbf{if}\; b\; \mathbf{then}\; \mathbf{return}\; (0,1)\; \mathbf{else}\; \mathbf{return}\; (0,1)) \end{aligned}$$

Notice that handling an expression with h_{inc} handles *all* occurrences of `inc`. This concept is known as *deep handling*.

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

$$\begin{aligned} &>\!>\!> h_{ND} \star run_{inc}\; 0\; c_{inc} \\ &[\,(0,1), (0,1)\,] \end{aligned}$$

In contrast, when handling `choose` first, the counter value is threaded through the successive branches.

$$\begin{aligned} &>\!>\!> run_{inc}\; 0\; (h_{ND} \star c_{inc}) \\ &([\,0,1\,], 2) \end{aligned}$$

Both kinds of interactions have their uses. In general, effect interaction is an asset that algebraic effects & handlers—just like monad transformers [Gill 2008]—provide both with minimal effort and in a modular, compositional way.

2.2 Scoped Operations

Not all effect operations satisfy the algebraicity property, as already realised by Plotkin and Power [2003]. Consider the $\texttt{once} : () \twoheadrightarrow ()$ operation, which takes a computation that calls the `choose` operation and returns only its first result [Piróg et al. 2018a]. We could attempt to syntactically write this as the algebraic operation $\mathbf{op}\; \texttt{once}\; ()\; (y\,.\,c)$ and a corresponding handler. In fact, this handler is an extension of h_{ND}, with a new clause for `once`.

$$h_{once} = \mathbf{handler}\,\{\,\ldots, \mathbf{op}\; \texttt{once}\; \_\; k \mapsto \mathbf{do}\; ts \leftarrow k\; ()\,;\, \mathsf{head}\; ts\}$$

Yet, the effect we intend to model with `once` does not satisfy the algebraicity property:

$$\mathbf{do}\; x \leftarrow \mathbf{op}\; \texttt{once}\; ()\; (y\,.\,c_1)\,;\, c_2 \;\not\equiv\; \mathbf{op}\; \texttt{once}\; ()\; (y\,.\,\mathbf{do}\; x \leftarrow c_1\,;\, c_2)$$

Indeed, we do not intend c_2 to be affected by `once`; the effect of `once` should be limited to c_1. Hence, by definition, `once` is not an algebraic operation. There are many more examples of such non-algebraic operations, e.g.,

- `call` for creating a scope in a nondeterministic program, where branches can be cut using the algebraic `cut` operation (Section 7.2).
- `catch` for catching exceptions that are raised during program execution (Section 7.3);
- `local` for creating local variables (local state) (Section 7.4);
- `depth` for bounding the depth in the depth-bounded search strategy (Section 7.5).

What these operations have in common is that they have a delimited scope, which splits the rest of the program into two parts: a computation for which the effect is in scope, and a continuation where it is out of scope. Following Wu et al. [2014] we call them *scoped operations*.

Technically, scoped effects that are modeled as algebraic operations, like `once` above, implement valid algebraic effects. Yet, the behavior of these operations does not correspond to the one we intend them to have. Several workarounds have been proposed to circumvent the non-algebraicity of these operations. First, one could model scoped operations as handlers. Alternatively, one could model them as parameters of algebraic operations. However, both workarounds come at the expense of modularity.

Scoped Operations as Handlers. Plotkin and Pretnar [2009] have proposed to incorporate scoped operations in the algebraic effects framework by modelling them as handlers instead of operations.

$$\texttt{once}\ c \ \equiv\ h_{1st} \star c$$

Yet, this way, scoped operations lose the flexibility of algebraic operations that comes from separating the operation calls from assigning meaning to these calls. Firstly, by fixing a handler, scoped operations cannot be given different meanings (overloading). Secondly, as the position of the handler application is fixed by the call to the scoped operation, its interaction with other effects cannot be changed by altering the order of handler applications. Hence, by implementing scoped operations as handlers, they become second-class citizens that do not have the same modular reusability as algebraic operations.

Scoped Computations as Parameters of Algebraic Operations. Another workaround is to model scoped effects as algebraic operations in which the parameter is the scoped computation and to use general recursion to handle them. For example, we replace the original parameter $()$ of `once` with a pair of $()$ and the scoped computation c_1. [3]

$$\textbf{op}\ \texttt{once}\ ((), c_1)\ (y\,.\,c_2)$$

Again, this works well in isolation, but does not have the modularity benefits of regular algebraic operations. When applying the handler h of another effect to this encoding, the deep handling mechanism only applies h recursively to the continuation c_2 and not to the scoped computation c_1. The reason for this is that the algebraic effects framework does not reveal whether or not the parameter $((), c_1)$ of an unknown operation contains a scoped computation.

To conclude, none of these encodings are sufficient to model scoped effects so that they behave as intended and can be composed in the same modular way as algebraic effects can. Instead, we require a calculus for first-class scoped effects and handlers.

[3] We can also directly remove the $()$ in this example. Bu other scoped operations may have non-trivial parameters which should not be removed.

2.3 A Calculus for Scoped Effects and Handlers

This paper aims to preserve the good properties of algebraic operations for scoped operations. It follows a line of work [Piróg et al. 2018a; Wu et al. 2014; Yang et al. 2022] that has developed denotational semantic domains, backed by category-theorical models. What is lacking from the literature is a calculus that supports scoped effects as *first-class operations*. This work develops such a calculus, called λ_{sc}, which allows programming with both algebraic and scoped operations and their handlers, and, in particular, supports their modular composition. The design of λ_{sc} is inspired by Eff [Bauer and Pretnar 2013, 2015; Pretnar 2015], adopts the row-based typing approach of Koka [Leijen 2017] and incorporates polymorphic types and type operators à la System F_ω.

We extend Eff, which only supports algebraic operations, with a new construct for representing scoped operations, and define a variant of the algebraicity property to express the desired relation between scoped operations and sequencing.

Scoped Operations. To realize the delimited scope of operations like `once`, we introduce a new notation for those operations, signalled by the **sc** keyword.

$$\textbf{sc}\ \texttt{once}\ ()\ (y\,.\,c_1)\ (z\,.\,c_2)$$

Similar to algebraic operations, scoped operations feature a label `once` to identify the operation, a parameter—in this case ()—, and a continuation $(z\,.\,c_2)$. However, scoped operations differ from algebraic operations by their additional *scoped computation* $(y\,.\,c_1)$. The idea is that `once` affects the choices in this scoped computation c_1, but not in the continuation c_2. The dataflow allows for a value y to be passed from the operation to c_1 and a value z from c_1 to c_2.

As a consequence of this changed notation, the signature of scoped operations also differ. For a signature $A \twoheadrightarrow B$ of a scoped operation, B now refers to the type of the scoped computation's argument. For example, `once` has signature $() \twoheadrightarrow ()$, where the first () refers to the type of the parameter of the scoped operation and the second () refers to the type of the variable y in $\textbf{sc}\ \texttt{once}\ ()\ (y\,.\,c_1)\ (z\,.\,c_2)$.

We can construct a scoped operation as follows:

$$\begin{aligned} c_{once} = \textbf{sc}\ \texttt{once}\ ()\ (\_\,.\,\textbf{op}\ \texttt{choose}\ ()\ (x\,.\,\textbf{return}\ x)) \\ (b\,.\,\textbf{if}\ b\ \textbf{then return}\ \texttt{"heads"}\ \textbf{else return}\ \texttt{"tails"}) \end{aligned}$$

In a similar way as for algebraic operations, we type this computation c_{once}:$\mathsf{String}\,!\,\langle \texttt{choose}\,;\texttt{once}\rangle$. Accordingly, we redefine the handler h_{once} with `once` as a *scoped* operation.

$$\begin{aligned} h_{once} = \textbf{handler}\ \{&\textbf{return}\ x \mapsto \textbf{return}\ [x] \\ &,\textbf{op}\ \texttt{choose}\ \_\ k \mapsto \textbf{do}\ xs \leftarrow k\ \mathsf{true}\,;\textbf{do}\ ys \leftarrow k\ \mathsf{false}\,;xs \mathbin{+\!\!+} ys \\ &,\textbf{sc}\ \texttt{once}\ \_\ p\ k \mapsto \textbf{do}\ ts \leftarrow p\ ()\,;\textbf{do}\ t \leftarrow \mathsf{head}\ ts\,;k\ t\} \end{aligned}$$

Evaluating scoped computation p yields all results inside the scope as *ts*. Then, head implements the behavior of `once`: it only keeps the first result. For example, $h_{once} \star c_{once}$ evaluates to the following:

```
>>> h_once ★ c_once
["heads"]
```

Algebraicity. Adding scoped operations gives rise to a variant of the algebraicity property, specific to scoped effects, which models the desired behavior of sequencing in combination with scoped operations. Scoped operations commute with sequencing in the continuation, but leave the scoped computation intact.

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

Scoped operation `once` satisfies this property.

$$\textbf{do}\ x \leftarrow \textbf{sc}\ \texttt{once}\ ()\ (y\,.\,c_1)\ (z\,.\,c_2)\,;c_3\ \equiv\ \textbf{sc}\ \texttt{once}\ ()\ (y\,.\,c_1)\ (z\,.\,\textbf{do}\ x \leftarrow c_2\,;c_3)$$

2.4 Polymorphism and Forwarding

When designing a calculus with first-class scoped operations & handlers, two challenges arise.

If an operation is encountered while deeply handling a computation, the handler is recursively applied to all of this operation's subcomputations. Whereas algebraic operations contain a single subcomputation, scoped operations contain *two* of them: the scoped computation and the continuation. The result of the scoped computation is the argument of the continuation. This means they must agree on a type of this result, which we name the *scoped result type*. For example, consider c_{once}, with overall type $\mathsf{String}\,!\,\langle \texttt{once}\,;\texttt{choose}\rangle$. It comprises the scoped result type Bool, which is returned by the scoped computation, and expected by the continuation.

$$\textbf{sc}\ \texttt{once}\ ()\ \underbrace{\_\,.\,\textbf{op}\ \texttt{choose}\ ()\ (x\,.\,\textbf{return}\ x)}_{()\to \mathsf{Bool}\,!\,\langle \texttt{once}\,;\texttt{choose}\rangle}\ \underbrace{b\,.\,\textbf{if}\ b\ \textbf{then return}\ \texttt{"heads"}\ \textbf{else}\ \ldots}_{\mathsf{Bool}\to \mathsf{String}\,!\,\langle \texttt{once}\,;\texttt{choose}\rangle}$$

This fact introduces two complications, which we discuss in the remainder of this section.

Polymorphic Handlers. Evaluating the application of a handler to some operation involves recursively applying the handler to the operation'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, even though we have used polymormic handlers in our presentation of algebraic effects, calculi that only support algebraic effects & handlers, such as Eff, can (and do) type handlers monomorphically, without severe limitations.

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 a monomorphic typing of handler h_{once} when applied to computation $c_{once} : \mathsf{String}\,!\,\langle \texttt{choose}\,;\texttt{once}\rangle$:

$$h_{once} : \mathsf{String}\,!\,\langle \texttt{choose}\,;\texttt{once}\rangle \Rightarrow \mathsf{List}\ \mathsf{String}\,!\,\langle\rangle$$

This monomorphic type requires the scoped result type to be String as well, as it is the only type of computation monomorphic h_{once} can handle, which 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 without considerably limiting their implementation freedom.

The solution is to let handlers abstract over the value type of computations. This allows for scoped operations with *any* scoped result type. This way, h_{once} can be typed as follows:

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

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

Forwarding Unknown Operations. In order to retain the modularity of composing different effects, as discussed in Section 2.1, we write dedicated handlers that interpret only their part of the syntax, and *forward* all remaining operations to other handlers. For algebraic effects forwarding

happens generically. For example, consider the forwarding of h_{once} applied to an algebraic operation with $\texttt{inc} : () \twoheadrightarrow \mathsf{Int}$.

$$h_{once} \star \mathbf{op}\ \texttt{inc}\ ()\ (y\,.\,\mathbf{return}\ y) \rightsquigarrow \mathbf{op}\ \texttt{inc}\ ()\ (y\,.\,h_{once} \star \mathbf{return}\ y)$$

As the continuation may contain other effects, h_{once} is recursively reapplied to this continuation. The original operation **op** `inc` is preserved to be handled by other handlers. Notice that this transformation maintains the type of the overall computation.

One might hope to forward scoped effects in a similar way. For example, consider applying h_{once} to a scoped operation $\texttt{catch} : \mathsf{String} \twoheadrightarrow \mathsf{Bool}$ for catching exceptions.

$$h_{once} \star \mathbf{sc}\ \texttt{catch}\ \texttt{"oops"}\ (b\,.\,\mathbf{if}\ b\ \mathbf{then\ return}\ 5\ \mathbf{else}\ \ldots)\ (x\,.\,\mathbf{return}\ x)$$
$$\rightsquigarrow$$
$$\mathbf{sc}\ \texttt{catch}\ \texttt{"oops"}\ (b\,.\,h_{once} \star \mathbf{if}\ b\ \mathbf{then\ return}\ 5\ \mathbf{else}\ \ldots)\ (x\,.\,h_{once} \star \mathbf{return}\ x)$$

Unfortunately, this does not work. Again, the hurdle is in the scoped result type. In particular, h_{once} introduces a type constructor List when handling a computation. The scoped computation now has type $\mathsf{Bool} \to \mathsf{List\ Int}\,!\,\langle\texttt{catch}\rangle$, whereas the continuation has type $\mathsf{Int} \to \mathsf{List\ Int}\,!\,\langle\texttt{catch}\rangle$.

$$\mathbf{sc}\ \texttt{catch}\ \texttt{"oops"}\ \underbrace{(b\,.\,h_{once} \star \mathbf{if}\ b\ \mathbf{then\ return}\ 5\ \mathbf{else}\ \ldots)}_{\mathsf{Bool} \to \mathsf{List\ Int}\,!\,\langle\texttt{catch}\rangle}\ \underbrace{(x\,.\,h_{once} \star \mathbf{return}\ x)}_{\mathsf{Int} \to \mathsf{List\ Int}\,!\,\langle\texttt{catch}\rangle}$$

Indeed, applying a handler to a computation changes its type: not only does it remove labels from the effect row, it also applies a type constructor M—in this case List—to the type. For scoped operations this is problematic, as the return type of the scoped computation has changed (List Int), whereas the continuation still expects the original type (Int).

Thus, scoped effects cannot be forwarded generically. Therefore, we require that every handler is equipped with an explicit forwarding clause for unknown scoped operations. When a handler is defined it is clear what type constructor M the handler applies. Hence, it can provide an M-specific way of bridging between $M\ X$ and X.

For example, h_{once} mitigates the discrepancy between List Int and Int by settling the scoped result type on List Int as a compromise. Indeed, concatMap ensures that the transformed continuation now takes a value of type List Int as argument. The forwarding clause of h_{once} is the following:

$$h_{once} = \mathbf{handler}\ \{\ \ldots, \mathbf{fwd}\ f\ p\ k \mapsto f\ (p, (\boldsymbol{\lambda} z\,.\,\mathsf{concatMap}\ z\ k))\ \}$$

Here, f essentially is a partial application of $\mathbf{sc}\ \ell\ v$: it takes the pair of a (possibly transformed) scoped computation p' and continuation k' and re-introduces the original scoped operation with these parameters.

$$f = \boldsymbol{\lambda}(p', k')\,.\,\mathbf{sc}\ \ell\ v\ (y\,.\,p'\ y)\ (z\,.\,k'\ z)$$

This reflects the same idea as for algebraic operations: the original (unknown) operation is preserved to be handled by other handlers, but the transformations of the scoped computation and continuation resolve the disparity between their types.

Towards λ_{sc}. In summary, the challenges that come with supporting scoped effects necessitate support for both polymorphic handlers and explicit forwarding in λ_{sc}, which the next sections formally introduce. We take an approach that follows our informal discussion in this section. First, we present our starting point: a calculus that features algebraic effects & handlers, extended with row-typing in the style of Koka (Section 3). Then we present λ_{sc}, focussing on the extensions made to our starting point to support scoped effects & handlers (Sections 4 to 6).

$$
\begin{array}{rcll}
\text{values } v & ::= & () \mid (v_1, v_2) \mid x \mid \boldsymbol{\lambda} x\,.\,c \mid h & \\
\text{handlers } h & ::= & \mathbf{handler}\,\{\mathbf{return}\ x \mapsto c & \text{return clause} \\
 & & \quad , \mathit{oprs}\} & \text{effect clauses} \\
\mathit{oprs} & ::= & \cdot & \\
 & \mid & \mathbf{op}\ \ell\ x\ k \mapsto c, \mathit{oprs} & \text{algebraic effect clauses} \\
 & & & \\
\text{computations } c & ::= & \mathbf{return}\ v & \text{return value} \\
 & \mid & \mathbf{op}\ \ell\ v\ (y\,.\,c) & \text{algebraic operation} \\
 & \mid & v \star c & \text{handle} \\
 & \mid & \mathbf{do}\ x \leftarrow c_1\,;\,c_2 & \text{do-statement} \\
 & \mid & v_1\ v_2 & \text{application} \\
 & & & \\
\text{value types } A, B & ::= & () \mid (A, B) & \text{unit type, pair type} \\
 & \mid & A \to \underline{C} & \text{function type} \\
 & \mid & \underline{C} \Rightarrow \underline{D} & \text{handler type} \\
 & & & \\
\text{type schemes } \sigma & ::= & A \mid \forall\, \mu\,.\,\sigma & \\
\text{computation types } \underline{C}, \underline{D} & ::= & A\,!\,\langle E \rangle & \\
\text{effect rows } E, F & ::= & \cdot \mid \mu \mid \ell\,;E & \\
 & & & \\
\text{signature contexts } \Sigma & ::= & \cdot \mid \Sigma, \ell : A \twoheadrightarrow B & \\
\text{type contexts } \Gamma & ::= & \cdot \mid \Gamma, x : A \mid \Gamma, \mu &
\end{array}
$$

Fig. 1. Term and type syntax of Eff.

3 STARTING POINT: A CALCULUS FOR ALGEBRAIC EFFECTS AND HANDLERS

The starting point is a version of Eff [Bauer and Pretnar 2013, 2015], a calculus supporting algebraic effects & handlers. Our version (Figure 1) differs from Bauer and Pretnar [2013, 2015] in two ways. Firstly, we have made a number of cosmetic changes that improve the readability of the calculus, which becomes especially apparent once we present our extensions to support scoped effects & handlers. Secondly, this version features row-based typing in the style of Koka [Leijen 2017].

3.1 Terms

Implementing fine-grained call-by-value [Levy et al. 2003], terms are split into values and computations, the latter of which can be reduced and thus can be effectful.

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 two kinds of clauses: one return clause, and zero or more operation clauses. The return clause $\mathbf{return}\ x \mapsto c$ denotes that the result x of a computation is processed by computation c. Algebraic operation clauses $\mathbf{op}\ \ell\ x\ k \mapsto c$ specify that handling an effect with label ℓ, parameter x and continuation k is processed by computation c (e.g., h_{ND}, h_{get}, h_{inc}).

Computations may return a value v using the **return** keyword. Algebraic operations $\mathbf{op}\ \ell\ v\ (y\,.\,c)$ use the keyword **op** accompanied by label ℓ, parameter value v, and continuation $(y\,.\,c)$. This continuation specifies a computation c that, given the result of the effect bound to y, contains the remainder of the program. For example, c_{ND}, $c_{c,g}$, c_{inc} are variations of an algebraic operation with

label choose. Notice the double usage of the **op** keyword. As a handler clause, **op** signifies the *handling* of an effect with label ℓ, while as an operation, **op** signifies the *invocation* of an effect with label ℓ. Computations are handled by value v using the $\star$-operator ($v \star c$), where the value is a handler. For example, $h_{ND} \star c_{ND}$. Computations can be sequenced by a do-statement **do** $x \leftarrow c_1 \,;\, c_2$, where the result of c_1 is available as x in c_2. Finally, we support application $v_1\ v_2$.

3.2 Types

Types are split in a similar way: terms have value types, computations have computation types.

Value types consist of the unit type $()$, pair types (A, B), function types $A \to \underline{C}$ and handler types $\underline{C} \Rightarrow \underline{D}$. In order to allow meaningful examples, we typically add base types to the calculus, such as String, Int and Bool. Functions take a value of type A as argument and 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 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 [Leijen 2005] in the style of Koka [Leijen 2017]. Therefore, rows E are represented as collections of the previously discussed atomic labels ℓ, optionally terminated by a row variable μ. When a row variable μ is used in the typing of a value or computation, we quantify over it. This implies the presence of type schemes σ in this variant of the Eff-calculus. For example, our nondeterminism handler is typed using a type scheme.

$$h_{ND} : \forall\, \mu\,.\, \mathsf{String}\,!\,\langle \mathtt{choose}\,;\mu\rangle \Rightarrow \mathsf{List}\ \mathsf{String}\,!\,\langle\mu\rangle$$

The calculus includes a designated *signature context* Σ to keep track of the signatures of operations, denoted by their label. It gives information about the typing of operations. For example,

$$\Sigma \;\equiv\; \{\mathtt{choose} : () \twoheadrightarrow \mathsf{Bool}, \mathtt{get} : () \twoheadrightarrow \mathsf{String}, \mathtt{inc} : () \twoheadrightarrow \mathsf{Int}, \mathtt{catch} : \mathsf{String} \twoheadrightarrow \mathsf{Bool}\}$$

Furthermore, a type context Γ keeps track of variables. For the operational semantics and typing rules of Eff, we refer to Pretnar [2015]. However, these rules do not account for the row-typing included in our version presented here. The rules *with* row-polymorphism can be obtained by ignoring the highlighted parts from the semantics (Section 5) and typing rules (Section 6) of λ_{sc}.

In the remaining sections, we extend this version of Eff to include support for scoped effects & handlers, starting with the syntax.

4 SYNTAX

Figure 2 shows the term and type syntax for λ_{sc}. The extensions to Eff are highlighted and can be summarized by two new handler clauses and the inclusion of let-polymorphism in the terms, and type variables and kinds in the types.

Scoped Effect Clauses and Operations. Following the reasoning of Section 2.3, we add a second effect keyword **sc** to model scoped effects. Consequently, **op** now ranges over algebraic effects only. Like their algebraic counterparts, scoped effect *clauses* **sc** $\ell\ x\ p\ k \mapsto c$ feature a label ℓ, parameter x and continuation k. In addition, they contain a scoped computation p. This way, the scope of effect ℓ is delimited: (scoped) computation p is in scope, continuation k is not. Just like **op**, there are two usages of the **sc** keyword: as a clause and as an operation. For scoped *operations* **sc** $\ell\ v\ (y\,.\,c_1)\ (z\,.\,c_2)$ the extension is analogous: they feature an effect label ℓ, a parameter v and a continuation $(z\,.\,c_2)$, like algebraic operations, extended with a scoped computation $(y\,.\,c_1)$.

values v	::=	$() \mid (v_1, v_2) \mid x \mid \boldsymbol{\lambda} x\,.\,c \mid h$	
handlers h	::=	$\mathbf{handler}\ \{\mathbf{return}\ x \mapsto c_r$	return clause
		$\quad, oprs$	effect clauses
		$\quad, \mathbf{fwd}\ f\ p\ k \mapsto c_f\ \}$	forwarding clause
operation clauses $oprs$	::=	$\cdot$	
	$\mid$	$\mathbf{op}\ \ell\ x\ k \mapsto c, oprs$	algebraic effect clauses
	$\mid$	$\mathbf{sc}\ \ell\ x\ p\ k \mapsto c\ , oprs$	scoped effect clauses
computations c	::=	$\mathbf{return}\ v$	return value
	$\mid$	$\mathbf{op}\ \ell\ v\ (y\,.\,c)$	algebraic operation
	$\mid$	$\mathbf{sc}\ \ell\ v\ (y\,.\,c_1)\ (z\,.\,c_2)$	scoped operation
	$\mid$	$v \star c$	handle
	$\mid$	$\mathbf{do}\ x \leftarrow c_1\,;\,c_2$	do-statement
	$\mid$	$v_1\ v_2$	application
	$\mid$	$\mathbf{let}\ x = v\ \mathbf{in}\ c$	let
value types A, B, M	::=	$() \mid (A, B) \mid A \longrightarrow \underline{C} \mid \underline{C} \Rightarrow \underline{D}$	
	$\mid$	α	type variable
	$\mid$	$\lambda\,\alpha\,.\,A$	type operator abstraction
	$\mid$	$M\ A$	type application
type schemes σ	::=	$A \mid \forall\,\mu\,.\,\sigma \mid \forall\,\alpha\,.\,\sigma$	
computation types $\underline{C}, \underline{D}$	::=	$A\,!\,\langle E \rangle$	
effect rows E, F	::=	$\cdot \mid \mu \mid \ell\,;E$	
kinds K	::=	$* \mid K \longrightarrow K$	
signature contexts Σ	::=	$\cdot \mid \Sigma, \ell : A \twoheadrightarrow B$	
type contexts Γ	::=	$\cdot \mid \Gamma, x : A \mid \Gamma, \mu \mid \Gamma, \alpha$	

Fig. 2. Syntax λ_{sc}.

Explicit Forwarding. In Section 2.4 we have motivated that the presence of scoped effects requires all handlers to feature an explicit forwarding clause of shape $\mathbf{fwd}\ f\ p\ k \mapsto c_f$. Computations c_f have access to the scoped computation p and continuation k of the unknown effect they are forwarding. Furthermore, c_f must be able to re-invoke the unknown effect $\mathbf{sc}\ \ell\ v$ so that other handlers can handle it, which is implemented by a function f.

Polymorphic handlers. To allow freedom in the scoped result type, handlers in λ_{sc} abstract over the value type of computations (Section 2.4). This way, handlers can be reapplied to subcomputations of varying value types by polymorphic recursion. To support this, we incorporate let-polymorphism and F_ω-style type operators. Consequently, value types are extended with type variables α, and type schemes are extended with quantification over them: $\forall\,\alpha\,.\,\sigma$. To support type

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

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

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

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

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

$$\frac{c \rightsquigarrow c'}{h \star c \rightsquigarrow h \star c'}\ \text{E-Hand} \qquad \frac{(\textbf{return}\ x \mapsto c_r)\ \in\ h}{h \star \textbf{return}\ v \rightsquigarrow c_r\ [\,v\,/\,x\,]}\ \text{E-HandRet}$$

$$\frac{(\textbf{op}\ \ell\ x\ k \mapsto c)\ \in\ h}{h \star \textbf{op}\ \ell\ v\ (y\,.\,c_1) \rightsquigarrow c\ [\,v\,/\,x, (\lambda y\,.\,h \star c_1)\,/\,k\,]}\ \text{E-HandOp}$$

$$\frac{(\textbf{op}\ \ell\ \_\ \_) \notin h}{h \star \textbf{op}\ \ell\ v\ (y\,.\,c_1) \rightsquigarrow \textbf{op}\ \ell\ v\ (y\,.\,h \star c_1)}\ \text{E-FwdOp}$$

$$\frac{(\textbf{sc}\ \ell\ x\ p\ k \mapsto c)\ \in\ h}{h \star \textbf{sc}\ \ell\ v\ (y\,.\,c_1)\ (z\,.\,c_2) \rightsquigarrow c\ [\,v\,/\,x, (\lambda y\,.\,h \star c_1)\,/\,p, (\lambda z\,.\,h \star c_2)\,/\,k\,]}\ \text{E-HandSc}$$

$$\frac{\begin{array}{c}(\textbf{sc}\ \ell\ \_\ \_\ \_) \notin h \\ (\textbf{fwd}\ f\ p\ k \mapsto c_f)\ \in\ h \qquad g = \lambda(p', k')\,.\,\textbf{sc}\ \ell\ v\ (y\,.\,p'\ y)\ (z\,.\,k'\ z)\end{array}}{h \star \textbf{sc}\ \ell\ v\ (y\,.\,c_1)\ (z\,.\,c_2) \rightsquigarrow c_f\ [\,(\lambda y\,.\,h \star c_1)\,/\,p, (\lambda z\,.\,h \star c_2)\,/\,k, g\,/\,f\,]}\ \text{E-FwdSc}$$

Fig. 3. Operational semantics of λ_{sc}.

operators, we add type abstraction $\lambda\ \alpha\,.\,A$, type application $A\ B$, and kinds K. Kinds are used for arity only, so we need a base kind $*$ for types, and kind arrows $K \longrightarrow K$ for type operators.

5 OPERATIONAL SEMANTICS

Figure 3 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 non-highlighted base rules are the operational semantics of Eff.. 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 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 $\mathbf{do}\ x \leftarrow c_1\,;\,c_2$, 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 2.3) is used to rewrite the computation (E-DoSc).

Handling. For handling computations with a handler of the form $h \star c$, 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 $\mathbf{return}\ x \mapsto c_r$, switching evaluation to c_r with x replaced by v (E-DoRet).

If computation c is an algebraic operation $\mathbf{op}\ \ell\ v\ (y\,.\,c_1)$, 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 continuation. For example, $h_{ND} \star c_{ND}$ (p. 3) reduces as follows.

$$
\begin{aligned}
&h_{ND} \star c_{ND}\\
&\rightsquigarrow\ \mathbf{do}\ xs \leftarrow (\lambda b\,.\,\mathbf{if}\ b\ \mathbf{then}\ h_{ND} \star \mathbf{return}\ \texttt{"heads"}\ \mathbf{else}\ h_{ND} \star \mathbf{return}\ \texttt{"tails"})\ \mathsf{true}\ ;\\
&\quad\ \mathbf{do}\ ys \leftarrow (\lambda b\,.\,\mathbf{if}\ b\ \mathbf{then}\ h_{ND} \star \mathbf{return}\ \texttt{"heads"}\ \mathbf{else}\ h_{ND} \star \mathbf{return}\ \texttt{"tails"})\ \mathsf{false}\,;\\
&\qquad xs + ys\\
&\rightsquigarrow^*\ \mathbf{return}\ [\texttt{"heads"}, \texttt{"tails"}]
\end{aligned}
$$

If h does not contain a clause for label ℓ, 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, $run_{inc}\ 0\ c_{inc}$ (p. 5) forwards the algebraic operation `choose`, which remains in the resulting computation.

$$
\begin{aligned}
&run_{inc}\ 0\ c_{inc}\\
&\rightsquigarrow\ \mathbf{do}\ p' \leftarrow \mathbf{op}\ \texttt{choose}\ ()\ (b\,.\,\mathbf{if}\ b\ \mathbf{then}\ h_{inc} \star \mathbf{op}\ \texttt{inc}\ ()\ (x\,.\,\mathbf{return}\ x)\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{else}\ \ h_{inc} \star \mathbf{op}\ \texttt{inc}\ ()\ (y\,.\,\mathbf{return}\ y))\,;p'\ 0\\
&\rightsquigarrow^*\ \mathbf{op}\ \texttt{choose}\ ()\ (b\,.\,\mathbf{if}\ b\ \mathbf{then}\ (0,1)\ \mathbf{else}\ (0,1)
\end{aligned}
$$

If computation c is a scoped operation $\mathbf{sc}\ \ell\ v\ (y\,.\,c_1)\ (z\,.\,c_2)$, we again distinguish two situations: the case where h contains a clause for ℓ, and where it does not.

If h contains a clause for label ℓ, 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 $h_{once} \star c_{once}$ (p. 7).

$$
\begin{aligned}
&h_{once} \star c_{once}\\
&\rightsquigarrow\ \mathbf{do}\ ts \leftarrow (\lambda y\,.\,h_{once} \star \mathbf{op}\ \texttt{choose}\ ()\ (x\,.\,\mathbf{return}\ x))\ ()\,;\\
&\quad\ \mathbf{do}\ t\ \ \leftarrow \mathsf{head}\ ts\,;(\lambda b\,.\,\mathbf{if}\ b\ \mathbf{then}\ h_{once} \star \mathbf{return}\ \texttt{"heads"}\ \mathbf{else}\ h_{once} \star \mathbf{return}\ \texttt{"tails"})\ t\\
&\rightsquigarrow^*\ \mathbf{return}\ [\texttt{"heads"}]
\end{aligned}
$$

When h does not contain a clause for label ℓ, we must once again forward the effect. As we argued in Section 2.4, forwarding scoped effects cannot happen generically, but rather proceeds via the handler's forwarding clause $\mathbf{fwd}\ f\ p\ k \mapsto c_f$. From there, evaluation switches to computation c_f,

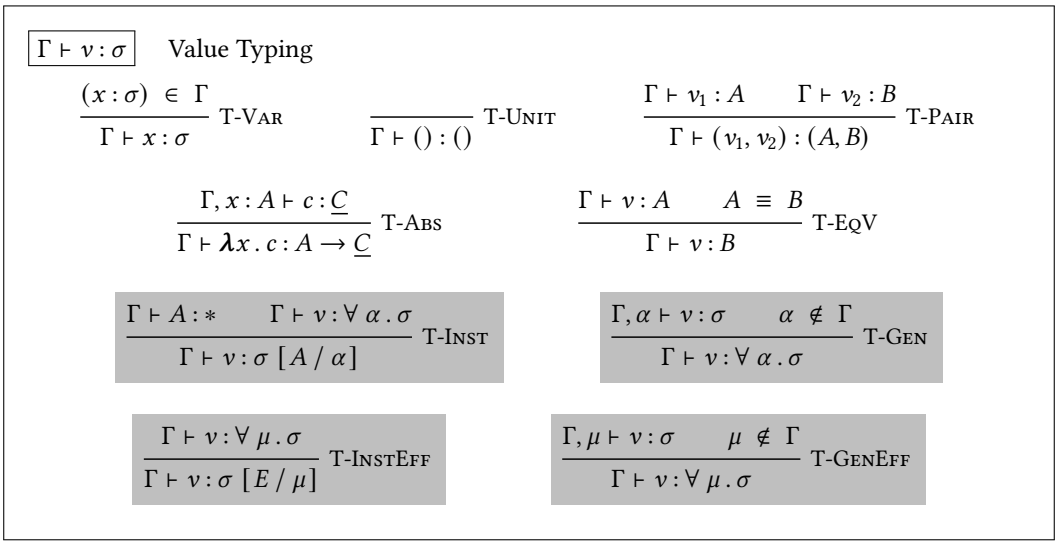

Fig. 4. Value typing.

in which the usages of scoped computation p and continuation k are replaced by their already-handled equivalents. Furthermore, we replace f by a function that takes (p', k') to re-invoke the unknown scoped operation $\mathbf{sc}\ \ell\ v\ p'\ k'$ so that it can be handled by other handlers. The idea is that c_f transforms the original scoped computation p and continuation k to p' and k' to be provided to f as arguments.

For example, consider h_{inc}, which transforms unknown scoped operations so that they can pass around the (incremented) counter s.

$$h_{inc} = \mathbf{handler}\ \{\ldots, \mathbf{fwd}\ f\ p\ k \mapsto \mathbf{return}\ (\boldsymbol{\lambda} s\,.\, f\ (\boldsymbol{\lambda} y\,.\, p\ y\ s, \boldsymbol{\lambda}(z, s')\,.\, k\ z\ s'))\}$$

For a computation, consider the following extension of c_{inc} which uses `choose` and `inc` with `once`:

$$c_{fwd} = \mathbf{sc}\ \texttt{once}\ ()\ (\_\,.\,c_{inc})\ (x\,.\,\mathbf{op}\ \texttt{inc}\ ()\ (y\,.\,\mathbf{return}\ (x + y)))$$

As h_{inc} does not handle `once`, applying it to c_{fwd} means its forwarding clause is used:

$$\begin{aligned}
&h_{once} \star (run_{inc}\ 0\ c_{fwd})\\
&\rightsquigarrow\ h_{once} \star (\mathbf{do}\ c \leftarrow \mathbf{return}\ (\boldsymbol{\lambda} s\,.\,(\boldsymbol{\lambda}(p', k')\,.\,\mathbf{sc}\ \texttt{once}\ ()\ (y\,.\,p'\ y)\ (z\,.\,k'\ z))\\
&\qquad\qquad (\boldsymbol{\lambda} y\,.\quad (\boldsymbol{\lambda}_\,.\,h_{inc} \star c_{inc}) \qquad\qquad\qquad\qquad y\ s,\\
&\qquad\qquad \boldsymbol{\lambda}(z, s')\,.\,(\boldsymbol{\lambda} x\,.\,h_{inc} \star \mathbf{op}\ \texttt{inc}\ ()\ (y\,.\,\mathbf{return}\ (x + y)))\ z\ s'))\,;\\
&\qquad\qquad c\ 0)\\
&\rightsquigarrow^{*}\ \mathbf{return}\ [\,(1, 2)\,]
\end{aligned}$$

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. We discuss the main differences with Eff, which are highlighted in the figures. The kinding rules for types can be found in Appendix C.

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

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

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

$$\frac{\Gamma \vdash v : A}{\Gamma \vdash \mathbf{return}\ v : A!\langle E\rangle}\ \text{T-Ret} \qquad \frac{\Gamma \vdash v : \forall\, \alpha\,.\, \alpha\,!\langle E\rangle \Rightarrow M\ \alpha\,!\langle F\rangle \qquad \Gamma \vdash c : A!\langle E\rangle}{\Gamma \vdash v \star c : M\ A!\langle F\rangle}\ \text{T-Hand}$$

$$\frac{(\ell : A_\ell \twoheadrightarrow B_\ell) \in \Sigma \qquad \Gamma \vdash v : A_\ell \qquad \Gamma, y : B_\ell \vdash c : A!\langle \ell\,; E\rangle}{\Gamma \vdash \mathbf{op}\ \ell\ v\ (y\,.\,c) : A!\langle \ell\,; E\rangle}\ \text{T-Op}$$

$$\frac{(\ell : A_\ell \twoheadrightarrow B_\ell) \in \Sigma \qquad \Gamma \vdash v : A_\ell \qquad \Gamma, y : B_\ell \vdash c_1 : B!\langle \ell\,; E\rangle \qquad \Gamma, z : B \vdash c_2 : A!\langle \ell\,; E\rangle}{\Gamma \vdash \mathbf{sc}\ \ell\ v\ (y\,.\,c_1)\ (z\,.\,c_2) : A!\langle \ell\,; E\rangle}\ \text{T-Sc}$$

Fig. 5. Computation typing.

6.1 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 of the following two rules.

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

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 variables μ with any effect row E, and T-GenEff abstracts over a row variable.

6.2 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. Its implementation is standard.

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 6.3. A handler of type $\forall\ \alpha\ .\ \alpha\,!\,\langle E\rangle \Rightarrow M\ \alpha\,!\,\langle F\rangle$ denotes a polymorphic handler that transforms computations of type $\alpha\,!\,\langle E\rangle$ to a computation of type $M\ \alpha\,!\,\langle F\rangle$. It applies to any computation of value type A with the same effect row E, instantiating α with A for the result type.

Rule T-Op types algebraic effects. Looking up label ℓ in Σ yields a signature $A_\ell \twoheadrightarrow B_\ell$, where A_ℓ is the type of the operation's parameter v, and B_ℓ is the type of argument y of the continuation. The resulting effect row includes ℓ as well as a free effect row E. Finally, the operation's type equals that of continuation c.

Similarly, rule T-Sc types scoped effects. Again, looking up label ℓ in Σ yields signature $A_\ell \twoheadrightarrow B_\ell$ where A_ℓ corresponds to the type of the operation's parameter v. However, where B_ℓ in the algebraic case refers to the *continuation's* argument, B_ℓ now describes the *scoped computation's* argument. This leaves the the scoped result type undescribed by the signature, but as discussed in Section 2.4, this freedom is exactly what we want. As for the effect rows, T-Sc requires the rows of the scoped computation to match.

6.3 Handler typing

The typing rules for handlers and handler clauses are shown in Figure 6. It consists of three judgments. Judgment $\Gamma \vdash \mathit{oprs} : M\ A\,!\,\langle E\rangle$ types operation clauses, and $\Gamma \vdash \mathbf{fwd}\ f\ p\ k \mapsto c : M\ A\,!\,\langle E\rangle$ types forwarding clauses. Finally, $\Gamma \vdash h : \forall\ a\ .\ \alpha\,!\,\langle F\rangle \Rightarrow M\ \alpha\,!\,\langle E\rangle$ types handlers, using the first two judgments.

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 $\mathbf{op}\ \ell\ x\ k \mapsto c$. Looking up label ℓ in Σ yields signature $A_\ell \twoheadrightarrow B_\ell$, where A_ℓ describes the type of parameter x, and B_ℓ 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 $M\ A\,!\,\langle E\rangle$.

Once again, the case for typing a scoped clause $\mathbf{sc}\ \ell\ x\ p\ k \mapsto c$ (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 $\mathbf{sc}\ \ell\ x\ p\ k \mapsto c$, 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:

$$p : B_\ell \to M\ \beta\,!\,\langle E\rangle \qquad\qquad k : \beta \to M\ A\,!\,\langle E\rangle$$

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. Rule T-Fwd types forwarding clauses of the form $\mathbf{fwd}\ f\ p\ k \mapsto c_f$. As the forwarding clause needs to be able to forward any scoped effect in E, it cannot make any assumptions about the specific operation $\ell : A_\ell \twoheadrightarrow B_\ell$ to expect. Instead, it makes abstraction of the operation and treats all possibilities uniformly. This abstraction comes in two parts. Firstly, the function f abstracts over the possible scoped operation calls $\mathbf{sc}\ \ell\ v$ (i.e., where ℓ is any possible label and v a possible parameter of ℓ of associated type A_ℓ). Secondly, the type variable α abstracts

$$\boxed{\Gamma \vdash \mathit{oprs} : M\ A\,!\,\langle E\rangle} \quad \boxed{\Gamma \vdash \mathbf{fwd}\ f\ p\ k \mapsto c : M\ A\,!\,\langle E\rangle} \quad \boxed{\Gamma \vdash h : \forall\ \alpha\ .\ \alpha\,!\,\langle E\rangle \Rightarrow M\ \alpha\,!\,\langle F\rangle}$$

$$\frac{}{\Gamma \vdash \cdot : M\ A\,!\,\langle E\rangle}\ \textsc{T-Empty}$$

$$\frac{(\ell : A_\ell \twoheadrightarrow B_\ell)\ \in\ \Sigma \qquad \begin{array}{c}\Gamma \vdash \mathit{oprs} : M\ A\,!\,\langle E\rangle \\ \Gamma, x : A_\ell, k : B_\ell \to M\ A\,!\,\langle E\rangle \vdash c : M\ A\,!\,\langle E\rangle\end{array}}{\Gamma \vdash \mathbf{op}\ \ell\ x\ k \mapsto c, \mathit{oprs} : M\ A\,!\,\langle E\rangle}\ \textsc{T-OprOp}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathit{oprs} : M\ A\,!\,\langle E\rangle \qquad (\ell : A_\ell \twoheadrightarrow B_\ell)\ \in\ \Sigma \\ \Gamma, \beta, x : A_\ell, p : B_\ell \to M\ \beta\,!\,\langle E\rangle, k : \beta \to M\ A\,!\,\langle E\rangle \vdash c : M\ A\,!\,\langle E\rangle\end{array}}{\Gamma \vdash \mathbf{sc}\ \ell\ x\ p\ k \mapsto c, \mathit{oprs} : M\ A\,!\,\langle E\rangle}\ \textsc{T-OprSc}$$

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

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

Fig. 6. Handler typing.

over the possible argument types B_ℓ of the scoped computation, and type variable β over the scoped result type.

$$p : \alpha \to M\ \beta\,!\,\langle E\rangle \qquad\qquad k : \beta \to M\ A\,!\,\langle E\rangle$$

Notice the difference between the type of the continuation k and the continuation in the argument of function f: it expects as argument (p', k') a transformed version of the scoped computation and continuation so that they agree on the type $M\ \beta$:

$$p' : \alpha \to M\ \beta\,!\,\langle E\rangle \qquad\qquad k' : M\ \beta \to M\ A\,!\,\langle E\rangle$$

Transforming the original continuation to agree on this type is the exact purpose of our explicit forwarding clauses.

Handler. Rule T-Handler types handlers with a polymorphic type of the form $\forall\ \alpha\ .\ \alpha\,!\,\langle E\rangle \Rightarrow M\ \alpha\,!\,\langle F\rangle$. A handler consists of a **return**-clause, 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 $M\ \alpha\,!\,\langle F\rangle$. 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).

6.4 Metatheory

The type-and-effect system of λ_{sc} is type safe. In this section we briefly state the theorems to show this; the proofs and used lemmas can be found in Appendix D. We prove type safety by proving

Subject Reduction and Progress. As values are inert , these theorems range over computations only. The formulation of Subject Reduction is standard:

Theorem 6.1 (Subject Reduction). *If* $\Gamma \vdash c : \underline{C}$ *and* $c \rightsquigarrow c'$, *then* $\Gamma \vdash c' : \underline{C}$.

Apart from an additional normal form $\mathbf{sc}\ \ell\ v\ (y\,.\,c_1)\ (z\,.\,c_2)$, progress is standard as well:

Theorem 6.2 (Progress). *If* $\cdot \vdash c : \underline{C}$, *then either:*

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

7 EXAMPLES

This section demonstrates the expressiveness of λ_{sc} with a range of scoped effect examples. 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. Also, we drop trivial **return** continuations of operations:

$$\mathbf{op}\ \ell\ x\ \equiv\ \mathbf{op}\ \ell\ x\ (y\,.\,\mathbf{return}\ y) \qquad\qquad \mathbf{sc}\ \ell\ x\ (y\,.\,c_1)\ \equiv\ \mathbf{sc}\ \ell\ x\ (y\,.\,c_1)\ (z\,.\,\mathbf{return}\ z)$$

Moreover, we use the following syntactic sugar for types to simplify the code:

$$A \longrightarrow^{E} B\ \equiv\ A \to B\,!\,E \qquad\qquad A\,!\,\langle\mu\rangle \equiv\ A\,!\,\mu$$

Another useful syntactic sugar is **lift**, which captures a recurring pattern of the forwarding clause.

$$\mathbf{lift}\ z\ k \mapsto c_f\ \equiv\ \mathbf{fwd}\ f\ p\ k \mapsto f\ (p, \boldsymbol{\lambda} z\,.\,c_f)\ \mathbf{where}\ f, p\ \notin\ FV\ (c_f)$$

Intuitively, it means reconciling the result $z : M\ \alpha$ of the scoped computation with the continuation $k : \alpha \longrightarrow^{\mu} M\ \beta$. We can observe this simplified pattern often, but sometimes we still require the more general forwarding clause such as the handlers for local state (Section 7.4) and depth-bounded search (Section 7.5). For the sake of space the examples are abridged; we refer to the Supplementary Material for the full code and an interpreter for λ_{sc} implemented in Haskell.

7.1 Nondeterminism with Once

Before we move on to new examples, we complete the handler for nondeterminism with `once` from Section 2.2 to obtain the full example given by Piróg et al. [2018a]. This means adding an operation `fail` : () → Empty that is used to model failure. The full handler below combines all previously defined clauses as well as a clause for `fail`, mapping it to the empty list. Note how the forwarding clause of Section 2.4 can be expressed in terms of **lift**.

$$\begin{aligned}
&h_{once}\ :\ \forall\ \alpha\ \mu\,.\,\alpha\,!\,\langle \texttt{choose}; \texttt{fail}; \texttt{once}; \mu\rangle \Rightarrow \mathsf{List}\ \alpha\,!\,\mu\\
&h_{once} = \mathbf{handler}\ \{\mathbf{return}\ x \qquad\quad \mapsto [x]\\
&\qquad\qquad\quad , \mathbf{op}\ \texttt{fail}\ _\ _ \quad \mapsto [\,]\\
&\qquad\qquad\quad , \mathbf{op}\ \texttt{choose}\ x\ k \mapsto k\ \mathsf{true} +\!\!+ k\ \mathsf{false}\\
&\qquad\qquad\quad , \mathbf{sc}\ \texttt{once}\ _\ p\ k \ \mapsto k\ (\mathsf{head}\ (p\ ()))\\
&\qquad\qquad\quad , \mathbf{lift}\ z\ k \qquad\quad \mapsto \mathsf{concatMap}\ z\ k\}
\end{aligned}$$

For the forwarding clause, we use concatMap, which maps k over the list z and flattens the resulting nested list. We refer to Appendix E.1 for the full code.

$$\mathsf{concatMap} : \forall\ \alpha\ \beta\ \mu\,.\,\mathsf{List}\ \beta \longrightarrow^{\mu} (\beta \longrightarrow^{\mu} \mathsf{List}\ \alpha) \longrightarrow^{\mu} \mathsf{List}\ \alpha$$

Typically, it is useful to have a wrapper for `fail` with a polymorphic return type rather than Empty. The function absurd turns the type Empty to any type.

$$\mathsf{failure} : \forall\ \alpha\ \mu\ .\ () \to^{\langle \mathtt{fail};\mu \rangle} \alpha$$
$$\mathsf{failure}\ \_ = \mathbf{op}\ \mathtt{fail}\ ()\ (r\ .\ \mathsf{absurd}\ r)$$

7.2 Nondeterminism with Cut

The algebraic operation `cut` : () → () provides a different flavor of pruning nondeterminism. 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 `call` : () → (), as proposed by Wu et al. [2014]. To handle `cut` and `call`, we use the approach of Piróg and Staton [2017], which is based on the CutList datatype.

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

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

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

The handler for nondeterminism with cut is defined as follows:

$$\begin{aligned}
&h_{cut}\ :\ \forall\ \alpha\ \mu\ .\ \alpha\ !\ \langle \mathtt{choose};\mathtt{fail};\mathtt{cut};\mathtt{call};\mu \rangle \Rightarrow \mathsf{CutList}\ \alpha\ !\ \mu \\
&h_{cut} = \mathbf{handler}\ \{\mathbf{return}\ x \qquad\quad \mapsto \mathsf{opened}\ [x] \\
&\qquad\qquad , \mathbf{op}\ \mathtt{fail}\ _\ _ \quad \mapsto \mathsf{opened}\ [\,] \\
&\qquad\qquad , \mathbf{op}\ \mathtt{choose}\ x\ k \mapsto \mathsf{append}\ (k\ \mathsf{true})\ (k\ \mathsf{false}) \\
&\qquad\qquad , \mathbf{op}\ \mathtt{cut}\ _\ k \quad \mapsto \mathsf{close}\ (k\ ()) \\
&\qquad\qquad , \mathbf{sc}\ \mathtt{call}\ _\ p\ k \ \mapsto \mathsf{concatMap}_{\mathsf{CutList}}\ (\mathsf{open}\ (p\ ()))\ k \\
&\qquad\qquad , \mathbf{lift}\ z\ k \qquad \mapsto \mathsf{concatMap}_{\mathsf{CutList}}\ z\ k\}
\end{aligned}$$

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

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

The function $\mathsf{concatMap}_{\mathsf{CutList}}$ is the cutlist counterpart of concatMap which takes the extensibility of CutList (signalled by opened and closed) into account when concatenating. We refer to Appendix E.2 for the full code.

$$\mathsf{concatMap}_{\mathsf{CutList}} : \forall\ \alpha\ \beta\ \mu\ .\ \mathsf{CutList}\ \beta \to^{\mu} (\beta \to^{\mu} \mathsf{CutList}\ \alpha) \to^{\mu} \mathsf{CutList}\ \alpha$$

In Section 7.6, we give an example usage of `cut` to improve parsers.

7.3 Exceptions

Wu et al. [2014] have shown how to catch exceptions with a scoped operation. Raising an exception is an algebraic operation `raise`:String → Empty, and catching an exception is a scoped operation `catch` : String → Bool. For example, consider the following program which raises an error when the state exceeds 10:

$$c_{raise} = \mathbf{do}\ x \leftarrow \mathbf{op}\ \mathtt{inc}\ ()\ ;\mathbf{if}\ x > 10\ \mathbf{then}\ \mathbf{op}\ \mathtt{raise}\ \texttt{"Overflow"}\ (y\ .\ \mathsf{absurd}\ y)\ \mathbf{else}\ \mathbf{return}\ x$$

We can use the scoped operation `catch` to catch the error and return 10:

$$c_{catch} = \textbf{sc}\ \texttt{catch}\ \texttt{"Overflow"}\ (b\,.\,\textbf{if}\ b\ \textbf{then}\ c_{raise}\ \textbf{else return}\ 10)$$

The scoped computation's true branch is the program that may *raise* an error, while the false branch *deals with* the `"Overflow"` error. Our handler interprets exceptions in terms of a sum type:

$$\textbf{data}\ \alpha + \beta = \mathsf{left}\ \alpha\ \mid\ \mathsf{right}\ \beta$$

Here, $\mathsf{left}\ v$ denotes an error and $\mathsf{right}\ v$ a result.

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

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 true raises an exception e, it is caught by `catch` and replaced by the scoped computation (p false). Otherwise, it continues with p true and its results are passed to the continuation k. The forwarding clause is defined in terms of **lift** $z\ k$. Essentially, we return the exception if z fails ($\mathsf{left}\ e$), and we apply the continuation k to z if z succeeds ($\mathsf{right}\ x$).

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

Now we can handle the program c_{catch} with h_{except} and h_{inc} (p. 15). When an error is raised, the `inc` operation does not influence the result and the final value of the counter remains 42.

$$\begin{aligned}
&>\!>\!> h_{except} \star ((h_{inc} \star c_{catch})\ 42)\\
&\mathsf{right}\ (10, 42)
\end{aligned}$$

Also, we can swap the order of the two handlers:

$$\begin{aligned}
&>\!>\!> (h_{inc} \star (h_{except} \star c_{catch}))\ 42\\
&(\mathsf{right}\ 10, 43)
\end{aligned}$$

The final value of the counter becomes 43 because the counter is global now. The `inc` operation does have effect regardless of whether any error is raised or not.

7.4 Local State

Scoped effects give us the ability to augment the traditional state operations $\texttt{get} : \mathsf{String} \rightarrow \mathsf{Int}$ and $\texttt{put} : (\mathsf{String}, \mathsf{Int}) \rightarrow \mathsf{Int}$ with a new operation `local` for local variables [Piróg et al. 2018a].[4]

$$\texttt{local} : (\mathsf{String}, \mathsf{Int}) \rightarrow ()$$

[4]We assume the type of the state to be Int as λ_{sc} does not support parameterized effects or polymorphic effects.

The scoped operation **sc** `local` (x, v) $(y\,.\,c_1)$ $(z\,.\,c_2)$ creates a local variable with name x and value v, which can only be reached in the scoped computation $y\,.\,c_1$. We define the return type of the handler as Mem String Int $\to^{\mu}$ $(\alpha,$ Mem String Int$)$, where Mem α β is a map from keys of type α to values of type β. It has three auxiliary functions:

$$\begin{aligned}
&\mathsf{newmem} : \forall\ \alpha\ \beta\ \mu\,.\,() \to^{\mu} \mathsf{Mem}\ \alpha\ \beta\\
&\mathsf{retrieve}\ \ : \forall\ \alpha\ \beta\ \mu\,.\,\alpha \to^{\mu} \mathsf{Mem}\ \alpha\ \beta \to^{\mu} \beta\\
&\mathsf{update}\ \ : \forall\ \alpha\ \beta\ \mu\,.\,(\alpha,\beta) \to^{\mu} \mathsf{Mem}\ \alpha\ \beta \to^{\mu} \mathsf{Mem}\ \alpha\ \beta
\end{aligned}$$

The function newmem () creates an empty map, retrieve a m gets the value of the key a from the memory m, and update (a, b) m updates the value of a in m to b. The handler is given as follows.

$$\begin{aligned}
&h_{state} : \forall\ \alpha\ \mu\,.\,\alpha\ !\ \langle \texttt{get}\,;\texttt{put}\,;\texttt{local}\,;\mu\rangle \Rightarrow (\mathsf{Mem\ String\ Int} \to^{\mu} (\alpha, \mathsf{Mem\ String\ Int}))\ !\ \mu\\
&h_{state} = \mathbf{handler}\ \{\mathbf{return}\ x \qquad\qquad \mapsto \boldsymbol{\lambda} m\,.\,(x, m)\\
&\qquad\qquad , \mathbf{op}\ \texttt{get}\ x\ k \qquad \mapsto \boldsymbol{\lambda} m\,.\,k\ (\mathsf{retrieve}\ x\ m)\ m\\
&\qquad\qquad , \mathbf{op}\ \texttt{put}\ pa\ k \qquad \mapsto \boldsymbol{\lambda} m\,.\,k\ ()\ (\mathsf{update}\ pa\ m)\\
&\qquad\qquad , \mathbf{sc}\ \texttt{local}\ (x, v)\ p\ k \mapsto \boldsymbol{\lambda} m\,.\,\mathbf{do}\ (t, m') \leftarrow p\ ()\ (\mathsf{update}\ (x, v)\ m)\,;\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad k\ t\ (\mathsf{update}\ (x, \mathsf{retrieve}\ x\ m)\ m')\\
&\qquad\qquad , \mathbf{fwd}\ f\ p\ k \qquad\qquad \mapsto \boldsymbol{\lambda} m\,.\,f\ (\boldsymbol{\lambda} y\,.\,p\ y\ m, \boldsymbol{\lambda}(z, m')\,.\,k\ z\ m')\}
\end{aligned}$$

Note that the forwarding clause of h_{state} is the same as the forwarding clause of h_{inc} in Section 5. For example, consider the following program:

$$\begin{aligned}
&c_{state} = \mathbf{do}\ \mathbf{op}\ \texttt{put}\ (\texttt{"x"}, 10)\,;\\
&\qquad\quad \mathbf{do}\ x_1 \leftarrow \mathbf{sc}\ \texttt{local}\ (\texttt{"x"}, 42)\ (_\,.\,\mathbf{op}\ \texttt{get}\ \texttt{"x"})\,;\\
&\qquad\quad \mathbf{do}\ x_2 \leftarrow \mathbf{op}\ \texttt{get}\ \texttt{"x"}\,;\\
&\qquad\qquad \mathbf{return}\ (x_1, x_2)
\end{aligned}$$

The local variable "x" does not influence the value of the global variable "x".

```
>>> fst ((h_state ⋆ c_state) (newmem ()))
(42, 10)
```

7.5 Depth-Bounded Search

The handlers for nondeterminism shown so far implement the *depth-first search* (DFS) strategy. However, with scoped effects and handlers we can implement other search strategies, such as *depth-bounded search* (DBS) [Yang et al. 2022], which uses the scoped operation `depth` : Int $\to$ () to bound the depth of the branches in the scoped computation. The handler uses return type Int $\to^{\mu}$ List $(\alpha,$ Int$)$. Here, the Int parameter is the current depth bound, and the result is a list of $(\alpha,$ Int$)$ pairs, where α denotes the result and Int reflects the remaining global depth bound.[5]

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

[5] These pairs $(\alpha,$ Int$)$ differ from Yang et al. [2022]'s α in order to make the forwarding clause work.

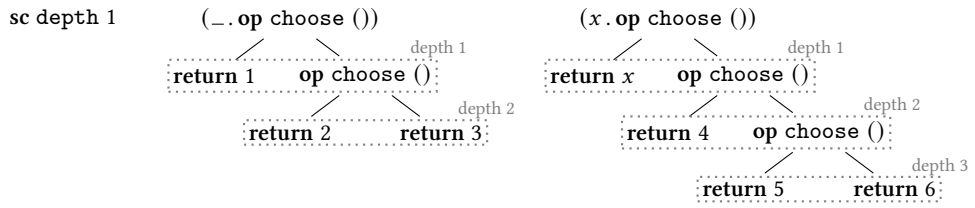

Fig. 7. Visual representation of c_{depth}.

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. For example, the following program (Figure 7) 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.

$$
\begin{aligned}
c_{depth} = \mathbf{sc}\ \mathtt{depth}\ 1\ (\_ .\ & \mathbf{do}\ b \leftarrow \mathbf{op}\ \mathtt{choose}\ ()\,;\mathbf{if}\ b\ \mathbf{then\ return}\ 1\ \mathbf{else} \\
& \quad \mathbf{do}\ b' \leftarrow \mathbf{op}\ \mathtt{choose}\ ()\,;\mathbf{if}\ b'\ \mathbf{then\ return}\ 2\ \mathbf{else\ return}\ 3) \\
(x .\ & \mathbf{do}\ b \leftarrow \mathbf{op}\ \mathtt{choose}\ ()\,;\mathbf{if}\ b\ \mathbf{then\ return}\ x\ \mathbf{else} \\
& \quad \mathbf{do}\ b' \leftarrow \mathbf{op}\ \mathtt{choose}\ ()\,;\mathbf{if}\ b'\ \mathbf{then\ return}\ 4\ \mathbf{else} \\
& \quad\ \ \mathbf{do}\ b'' \leftarrow \mathbf{op}\ \mathtt{choose}\ ()\,;\mathbf{if}\ b''\ \mathbf{then\ return}\ 5\ \mathbf{else\ return}\ 6)
\end{aligned}
$$

$$
\begin{aligned}
&>\!\!>\!\!> (h_{depth} \star c_{depth})\ 2 \\
&[\,(1,1),(4,0)\,]
\end{aligned}
$$

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. For a different implementation, we refer to the Supplementary Material.

7.6 Parsers

A parser effect can be achieved by combining the nondeterminism-with-cut effect and a token-consuming effect [Wu et al. 2014]. The latter features the algebraic operation $\mathtt{token} : \mathsf{Char} \to \mathsf{Char}$ where **op** `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.

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

Figure 8 shows 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 $x \diamond y \ \equiv\ \mathbf{op}\ \mathtt{choose}\ (b\ .\ \mathbf{if}\ b\ \mathbf{then}\ x\ \mathbf{else}\ y)$. The expr parser is naive and can be improved by two types 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 +.

```
digit    : ∀ μ . () → Char ! ⟨token ; choose ; μ⟩
digit _  = op token '0' ⋄ op token '1' ⋄ ... ⋄ op token '9'

many1    : ∀ α μ . (() → α ! μ) → List α ! μ
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
```

Fig. 8. A small expression parser.

```
expr1    : ∀ μ . () → Int ! ⟨token ; choose ; cut ; μ⟩
expr1 _  = do i ← term () ;
               sc call () (_ . (do op token '+' ; op cut () ; j ← expr1 () ; return (i + j)) ⋄ i)
```

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

```
>>> h_cut ★ (h_token ★ expr1 ()) "(2+5)*8"
opened [(56, ""), (7, "*8")]
```

There are two results in the cutlist. Usually we are only interested in the full parsers, i.e., those that have consumed the entire input string.

8 RELATED WORK

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

8.1 Languages and Packages for Algebraic Effects & Handlers

Many research languages for algebraic effects have been proposed, including Eff [Bauer and Pretnar 2013; Pretnar 2015], Frank [Lindley et al. 2017], Effekt [Brachthäuser et al. 2020], or have been extended to include them, such as Links [Hillerström and Lindley 2016], Koka [Leijen 2017], and Multicore OCaml [Sivaramakrishnan et al. 2021]. There are also many packages for writing effect handlers in general purpose languages [King 2019; Kiselyov et al. 2019; Kiselyov and Sivaramakrishnan 2018; Maguire 2019; Rix et al. 2018]. Yet, as far as we know, λ_{sc} is the first calculus that supports scoped effects & handlers.

8.2 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 [Bauer and Pretnar 2013; Pretnar 2015] 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. We did not choose to use the subtyping-based effect types in λ_{sc} as that would require complex subtyping for type operators.

Row polymorphism is another mainstream approach to effect systems. Links [Hillerström and Lindley 2016] uses the Rémy style row polymorphism [Rémy 1994], where the row types are able

to represent the absence of labels and each label is restricted to appear at most once. Koka [Leijen 2017] uses row polymorphism based on scoped labels [Leijen 2005], 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.

8.3 Scoped Effects & Handlers

Wu et al. [2014] 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 higher-order syntax, an approach to scoped effects & handlers that has already been implemented in several Haskell packages [King 2019; Maguire 2019; Rix et al. 2018]. They use higher-order signatures, which impose less restrictions on the shape of the signatures of scoped operations and allow programmers to delimit the scopes in a freer way than λ_{sc}. The cost of this freedom is the need for programmers to write more functions to distribute handlers for each signature. The higher-order signatures are also not suitable for use in a calculus as the signatures of operations are usually characterised by a pair of types in a calculus.

Piróg et al. [2018b] and Yang et al. [2022] have developed denotational semantic domains of scoped effects, backed by category theorical 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. [2022] 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. Our λ_{sc} calculus uses a similar idea to the 'explicit substitution' monad of Piróg et al. [2018b], a generalization of Ghani and Uustalu [2003]'s monad of explicit substitutions where each operation is associated with two computations representing the computation in scope and out of the scope (continuation) respectively. While the composition of scoped effects has not been considered in their categorical models, we introduced forwarding clauses for the composition, and further restrict handlers to be polymorphic to simplify handling and composing scoped effects.

9 CONCLUSION AND FUTURE WORK

In this work, we have presented λ_{sc}, a novel calculus in which scoped effects & handlers are first-class. We have started from Eff, extended with row-typing in the style of Koka, and added scoped effect clauses and operations, polymorphic handlers, and explicit forwarding clauses. 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 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. We believe that for practical programming implementations our calculus can be easily extended to support both kinds of handlers side by side. Future areas of development include studying the structure of forwarding clauses, possibly automating this process. Furthermore, a richer type system with higher-rank polymorphism would allow us to write more complex scoped effect examples, such as multi-threading [Yang et al. 2022]

or aspect-oriented programming [Figueroa et al. 2014]. Finally, it could be interesting to extend the calculus to also support other classes of effects (e.g., latent effects [van den Berg et al. 2021], asynchronous effects [Ahman and Pretnar 2021], parallel effects [Xie et al. 2021]).

REFERENCES

Danel Ahman and Matija Pretnar. 2021. Asynchronous Effects. *Proc. ACM Program. Lang.* 5, POPL, Article 24 (jan 2021), 28 pages. https://doi.org/10.1145/3434305

Andrej Bauer and Matija Pretnar. 2013. An Effect System for Algebraic Effects and Handlers. In *Algebra and Coalgebra in Computer Science*, Reiko Heckel and Stefan Milius (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–16.

Andrej Bauer and Matija Pretnar. 2015. Programming with algebraic effects and handlers. *Journal of Logical and Algebraic Methods in Programming* 84, 1 (2015), 108–123. https://doi.org/10.1016/j.jlamp.2014.02.001 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.

Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. 2020. Effects as Capabilities: Effect Handlers and Lightweight Effect Polymorphism. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 126 (Nov. 2020), 30 pages. https://doi.org/10.1145/3428194

Ismael Figueroa, Tom Schrijvers, Nicolas Tabareau, and Éric Tanter. 2014. Compositional Reasoning about Aspect Interference. In *Proceedings of the 13th International Conference on Modularity* (Lugano, Switzerland) *(MODULARITY '14)*. Association for Computing Machinery, New York, NY, USA, 133–144. https://doi.org/10.1145/2577080.2577093

Neil Ghani and Tarmo Uustalu. 2003. Explicit Substitutions and Higher-Order Syntax. In *Proceedings of the 2003 ACM SIGPLAN Workshop on Mechanized Reasoning about Languages with Variable Binding* (Uppsala, Sweden) *(MERLIN '03)*. Association for Computing Machinery, New York, NY, USA, 1–7. https://doi.org/10.1145/976571.976580

Andy Gill. 2008. mtl: Monad transformer library. https://hackage.haskell.org/package/mtl-1.1.0.2.

Daniel Hillerström and Sam Lindley. 2016. Liberating Effects with Rows and Handlers. In *Proceedings of the 1st International Workshop on Type-Driven Development* (Nara, Japan) *(TyDe 2016)*. Association for Computing Machinery, New York, NY, USA, 15–27. https://doi.org/10.1145/2976022.2976033

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

Oleg Kiselyov, Amr Sabry, Cameron Swords, and Ben Foppa. 2019. extensible-effects: An Alternative to Monad Transformers. https://hackage.haskell.org/package/extensible-effects.

Oleg Kiselyov and KC Sivaramakrishnan. 2018. Eff Directly in OCaml. *Electronic Proceedings in Theoretical Computer Science* 285 (Dec 2018), 23–58. https://doi.org/10.4204/eptcs.285.2

Daan Leijen. 2005. Extensible records with scoped labels. *Trends in Functional Programming* 6 (2005), 179–194.

Daan Leijen. 2017. Type Directed Compilation of Row-Typed Algebraic Effects. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL 2017)*. Association for Computing Machinery, New York, NY, USA, 486–499. https://doi.org/10.1145/3009837.3009872

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

Sam Lindley, Conor McBride, and Craig McLaughlin. 2017. Do Be Do Be Do. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL 2017)*. Association for Computing Machinery, New York, NY, USA, 500–514. https://doi.org/10.1145/3009837.3009897

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

Eugenio Moggi. 1989. *An Abstract View of Programming Languages.* Technical Report ECS-LFCS-90-113. Edinburgh University, Department of Computer Science.

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

Maciej Piróg, Tom Schrijvers, Nicolas Wu, and Mauro Jaskelioff. 2018a. Syntax and Semantics for Operations with Scopes. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science* (Oxford, United Kingdom) *(LICS '18)*. Association for Computing Machinery, New York, NY, USA, 809–818. https://doi.org/10.1145/3209108.3209166

Maciej Piróg, Tom Schrijvers, Nicolas Wu, and Mauro Jaskelioff. 2018b. Syntax and Semantics for Operations with Scopes. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science* (Oxford, United Kingdom) *(LICS '18)*. Association for Computing Machinery, New York, NY, USA, 809–818. https://doi.org/10.1145/3209108.3209166

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

Gordon Plotkin and Matija Pretnar. 2009. Handlers of Algebraic Effects. In *Programming Languages and Systems*, Giuseppe Castagna (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 80–94. https://doi.org/10.1007/978-3-642-00590-9_7

Gordon D. Plotkin and John Power. 2003. Algebraic Operations and Generic Effects. *Appl. Categorical Struct.* 11, 1 (2003), 69–94. https://doi.org/10.1023/A:1023064908962

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

Didier Rémy. 1994. Type Inference for Records in a Natural Extension of ML. In *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*. Citeseer.

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

KC Sivaramakrishnan, Stephen Dolan, Leo White, Tom Kelly, Sadiq Jaffer, and Anil Madhavapeddy. 2021. Retrofitting Effect Handlers onto OCaml. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 206–221. https://doi.org/10.1145/3453483.3454039

Birthe van den Berg, Tom Schrijvers, Casper Bach Poulsen, and Nicolas Wu. 2021. Latent Effects for Reusable Language Components: Extended Version. *ArXiv* abs/2108.11155 (2021).

Philip Wadler. 1995. Monads for Functional Programming. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*. Springer-Verlag, Berlin, Heidelberg, 24â52. https://doi.org/10.5555/647698.734146

Nicolas Wu, Tom Schrijvers, and Ralf Hinze. 2014. Effect Handlers in Scope. In *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell* (Gothenburg, Sweden) *(Haskell '14)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/2633357.2633358

Ningning Xie, Daniel D. Johnson, Dougal Maclaurin, and Adam Paszke. 2021. Parallel Algebraic Effect Handlers. *CoRR* abs/2110.07493 (2021). arXiv:2110.07493 https://arxiv.org/abs/2110.07493

Zhixuan Yang, Marco Paviotti, Nicolas Wu, Birthe van den Berg, and Tom Schrijvers. 2022. Structured Handling of Scoped Effects. In *ESOP*. Springer.

A SEMANTIC DERIVATIONS

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

A.1 Nondeterminism

$$
\begin{array}{ll}
& h_{ND} \star c_{ND} \\
\rightsquigarrow & \{\text{- E-HandOp -}\} \\
& \textbf{do}\ xs \leftarrow (\lambda b\,.\,\textbf{if}\ b\ \textbf{then}\ h_{ND} \star \textbf{return}\ \texttt{"heads"}\ \textbf{else}\ h_{ND} \star \textbf{return}\ \texttt{"tails"})\ \mathsf{true}\ ; \\
& \textbf{do}\ ys \leftarrow (\lambda b\,.\,\textbf{if}\ b\ \textbf{then}\ h_{ND} \star \textbf{return}\ \texttt{"heads"}\ \textbf{else}\ h_{ND} \star \textbf{return}\ \texttt{"tails"})\ \mathsf{false}\,; \\
& xs \mathbin{+\!\!+} ys \\
\rightsquigarrow & \{\text{- E-Do and E-AppAbs -}\} \\
& \textbf{do}\ xs \leftarrow h_{ND} \star \textbf{return}\ \texttt{"heads"}\,; \\
& \textbf{do}\ ys \leftarrow h_{ND} \star \textbf{return}\ \texttt{"tails"}\,; \\
& xs \mathbin{+\!\!+} ys \\
\rightsquigarrow & \{\text{- E-Do and E-HandRet -}\} \\
& \textbf{do}\ xs \leftarrow \textbf{return}\ [\texttt{"heads"}]\,; \\
& \textbf{do}\ ys \leftarrow \textbf{return}\ [\texttt{"tails"}]\,; \\
& xs \mathbin{+\!\!+} ys \\
\rightsquigarrow & \{\text{- E-DoRet -}\} \\
& [\texttt{"heads"}, \texttt{"tails"}]
\end{array}
$$

A.2 Increment

$$
\begin{array}{ll}
& run_{inc}\ 0\ c_{inc} \ \equiv\ (\lambda c\ p\,.\,\textbf{do}\ p' \leftarrow h_{inc} \star p\,; p'\ c)\ 0\ c_{inc} \\
\rightsquigarrow & \{\text{- E-AppAbs -}\} \\
& \textbf{do}\ p' \leftarrow h_{inc} \star c_{inc}\,; p'\ 0 \\
\rightsquigarrow & \{\text{- E-Do and E-HandFwd -}\} \\
& \textbf{do}\ p' \leftarrow \textbf{op}\ \texttt{choose}\ ()\ (b\,.\,\textbf{if}\ b\ \textbf{then}\ h_{inc} \star \textbf{op}\ \texttt{inc}\ ()\ (x\,.\,\textbf{return}\ x) \\
& \qquad\qquad\qquad\qquad\qquad\qquad \textbf{else}\ h_{inc} \star \textbf{op}\ \texttt{inc}\ ()\ (y\,.\,\textbf{return}\ y))\,; p'\ 0 \\
\rightsquigarrow & \{\text{- E-DoOp -}\} \\
& \textbf{op}\ \texttt{choose}\ ()\ (b\,.\,\textbf{if}\ b\ \textbf{then do}\ p' \leftarrow h_{inc} \star \textbf{op}\ \texttt{inc}\ ()\ (x\,.\,\textbf{return}\ x)\,; p'\ 0 \\
& \qquad\qquad\qquad\qquad\quad \textbf{else}\ \ \textbf{do}\ p' \leftarrow h_{inc} \star \textbf{op}\ \texttt{inc}\ ()\ (y\,.\,\textbf{return}\ y)\,; p'\ 0) \\
\rightsquigarrow & \{\text{- E-Do and E-HandOp -}\} \\
& \textbf{op}\ \texttt{choose}\ ()\ (b\,.\,\textbf{if}\ b\ \textbf{then do}\ p' \leftarrow \textbf{return}\ (\lambda c\,.\,(\lambda x\,.\,h_{inc} \star \textbf{return}\ x)\ c\ (c+1))\,; p'\ 0 \\
& \qquad\qquad\qquad\qquad\quad \textbf{else}\ \ \textbf{do}\ p' \leftarrow \textbf{return}\ (\lambda c\,.\,(\lambda y\,.\,h_{inc} \star \textbf{return}\ y)\ c\ (c+1))\,; p'\ 0) \\
\rightsquigarrow & \{\text{- E-DoRet -}\} \\
& \textbf{op}\ \texttt{choose}\ ()\ (b\,.\,\textbf{if}\ b\ \textbf{then}\ (\lambda c\,.\,(\lambda x\,.\,h_{inc} \star \textbf{return}\ x)\ c\ (c+1))\ 0 \\
& \qquad\qquad\qquad\qquad\quad \textbf{else}\ \ (\lambda c\,.\,(\lambda y\,.\,h_{inc} \star \textbf{return}\ y)\ c\ (c+1))\ 0) \\
\rightsquigarrow & \{\text{- E-AppAbs -}\} \\
& \textbf{op}\ \texttt{choose}\ ()\ (b\,.\,\textbf{if}\ b\ \textbf{then}\ (\lambda x\,.\,h_{inc} \star \textbf{return}\ x)\ 0\ 1 \\
& \qquad\qquad\qquad\qquad\quad \textbf{else}\ \ (\lambda y\,.\,h_{inc} \star \textbf{return}\ y)\ 0\ 1 \\
\rightsquigarrow & \{\text{- E-AppAbs -}\} \\
& \textbf{op}\ \texttt{choose}\ ()\ (b\,.\,\textbf{if}\ b\ \textbf{then}\ (h_{inc} \star \textbf{return}\ 0)\ 1 \\
& \qquad\qquad\qquad\qquad\quad \textbf{else}\ \ (h_{inc} \star \textbf{return}\ 0)\ 1
\end{array}
$$

$\rightsquigarrow$ {- E-HandRet -}
op `choose` () (b . **if** b **then** (**return** (λc . **return** $(0, c)$)) 1
else (**return** (λc . **return** $(0, c)$)) 1
$\rightsquigarrow$ {- E-AppAbs -}
op `choose` () (b . **if** b **then** $(0, 1)$ **else** $(0, 1)$

A.3 Once

$h_{once} \star c_{once}$
$\rightsquigarrow$ {- E-HandSc -}
do $ts \leftarrow (\lambda y$. $h_{once} \star$ **op** `choose` () (x . **return** x)) () ;
do $t \leftarrow$ head ts ;
(λb . **if** b **then** $h_{once} \star$ **return** `"heads"` **else** $h_{once} \star$ **return** `"tails"`) t
$\rightsquigarrow$ {- E-Do and E-AppAbs -}
do $ts \leftarrow h_{once} \star$ **op** `choose` () (x . **return** x) ;
do $t \leftarrow$ head ts ;
(λb . **if** b **then** $h_{once} \star$ **return** `"heads"` **else** $h_{once} \star$ **return** `"tails"`) t
$\rightsquigarrow$ {- E-Do and E-HandOp -}
do $ts \leftarrow$ **do** $xs \leftarrow (\lambda x$. $h_{once} \star$ **return** x) true ; **do** $ys \leftarrow (\lambda x$. $h_{once} \star$ **return** x) false ; xs **+** ys ;
do $t \leftarrow$ head ts ;
(λb . **if** b **then** $h_{once} \star$ **return** `"heads"` **else** $h_{once} \star$ **return** `"tails"`) t
$\rightsquigarrow$ {- E-Do and E-AppAbs -}
do $ts \leftarrow$ **do** $xs \leftarrow h_{once} \star$ **return** true ; **do** $ys \leftarrow h_{once} \star$ **return** false ; xs **+** ys ;
do $t \leftarrow$ head ts ;
(λb . **if** b **then** $h_{once} \star$ **return** `"heads"` **else** $h_{once} \star$ **return** `"tails"`) t
$\rightsquigarrow$ {- E-Do and E-HandRet -}
do $ts \leftarrow$ **do** $xs \leftarrow$ **return** [true] ; **do** $ys \leftarrow$ **return** [false] ; xs **+** ys ;
do $t \leftarrow$ head ts ;
(λb . **if** b **then** $h_{once} \star$ **return** `"heads"` **else** $h_{once} \star$ **return** `"tails"`) t
$\rightsquigarrow$ {- E-DoRet -}
do $ts \leftarrow$ [true, false] ;
do $t \leftarrow$ head ts ;
(λb . **if** b **then** $h_{once} \star$ **return** `"heads"` **else** $h_{once} \star$ **return** `"tails"`) t
$\equiv$ (λb . **if** b **then** $h_{once} \star$ **return** `"heads"` **else** $h_{once} \star$ **return** `"tails"`) true
$\rightsquigarrow$ {- E-AppAbs -}
$h_{once} \star$ **return** `"heads"`
$\rightsquigarrow$ {- E-HandRet -}
return [`"heads"`]

A.4 Forwarding

$h_{once} \star (run_{inc}\ 0 \star c_{fwd}) \equiv h_{once} \star$ (**do** $p' \leftarrow h_{inc} \star c_{fwd}$; p' 0)
$\rightsquigarrow$ {- E-Hand and E-Do and E-FwdSc -}
$h_{once} \star$ (**do** $p' \leftarrow$ **return** (λc . ($\lambda(p, k)$. **sc** `once` () (y . p y) (z . k z))
(λy . ($\lambda\_$. $h_{inc} \star c_{inc}$) y c,

$$\lambda(z, c') \,.\, (\lambda x \,.\, h_{inc} \star \mathbf{op}\ \mathtt{inc}\ ()\ (y \,.\, \mathbf{return}\ (x + y)))\ z\ c'))$$
$$;\, p'\ 0)$$

$\rightsquigarrow$ {- E-Hand and E-DoRet -}

$$h_{once} \star ((\lambda c \,.\, (\lambda (p, k) \,.\, \mathbf{sc}\ \mathtt{once}\ ()\ (y \,.\, p\ y)\ (z \,.\, k\ z))$$
$$(\lambda y \,.\quad (\lambda \_ \,.\, h_{inc} \star c_{inc})\ y\ c,$$
$$\lambda(z, c') \,.\, (\lambda x \,.\, h_{inc} \star \mathbf{op}\ \mathtt{inc}\ ()\ (y \,.\, \mathbf{return}\ (x + y)))\ z\ c'))\ 0)$$

$\rightsquigarrow$ {- E-Hand and E-AppAbs -}

$$h_{once} \star (\lambda (p, k) \,.\, \mathbf{sc}\ \mathtt{once}\ ()\ (y \,.\, p\ y)\ (z \,.\, k\ z))$$
$$(\lambda y \,.\quad (\lambda \_ \,.\, h_{inc} \star c_{inc})\ y\ 0,$$
$$\lambda(z, c') \,.\, (\lambda x \,.\, h_{inc} \star \mathbf{op}\ \mathtt{inc}\ ()\ (y \,.\, \mathbf{return}\ (x + y)))\ z\ c')$$

$\rightsquigarrow$ {- E-Hand and E-AppAbs -}

$$h_{once} \star (\mathbf{sc}\ \mathtt{once}\ ()\ (y \,.\, (\lambda y \,.\quad (\lambda \_ \,.\, h_{inc} \star c_{inc})\ y\ 0)\ y)$$
$$(z \,.\, (\lambda(z, c') \,.\, (\lambda x \,.\, h_{inc} \star \mathbf{op}\ \mathtt{inc}\ ()\ (y \,.\, \mathbf{return}\ (x + y)))\ z\ c')\ z))$$

$\rightsquigarrow$ {- E-HandSc -}

$$\mathbf{do}\ ts \leftarrow (\lambda y \,.\, h_{once} \star (\lambda y \,.\, (\lambda \_ \,.\, h_{inc} \star c_{inc})\ y\ 0)\ y)\ ()\,;$$
$$\mathbf{do}\ t \leftarrow \mathsf{head}\ ts\,;$$
$$(\lambda z \,.\, h_{once} \star (\lambda(z, c') \,.\, (\lambda x \,.\, h_{inc} \star \mathbf{op}\ \mathtt{inc}\ ()\ (y \,.\, \mathbf{return}\ (x + y)))\ z\ c')\ z)\ t$$

$\rightsquigarrow$ {- E-Do and E-AppAbs -}

$$\mathbf{do}\ ts \leftarrow h_{once} \star (\lambda y \,.\, (\lambda \_ \,.\, h_{inc} \star c_{inc})\ y\ 0)\ ())\,;$$
$$\mathbf{do}\ t \leftarrow \mathsf{head}\ ts\,;$$
$$(\lambda z \,.\, h_{once} \star (\lambda(z, c') \,.\, (\lambda x \,.\, h_{inc} \star \mathbf{op}\ \mathtt{inc}\ ()\ (y \,.\, \mathbf{return}\ (x + y)))\ z\ c')\ z)\ t$$

$\rightsquigarrow$ {- E-Do and E-Hand and E-AppAbs -}

$$\mathbf{do}\ ts \leftarrow h_{once} \star ((h_{inc} \star c_{inc})\ 0)\,;$$
$$\mathbf{do}\ t \leftarrow \mathsf{head}\ ts\,;$$
$$(\lambda z \,.\, h_{once} \star (\lambda(z, c') \,.\, (\lambda x \,.\, h_{inc} \star \mathbf{op}\ \mathtt{inc}\ ()\ (y \,.\, \mathbf{return}\ (x + y)))\ z\ c')\ z)\ t$$

$\rightsquigarrow$ {- E-Do and E-Hand and result of $(h_{inc} \star c_{inc})\ 0$ -}

$$\mathbf{do}\ ts \leftarrow h_{once} \star (\mathbf{op}\ \mathtt{choose}\ ()\ (b \,.\, \mathbf{if}\ b\ \mathbf{then}\ \mathbf{return}\ (0, 1)\ \mathbf{else}\ \mathbf{return}\ (0, 1)))\,;$$
$$\mathbf{do}\ t \leftarrow \mathsf{head}\ ts\,;$$
$$(\lambda z \,.\, h_{once} \star (\lambda(z, c') \,.\, (\lambda x \,.\, h_{inc} \star \mathbf{op}\ \mathtt{inc}\ ()\ (y \,.\, \mathbf{return}\ (x + y)))\ z\ c')\ z)\ t$$

$\rightsquigarrow$ {- E-Do and E-HandOp -}

$$\mathbf{do}\ ts \leftarrow (\mathbf{do}\ xs \leftarrow h_{once} \star ((b \,.\, \mathbf{if}\ b\ \mathbf{then}\ \mathbf{return}\ (0, 1)\ \mathbf{else}\ \mathbf{return}\ (0, 1))\ \mathsf{true})\,;$$
$$\mathbf{do}\quad ys \leftarrow h_{once} \star ((b \,.\, \mathbf{if}\ b\ \mathbf{then}\ \mathbf{return}\ (0, 1)\ \mathbf{else}\ \mathbf{return}\ (0, 1))\ \mathsf{false})\,;$$
$$xs + ys)\,;$$
$$\mathbf{do}\ t \leftarrow \mathsf{head}\ ts\,;$$
$$(\lambda z \,.\, h_{once} \star (\lambda(z, c') \,.\, (\lambda x \,.\, h_{inc} \star \mathbf{op}\ \mathtt{inc}\ ()\ (y \,.\, \mathbf{return}\ (x + y)))\ z\ c')\ z)\ t$$

$\rightsquigarrow$ {- E-Do and E-Hand and E-AppAbs -}

$$\mathbf{do}\ ts \leftarrow (\mathbf{do}\ xs \leftarrow h_{once} \star \mathbf{return}\ (0, 1)\,;$$
$$\mathbf{do}\quad ys \leftarrow h_{once} \star \mathbf{return}\ (0, 1)\,;$$
$$xs + ys)\,;$$
$$\mathbf{do}\ t \leftarrow \mathsf{head}\ ts\,;$$
$$(\lambda z \,.\, h_{once} \star (\lambda(z, c') \,.\, (\lambda x \,.\, h_{inc} \star \mathbf{op}\ \mathtt{inc}\ ()\ (y \,.\, \mathbf{return}\ (x + y)))\ z\ c')\ z)\ t$$

$\rightsquigarrow$ {- E-Do and E-HandRet -}

$$\mathbf{do}\ ts \leftarrow (\mathbf{do}\ xs \leftarrow \mathbf{return}\ [\,(0, 1)\,]\,;$$
$$\mathbf{do}\quad ys \leftarrow \mathbf{return}\ [\,(0, 1)\,]\,;$$

$$
\begin{array}{ll}
 & \quad\quad\quad \textbf{do} \quad\quad xs + ys)\,; \\
 & \quad \textbf{do}\ t \leftarrow \mathsf{head}\ ts\,; \\
 & \quad\quad (\lambda z\,.\,h_{once} \star (\lambda(z, c')\,.\,(\lambda x\,.\,h_{inc} \star \textbf{op}\ \mathtt{inc}\ ()\ (y\,.\,\textbf{return}\ (x+y)))\ z\ c')\ z)\ t \\
\equiv & \quad\quad (\lambda z\,.\,h_{once} \star (\lambda(z, c')\,.\,(\lambda x\,.\,h_{inc} \star \textbf{op}\ \mathtt{inc}\ ()\ (y\,.\,\textbf{return}\ (x+y)))\ z\ c')\ z)\ (0,1) \\
\rightsquigarrow & \quad \{\text{- E-Hand and E-AppAbs -}\} \\
 & \quad h_{once} \star ((\lambda(z, c')\,.\,(\lambda x\,.\,h_{inc} \star \textbf{op}\ \mathtt{inc}\ ()\ (y\,.\,\textbf{return}\ (x+y)))\ z\ c')\ (0,1)) \\
\rightsquigarrow & \quad \{\text{- E-Hand and E-AppAbs -}\} \\
 & \quad h_{once} \star ((\lambda x\,.\,h_{inc} \star \textbf{op}\ \mathtt{inc}\ ()\ (y\,.\,\textbf{return}\ (x+y)))\ 0\ 1) \\
\rightsquigarrow & \quad \{\text{- E-Hand and E-AppAbs -}\} \\
 & \quad h_{once} \star ((h_{inc} \star \textbf{op}\ \mathtt{inc}\ ()\ (y\,.\,\textbf{return}\ y))\ 1) \\
\rightsquigarrow & \quad \{\text{- E-Hand and E-Do and E-HandOp -}\} \\
 & \quad h_{once} \star (\textbf{return}\ (\lambda c\,.\,(\lambda y\,.\,h_{inc} \star \textbf{return}\ y)\ c\ (c+1))\ 1) \\
\rightsquigarrow & \quad \{\text{- E-Hand and E-DoRet -}\} \\
 & \quad h_{once} \star ((\lambda c\,.\,(\lambda y\,.\,h_{inc} \star \textbf{return}\ y)\ c\ (c+1))\ 1) \\
\rightsquigarrow & \quad \{\text{- E-Hand and E-AppAbs -}\} \\
 & \quad h_{once} \star ((\lambda y\,.\,h_{inc} \star \textbf{return}\ y)\ 1\ 2) \\
\rightsquigarrow & \quad \{\text{- E-Hand and E-AppAbs -}\} \\
 & \quad h_{once} \star ((h_{inc} \star \textbf{return}\ 1)\ 2) \\
\rightsquigarrow & \quad \{\text{- E-Hand and E-HandRet -}\} \\
 & \quad h_{once} \star (\textbf{return}\ (\lambda c\,.\,\textbf{return}\ (1, c))\ 2) \\
\rightsquigarrow & \quad \{\text{- E-HandRet and E-AppAbs -}\} \\
 & \quad \textbf{return}\ [\,(1,2)\,]
\end{array}
$$

B TYPE EQUIVALENCE RULES

This appendix shows the kinding rules of λ_{sc}. Figure 9 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

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

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

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

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

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

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

Fig. 9. Type equivalence of λ_{sc}.

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

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

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

Fig. 10. Row equivalence of λ_{sc}.

C KINDING RULES

This appendix shows the kinding rules of λ_{sc}. Figure 11 contains the rules.

$\boxed{\Gamma \vdash \sigma : K}$ $\boxed{\Gamma \vdash \underline{C}}$ Type kinding

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

$$\frac{\Gamma, \overline{\alpha} \vdash A : *}{\Gamma \vdash \forall\, \overline{\alpha}\,.\, A : *}\ \text{K-All} \qquad \frac{\Gamma \vdash A : *}{\Gamma \vdash A\,!\,\langle E \rangle}\ \text{K-Comp} \qquad \frac{\Gamma, \alpha \vdash A : K}{\Gamma \vdash \lambda\, \alpha\,.\, A : * \to K}\ \text{K-Abs}$$

$$\frac{\Gamma \vdash M : K_1 \to K_2 \qquad \Gamma \vdash A : K_1}{\Gamma \vdash M\ A : K_2}\ \text{K-App} \qquad \frac{\Gamma \vdash A : * \qquad \Gamma \vdash \underline{C}}{\Gamma \vdash A \to \underline{C} : *}\ \text{K-Fun}$$

$$\frac{\Gamma \vdash \underline{C} \qquad \Gamma \vdash \underline{D}}{\Gamma \vdash \underline{C} \Rightarrow \underline{D} : *}\ \text{K-Hand}$$

Fig. 11. Kinding rules of λ_{sc}.

D METATHEORY

D.1 Lemmas

Lemma D.1 (Function type canonical form). *For all* $\cdot \vdash v : \forall\ \overline{\overline{\alpha}\ \overline{\mu}}\ .\ A \to \underline{C}$, *if* Γ *does not contain any term bindings,* v *is of shape* $\boldsymbol{\lambda} x\ .\ c$.

Lemma D.2 (Handler type canonical form). *For all* $\Gamma \vdash v : \forall\ \overline{\overline{\alpha}\ \overline{\mu}}\ .\ A\ !\ E \Rightarrow M\ A\ !\ F$, *if* Γ *does not contain any term bindings,* v *is of shape* h *(a handler).*

Lemma D.3 (Canonical forms).

- *If* $\cdot \vdash v : A \to \underline{C}$ *then* v *is of shape* $\boldsymbol{\lambda} x\ .\ c$.
- *If* $\cdot \vdash v : \forall\ \alpha : *\ .\ \alpha\ !\ E \Rightarrow M\ \alpha\ !\ F$ *then* v *is of shape* h.

Lemma D.4 (Preservation of types under term substitution). *If* $\Gamma_1, x : \sigma, \Gamma_2 \vdash c : \underline{C}$ *and* $\Gamma_1 \vdash v : \sigma$, *then* $\Gamma_1, (\Gamma_2\ [\ v\ /\ x\ ]) \vdash c\ [\ v\ /\ x\ ] : \underline{C}$.

Lemma D.5 (Preservation of types under type substitution). *If* $\Gamma_1, \alpha, \Gamma_2 \vdash c : \underline{C}$ *and* $\Gamma_1 \vdash T : *$, *then Gamma1yt,* $(\Gamma_2\ [\ T\ /\ \alpha\ ]) \vdash c : (\underline{C}\ [\ T\ /\ \alpha\ ])$.

Lemma D.6 (Out of scope type substitution). *For all* α *and* T, *If* $\Gamma \vdash A : K$ *and* $\alpha \notin \Gamma$ *then* $A\ [\ T\ /\ \alpha\ ] = A$.

Lemma D.7 (Term weakening). *If* $\Gamma_1, \Gamma_3 \vdash c : \underline{C}$ *then* $\Gamma_1, \Gamma_2, \Gamma_3 \vdash c : \underline{C}$.

Lemma D.8 (Computation carrier kind). *If* $\Gamma \vdash v : A\ !\ E$ *then* $\Gamma \vdash A : *$.

Lemma D.9 (Bound type kind). *If* $\Gamma, x : A \vdash c : \underline{C}$ *then* $\Gamma \vdash A : *$.

Lemma D.10 (Kind term binding). *If* $\Gamma, x : B \vdash A : *$ *then* $\Gamma \vdash A : *$.

Lemma D.11 (Op membership). *If* $\Gamma \vdash$ **handler** $\{\ldots, oprs, \ldots\} : \forall\ \alpha\ .\ \alpha\ !\ E \Rightarrow M\ \alpha\ !\ F$ *and* **op** $\ell\ x\ k \mapsto c \in h$, *then there exists* $oprs_1$ *and* $oprs_2$ *such that* $oprs = oprs_1$, **op** $\ell\ k \vdash c, oprs_2$ *and* $\Gamma, \alpha \vdash$ **op** $\ell\ x\ k \mapsto c, oprs_2 : M\ \alpha\ !\ F$.

Lemma D.12 (Sc membership). *If* $\Gamma \vdash$ **handler** $\{\ldots, oprs, \ldots\} : \forall\ \alpha\ .\ \alpha\ !\ E \Rightarrow M\ \alpha\ !\ F$ *and* (**sc** $\ell\ x\ p\ k \mapsto c$) $\in h$, *then there exists* $oprs_1$ *and* $oprs_2$ *such that* $oprs = oprs_1$, **sc** $\ell\ x\ p\ k \vdash c, oprs_2$ *and* $\Gamma, \alpha \vdash$ **sc** $\ell\ x\ p\ k \mapsto c, oprs_2 : M\ \alpha\ !\ F$.

D.2 Subject reduction

Theorem 6.1 (Subject Reduction). *If* $\Gamma \vdash c : \underline{C}$ *and* $c \rightsquigarrow c'$, *then* $\Gamma \vdash c' : \underline{C}$.

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

- E-AppAbs: Inversion on $\Gamma \vdash (\boldsymbol{\lambda} x\ .\ c)\ \ v : B\ !\ F$ (T-App) gives $\Gamma \vdash \boldsymbol{\lambda} x\ .\ c : A \to B\ !\ F$ (1) and $\Gamma \vdash v : A$ (2). Inversion on fact 1 (T-Abs) gives $\Gamma, x : A \vdash c : B\ !\ F$ (3), which means the goal follows from facts 2 and 3 and Lemma D.4.
- E-Let: Inversion on $\Gamma \vdash$ **let** $x = v$ **in** $c : B\ !\ F$ (T-Let) gives $\Gamma \vdash v : \sigma$ (1) and $\Gamma, x : \sigma \vdash c : B\ !\ F$ (2), which means the goal follows from facts 1 and 2 and Lemma D.4.
- E-Do: Follows from the IH.
- E-DoRet: Inversion on $\Gamma \vdash$ **do** $x \leftarrow$ **return** v **in** $c : B\ !\ F$ (T-Do) gives $\Gamma \vdash$ **return** $v : A\ !\ F$ (1) and $\Gamma, x : A \vdash c : B\ !\ F$ (2). Inversion on (1) (T-Ret) gives $\Gamma \vdash v : A$ (3). The case follows from facts 2 and 3 and Lemma D.4.
- E-DoOp: Similar to E-DoSc. By inversion on $\Gamma \vdash$ **do** $x \leftarrow$ **op** $\ell\ v\ (y\ .\ c_1)$ **in** $c_2 : B\ !\ F$ (T-Do) we have that $\Gamma \vdash$ **op** $\ell\ v\ (y\ .\ c_1) : A\ !\ F$ (1) and $\Gamma, x : A \vdash c_2 : B\ !\ F$ (2). From inversion on fact 1

(T-Op) it follows that $F = \langle \ell, E\rangle$, $\ell : A_\ell \to B_\ell \in \Sigma$ (3), $\Gamma \vdash v : A_\ell$ (4), and $\Gamma, y : B_\ell \vdash c_1 : A \,!\, \langle \ell ; E\rangle$ (5). Lemma D.7 on (2) gives us $\Gamma, y : B_\ell, x : A \vdash c_2 : B \,!\, \langle \ell ; E\rangle$ (6). Facts 5 and 6 and rule T-Do give us $\Gamma, y : B_\ell \vdash \mathbf{do}\ x \leftarrow c_1\ \mathbf{in}\ c_2 : B \,!\, \langle \ell ; E\rangle$ (7). Our goal then follows from facts 3, 4, and 7 and rule T-Op.

- E-DoSc: Similar to E-DoOp. By inversion on $\Gamma \vdash \mathbf{do}\ x \leftarrow \mathbf{sc}\ \ell\ v\ (y\,.\,c_1)\ (z\,.\,c_2)\ \mathbf{in}\ c_3 : B \,!\, F$ (T-Do) we have that $\Gamma \vdash \mathbf{sc}\ \ell\ v\ (y\,.\,c_1)\ (z\,.\,c_2) : A \,!\, F$ (1) and $\Gamma, x : A \vdash c_3 : B \,!\, F$ (2). From inversion on fact 1 (T-Sc) it follows that $F = \langle \ell, E\rangle$, $\ell : A_\ell \to B_\ell \in \Sigma$ (3), $\Gamma \vdash v : A_\ell$ (4), $\Gamma, y : B_\ell \vdash c_1 : B' \,!\, \langle \ell ; E\rangle$ (5), and $\Gamma, z : B' \vdash c_2 : A \,!\, \langle \ell ; E\rangle$ (6). Lemma D.7 on (2) gives us $\Gamma, z : B', x : A \vdash c_3 : B \,!\, F$ (7), which means facts 6 and 7 and rule T-Do give us $\Gamma, z : B' \vdash \mathbf{do}\ x \leftarrow c_2\ \mathbf{in}\ c_3 : B \,!\, F$ (8). Our goal then follows from facts 3, 4, 5, and 8 and rule T-Sc.
- E-Hand: Follows from the IH.
- E-HandRet: By inversion on $\Gamma \vdash h \star \mathbf{return}\ v : B \,!\, F$ (T-Hand) we have that $B = M\ A$, $\Gamma \vdash h : \forall\ \alpha\,.\,\alpha \,!\, E \Rightarrow M\ \alpha \,!\, F$ (1), and $\Gamma \vdash \mathbf{return}\ v : A \,!\, E$(2). Inversion on fact 1 (T-Handler) gives us $\Gamma, \alpha, x : \alpha \vdash c_r : M\ \alpha \,!\, F$ (3). Inversion on fact 2 (T-Ret) gives us $\Gamma \vdash v : A$ (4). Lemma D.8 on fact 2 gives that $\Gamma \vdash A : *$ (5), which means that by facts 3 and 5 and Lemma D.5 we have that $\Gamma, x : A \vdash c_r : M\ A \,!\, F$ (6). Our goal now follows from facts 4 and 6 and Lemma D.4.
- E-HandOp: By inversion on $\Gamma \vdash h \star \mathbf{op}\ \ell\ v\ (y\,.\,c_1) : B \,!\, F$ (T-Hand) we have that $B = M\ A$, $\Gamma \vdash h : \forall\ \alpha\,.\,\alpha \,!\, E \Rightarrow M\ \alpha \,!\, F$ (1), and $\Gamma \vdash \mathbf{op}\ \ell\ v\ (y\,.\,c_1) : A \,!\, E$ (2). Inversion on fact 2 (T-Op) gives us that $\ell : A_\ell \to B_\ell \in \Sigma$ (3), $\Gamma \vdash v : A_\ell$ (4) and that $\Gamma, y : B_\ell \vdash c_1 : A \,!\, E$ (5). Lemma D.11 and fact 1 give us that $\Gamma, \alpha \vdash \mathbf{op}\ \ell\ x\ k \mapsto c, oprs : M\ \alpha \,!\, F$ (6). Inversion on fact 6 (T-OprOp) gives us that that $\Gamma, \alpha, x : A_\ell, k : B_\ell \to M\ \alpha \,!\, F \vdash c : M\ \alpha \,!\, F$ (7). Lemma D.8 on fact 2 gives that $\Gamma \vdash A : *$ (8). Facts 1 and 5 in combination with constructors T-Hand and T-Abs gives us that $\Gamma \vdash \boldsymbol{\lambda} y\,.\,h \star c_1 : B_\ell \to M\ A \,!\, F$ (9). Facts 7 and 8 and Lemma D.5 gives us that $\Gamma, x : A_\ell, k : B_\ell \to M\ A \,!\, F \vdash c : M\ A \,!\, F$ (10). Our goal now follows from facts 4, 9 and 10 and Lemma D.4.
- E-FwdOp By inversion on $\Gamma \vdash h \star \mathbf{op}\ \ell\ v\ (y\,.\,c) : B \,!\, F$ (T-Hand) have that $B = M\ A$, $\Gamma \vdash h : \forall\ \alpha\,.\,\alpha \,!\, E \Rightarrow M\ \alpha \,!\, F$ (1), and $\Gamma \vdash \mathbf{op}\ \ell\ v\ (y\,.\,c) : A \,!\, E$ (2). Inversion on fact 2 (T-Op) gives $\ell : A_\ell \to B_\ell$ (3), $\Gamma \vdash v : A_\ell$ (4), and $\Gamma, y : B_\ell \vdash c : A \,!\, E$ (5). The goal follows from facts 1, 2, and 4 and constructors T-Hand and T-Op.
- E-HandSc: By inversion on $\Gamma \vdash h \star \mathbf{sc}\ \ell\ v\ (y\,.\,c_1)\ (z\,.\,c_2) : B \,!\, F$ (T-Hand) we have that $B = M\ A, \Gamma \vdash h : \forall\ \alpha\,.\,\alpha \,!\, E \Rightarrow M\ \alpha \,!\, F$ (1), and $\Gamma \vdash \mathbf{sc}\ \ell\ v\ (y\,.\,c_1)\ (z\,.\,c_2) : A \,!\, E$ (2). Inversion on fact 2 (T-Sc) gives us that $\ell : A_\ell \to B_\ell \in \Sigma$ (3), $\Gamma \vdash v : A_\ell$ (4), $\Gamma, y : B_\ell \vdash c_1 : B \,!\, E$ (5), and $\Gamma, z : B \vdash c_2 : A \,!\, E$ (6). Lemma D.12 and fact 1 give us that $\Gamma, \alpha \vdash \mathbf{sc}\ \ell\ x\ p\ k \mapsto c, oprs : M\ \alpha \,!\, F$ (7). Inversion on fact 7 (T-OprSc) gives us that that $\Gamma, \alpha, \gamma, x : A_\ell, p : B_\ell \to M\ \gamma \,!\, F, k : \gamma \to M\ \alpha \,!\, F \vdash c : M\ \alpha \,!\, F$ (8). Lemma D.8 on fact 2 gives that $\Gamma \vdash A : *$ (9). Similarly, Lemmas D.8 and D.10 on fact 5 gives that $\Gamma \vdash B : *$ (10). Applying Lemmas D.5 and D.6 to facts 8 to 10 gives us that $\Gamma, x : A_\ell, p : B_\ell \to M\ B \,!\, F, k : B \to M\ A \,!\, F \vdash c : M\ A \,!\, F$ (11). Facts 1 and 5 in combination with constructors T-Hand and T-Abs gives us that $\Gamma \vdash \boldsymbol{\lambda} y\,.\,h \star c_1 : B_\ell \to M\ B \,!\, F$ (12). Similarly, facts 1 and 6 in combination with constructors T-Hand and T-Abs gives us that $\Gamma \vdash \boldsymbol{\lambda} z\,.\,h \star c_2 : B \to M\ A \,!\, F$ (13). Our goal then follows from facts 4 and 11 to 13 and Lemma D.4.
- E-FwdSc By inversion on $\Gamma \vdash h \star \mathbf{sc}\ \ell\ v\ (y\,.\,c_1)\ (z\,.\,c_2) : B \,!\, F$ (T-Hand) we have that $B = M\ A$, $\Gamma \vdash h : \forall\ \alpha\,.\,\alpha \,!\, E \Rightarrow M\ \alpha \,!\, F$ (1), and $\Gamma \vdash \mathbf{sc}\ \ell\ v\ (y\,.\,c_1)\ (z\,.\,c_2) : A \,!\, E$ (2). Pattern h must be of shape $\mathbf{handler}\ \{\ldots, oprs, \ldots\}$. Inversion on fact 1 (T-Handler) gives $\langle E\rangle = \langle labels\ (oprs) ; F\rangle$, and $\Gamma \vdash \mathbf{fwd}\ f\ p\ k \vdash c_f : M\ \alpha \,!\, \langle F\rangle$ (3). Inversion on fact 2 (T-Fwd) gives us that $\Gamma, \alpha, \beta, \gamma, p : \beta \to M\ \gamma \,!\, \langle F\rangle, k : \gamma \to M\ \alpha \,!\, \langle F\rangle, f : (\beta \to M\ \gamma \,!\, \langle F\rangle, M\ \gamma \to M\ \alpha \,!\, \langle F\rangle) \to M\ \alpha \,!\, \langle F\rangle \vdash c_f : M\ \alpha \,!\, \langle F\rangle$ (4). Inversion on fact 2 (T-Sc) gives $\ell : A_\ell \to B_\ell \in \Sigma$ (5), $\Gamma \vdash v : A_\ell$ (6), $\Gamma, y : B_\ell \vdash c_1 : B \,!\, \langle labels\ (oprs) ; F\rangle$ (7), and $\Gamma, z : B \vdash c_2 : A \,!\, \langle labels\ (oprs) ; F\rangle$ (8). Furthermore,

Since $(\mathbf{sc}\ \ell\ \_\ \_\ \_ \mapsto \_) \notin h$, ℓ is not in *labels* (*oprs*) of h, we have that ℓ is a member of F (9). Lemmas D.8 to D.10 on facts 2, 6 and 7 give us that $\Gamma \vdash A : *$ (10), $\Gamma \vdash B : *$ (11), and $\Gamma \vdash B_\ell : *$ (12). Facts 10 to 12 and Lemma D.5 give us that $\Gamma, p : B_\ell \to M\ B\,!\,\langle F\rangle, k : B \to M\ A\,!\,\langle F\rangle, f : (B_\ell \to M\ B\,!\,\langle F\rangle, M\ B \to M\ A\,!\,\langle F\rangle) \to M\ A\,!\,\langle F\rangle \vdash c_f : M\ A\,!\,\langle F\rangle$ (13). Facts 1 and 7 in combination with constructors T-Hand and T-Abs gives us that $\Gamma \vdash (\boldsymbol{\lambda} y\,.\,h \star c_1) : B_\ell \to M\ B\,!\,\langle F\rangle$ (14). Similarly, facts 1 and 8 in combination with constructors T-Hand and T-Abs gives us that $\Gamma \vdash \boldsymbol{\lambda} z\,.\,h \star c_2 : B \to M\ A\,!\,\langle F\rangle$ (15). From facts 5 and 6 and rules T-Var and T-App it follows that $\Gamma \vdash (\boldsymbol{\lambda}(p, k)\,.\,\mathbf{sc}\ \ell\ v\ (y\,.\,p\ y)\ (z\,.\,k\ z)) : (B_\ell \to M\ B\,!\,\langle F\rangle, M\ B \to M\ A\,!\,\langle F\rangle) \to M\ A\,!\,\langle F\rangle$ (16). Our goal then follows from facts 14 to 16 and Lemma D.4.

□

D.3 Progress

Theorem 6.2 (Progress). *If $\cdot \vdash c : \underline{C}$, then either:*

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

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

- T-App: Here, $\cdot \vdash v_1\ v_2$. Since v_1 has type $A \to B\,!\,F$, by Lemma D.3 it must be of shape $\boldsymbol{\lambda} x\,.\,c$, which means we can step by rule E-AppAbs.
- T-Do: Here, $\cdot \vdash \mathbf{do}\ x \leftarrow c_1\ \mathbf{in}\ c_2 : \underline{C}$. 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 $c_1 = \mathbf{op}\ \ell\ v\ (y\,.\,c)$ we can step by E-DoOp, and if $\mathbf{sc}\ \ell\ v\ (y\,.\,c_1')\ (z\,.\,c_2')$ we can step by E-DoSc.
- T-EqC: follows from the IH.
- T-Let: Here, $\cdot \vdash \mathbf{let}\ x = v\ \mathbf{in}\ c : \underline{C}$, which means we can step by E-Let.
- T-Ret, T-Op, and T-Sc: all of these are computation results (forms 1, 2, and 3, resp.).
- T-Hand: Here $\cdot \vdash v \star c : M\ A\,!\,F$. By Lemma D.3, 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 $\cdot \vdash h : \forall\ \alpha\,.\,\alpha\,!\,E \Rightarrow M\ \alpha\,!\,F$, there must be some $(\mathbf{return}\ x \mapsto c_r) \in h$ which means we can step by rule E-HandRet.
  (2) Case $c = \mathbf{op}\ \ell\ v\ (y\,.\,c')$. Depending on $(\mathbf{op}\ \ell\ x\ k \mapsto c) \in h$ we can step by E-HandOp or E-FwdOp.
  (3) Case $c = \mathbf{sc}\ \ell\ v\ (y\,.\,c_1)\ (z\,.\,c_2)$. If $(\mathbf{sc}\ \ell\ x\ p\ k \mapsto c) \in h$, we can step by E-HandSc. If not, since $\cdot \vdash h : \forall\ \alpha\,.\,\alpha\,!\,E \Rightarrow M\ \alpha\,!\,F$, there must be some $(\mathbf{fwd}\ f\ p\ k \mapsto c_f) \in h$ which means we can step by rule E-FwdSc.

□

E CODE FOR AUXILIARY FUNCTIONS

This appendix shows the code for some auxiliary functions in Section 7.

E.1 concatMap

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

E.2 concatMap$_{\mathsf{CutList}}$

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