First-class Names for Effect Handlers

Microsoft Technical Report, MSR-TR-2021-10, May 13, 2021, v1.

NINGNING XIE, Microsoft Research, USA
YOUYOU CONG, Tokyo Institute of Technology
DAANDAAN LEIJEN, Microsoft Research, USA

Algebraic effect handlers are a novel technique for adding composable computational effects to functional languages. While programming with distinct effects is concise, using multiple instances of the same effect is difficult to express. This work studies *named effect handlers*, such that an operation can explicitly yield to a specific handler by name. We propose a novel design of named handlers, where names are first-class values bound by regular lambdas, and are guaranteed not to escape using standard rank-2 polymorphism. We also formalize dynamically instantiated named handlers, which can express first-class isolated heaps with dynamic mutable references. Finally, we provide an implementation of named handlers in the Koka programming language, showing that the proposed ideas enable supporting named handlers with moderate effort.

1 INTRODUCTION

"*What's in a name? That which we call a rose by any other name would smell as sweet.*"
– William Shakespeare

Algebraic effect handlers [Plotkin and Power 2003; Plotkin and Pretnar 2013] are en vogue as a means to add composable computational effects to functional languages. By default, an effect operation is handled by the innermost handler that encloses it. As such, programming with distinct effects is concise. However, if one wishes an operation to be handled by a non-innermost handler, things become cumbersome. Such demand arises when programming with multiple *instances* of the *same* effect, which is necessary for modeling multiple mutable cells, multiple opened files, multiple sources of randomness, etc.

As an approach to handling multiple effect instances, Biernacki et al. [2019] and Zhang and Myers [2019] propose a generalization of effect handlers, where handlers are given explicit names, and where operations can name their preferred handler directly. This generalization is useful for implementing aforementioned examples, but it also complicates the theory and implementation of the language. Specifically, one must keep track of handler names that are currently available, and make sure that they do not *escape* their scope during evaluation. In previous studies, these are addressed by introducing names as *second-class* values using a special binder, and by employing a sophisticated type system with a form of dependent typing. Unfortunately, such treatment makes it harder to incorporate named handlers into an existing framework.

In this paper, we study a novel design of named handlers, where handler names are *first-class* values, and where well-scopedness of handler names is guaranteed without any non-standard binding or typing mechanism. The first-class status and well-scopedness guarantees are obtained via orthogonal extensions, and each of them is useful by itself. We combine these extensions in two different ways, both of which lead to a higher-order typed lambda calculus much like System F_ω but extended with row-polymorphic algebraic effects [Hillerström and Lindley 2016; Leijen 2014].

Our specific contributions can be summarized as follows:

- *Named handlers*: We propose a novel design of named effect handlers, where handler names are first-class, lambda-bound values. In Section 2.3, we give an overview of our design, and illustrate the flexibility of first-class names.
- *Scoped effects*: We next introduce *scoped effects*, which are a useful concept for associating resources to handlers. In Section 2.4, we show that scopes can be handled using standard rank-2

polymorphism [Jones 1996; McCracken 1984]. We then demonstrate that scoped effects allow us to implement a safe resource interface by modeling *locally isolated state* using scoped effects and *masking* [Biernacki et al. 2017; Convent et al. 2020; Leijen 2014; Wu et al. 2014].

- *Named handlers with scoped effects*: We then solve the name escaping problem of plain named handlers by incorporating scoped effects into the named handler calculus. In Section 2.5, we illustrate the design of the resulting system, focusing on how rank-2 types help us ensure well-scopedness of handler names. Then in Section 3, we formalize the system as System $\mathsf{F}^{\epsilon+\mathsf{sn}}$, and prove its type soundness.
- *Named handlers under scoped effects*: We further support *dynamic* creation of named handlers by incorporating the notion of *umbrella* effects [Leijen 2018]. In Section 2.6, we discuss implementation of reference cells as a motivation for dynamic named handlers, and show that umbrella effects allow us to implement isolated heaps with dynamically created references. Then in Section 4, we formalize the system as System $\mathsf{F}^{\epsilon+u}$, and prove that a practical subset of the system enjoys type soundness.
- We have implemented all the named handler variants, as well as locally isolated state, in the Koka programming language [Koka 2019]. We detail the implementation and provide examples in Section 5.

Finally, Section 6 discusses related work and Section 7 concludes.

Notes on the appendix. In the appendix of the supplementary material, we provide the full typing rules and soundness proofs of System $\mathsf{F}^{\epsilon+\mathsf{sn}}$ and System $\mathsf{F}^{\epsilon+u}$. We also present the specification of two systems that are not detailed in the paper, one featuring plain named handlers (Section 2.3) and the other featuring plain scoped effects (Section 2.4). These form the basis of the combinations discussed in Sections 3 and 4, but are not necessary for understanding this paper.

2 OVERVIEW

In this section, we shortly introduce algebraic effects and handlers, and outline the key ideas of our work.

2.1 Algebraic Effects

Algebraic effects [Plotkin and Power 2003] and handlers [Plotkin and Pretnar 2013] are a powerful abstraction of user-defined effects. When programming in a language with support for algebraic effects, one declares a new effect via an effect signature, consisting of a label and a list of *operations*. For example, an integer read effect has the following signature:

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

The effect has label *read* and a single operation *ask*. The type of *ask* tells us that this operation receives a unit argument, returns an integer value, and produces a read effect. To call the *ask* operation, one uses the perform keyword as shown below[1]:

$x \leftarrow \mathsf{perform}\ \mathit{ask}\ ();\ x\ +\ 1$

An effect signature only defines the type of operations; their interpretation is given separately by a *handler*. A handler takes the form $\mathsf{handler}\ h\ e$, where e is the computation to be handled (which we call an *action*), and h is a list of operation implementations[2]. Each operation implementation is a

[1]In examples we use the following syntactic sugar to express binding and sequencing: (1) $x \leftarrow e_1;\ e_2\ \triangleq\ (\lambda x.\ e_2)\ e_1$ and (2) $e_1;\ e_2\ \triangleq\ (\lambda\_.\ e_2)\ e_1$. For better illustration we often omit type (effect) abstractions, applications and annotations, but all examples can be rewritten and fully typed in our formalized systems presented later.

[2]For simplicity, we leave out the return clause of handlers. This does not cause loss of expressiveness, because we can always perform the computation of the return clause after obtaining a value from the handler.

function $\lambda x.\ \lambda k.\ e'$, where x stands for the argument of the operation, and k represents the delimited continuation within the handler, which can be used to resume the computation surrounding an operation. The following handler handles *ask* by applying k to 42, meaning that a call to *ask* is always interpreted as 42.

$$\begin{array}{ll} & \mathsf{handler}\ \{\ \mathit{ask} \mapsto \lambda x.\ \lambda k.\ k\ 42\ \}\ (x \leftarrow \mathsf{perform}\ \mathit{ask}\ ();\ x\ +\ 1) \\ \longmapsto^* & k\ 42 \quad \text{where } k\ =\ \lambda w.\ \mathsf{handler}\ \{\ \mathit{ask} \mapsto \lambda x.\ \lambda k.\ k\ 42\ \}\ (x \leftarrow w;\ x\ +\ 1) \\ \longmapsto^* & 43 \end{array}$$

Instead of resuming the continuation, a handler may abort the computation by discarding the continuation. Exceptions are a typical example of handlers that do not resume.

$$\mathit{exn}\ \{\ \mathit{throw}\ :\ \forall \alpha.\ ()\ \rightarrow^{\mathit{exn}}\ \alpha\ \}$$

Below is a handler for exceptions that converts any exceptional computation to a default value 42.

$$\begin{array}{ll} & \mathsf{handler}\ \{\ \mathit{throw} \mapsto \lambda x.\ \lambda k.\ 42\ \}\ (\mathsf{perform}\ \mathit{throw}\ ()) \\ \longmapsto^* & 42 \end{array}$$

Explicit handling of continuations also allows us to encode state. Here we implement *polymorphic* state by associating the label *st* with a type parameter α.

$$\mathit{st}\ \alpha\ \{\ \mathit{get}\ :\ ()\ \rightarrow^{\mathit{st}\ \alpha}\ \alpha,\ \ \mathit{set}\ :\ \alpha \rightarrow^{\mathit{st}\ \alpha}\ ()\ \}$$

$$\begin{array}{lll} h^{\mathit{st}} & \triangleq & \{\ \mathit{get} \mapsto \lambda x.\ \lambda k.\ (\lambda y.\ k\ y\ y)) \\ & & ,\ \mathit{set} \mapsto \lambda x.\ \lambda k.\ (\lambda y.\ k\ ()\ x))\ \} \end{array}$$

The above implementation directly corresponds to the monadic encoding [Kammar and Pretnar 2017]. In particular, performing an operation returns a function that takes in the current state. Below is an example illustrating how evaluation of *get* goes; note that the handler returns a function $\lambda z.\ x$ that ignores the final state z:

$$\begin{array}{ll} & \mathsf{handler}\ h^{\mathit{st}}\ (x \leftarrow \mathsf{perform}\ \mathit{get}\ ()\ +\ 1;\ \lambda z.\ x)\ 42 \\ \longmapsto^* & (\lambda y.\ k\ y\ y)\ 42 \quad \text{where } k\ =\ \lambda w.\ \mathsf{handler}\ h^{\mathit{st}}\ (x \leftarrow w\ +\ 1;\ \lambda z.\ x) \\ \longmapsto^* & 43 \end{array}$$

There exist many other applications of algebraic effects and handlers, including iterators/generators, async/await, coroutines, and probabilistic programming [Bauer and Pretnar 2015a; Leijen 2017; Pretnar 2015; Xie et al. 2020a]. In general, algebraic effects and handlers can express any control-flow manipulation that can be expressed using monads, at least in an untyped setting [Forster et al. 2019].

2.2 Named Handlers

It is sometimes inconvenient to work with plain effects and handlers, in particular when dealing with multiple handlers for a single effect. Let us consider the following program, which uses two handlers for the *read* effect:

$$\begin{array}{l} \mathsf{handler}\ \{\ \mathit{ask} \mapsto \lambda x.\ \lambda k.\ k\ 1\ \}\ (\\ \quad \mathsf{handler}\ \{\ \mathit{ask} \mapsto \lambda x.\ \lambda k.\ k\ 2\ \}\ (\mathsf{perform}\ \mathit{ask}\ ())) \end{array}$$

Operations are by default handled by the innermost handler. That means, the call to the *ask* operation in the above program is interpreted as 2, and thus the entire program evaluates to 2. But what if we wish *ask* to be handled by the outer handler? One possible solution is to use a technique called *masking* [Convent et al. 2020] (also called *injection* [Leijen 2014] and *lifting* [Biernacki et al. 2017] in the literature). Intuitively, masking allows one to skip the innermost handler surrounding an operation. For instance, the following program interprets *ask* using the outer *reader* handler, and thus evaluates to 1:

$$\begin{aligned}&\mathsf{handler}\ \{\ \mathit{ask} \mapsto \lambda x.\ \lambda k.\ k\ 1\ \}\ (\\ &\quad \mathsf{handler}\ \{\ \mathit{ask} \mapsto \lambda x.\ \lambda k.\ k\ 2\ \}\ (\mathsf{mask}^{\mathit{read}}\ (\mathsf{perform}\ \mathit{ask}\ ())))\end{aligned}$$

Programming with masking is however both cumbersome and fragile. In general, if we use mask to choose among nested handlers, we must know the exact number and order of handlers surrounding an expression.

A solution to the above problem is to *name* handlers explicitly and perform operations directly on named handlers. This has been attempted by previous work [Biernacki et al. 2019; Zhang and Myers 2019]. For instance, in the calculus of Biernacki et al. [2019], one can explicitly specify the handler we want to use through names a and b:

$$\begin{aligned}&\mathsf{handler}_a\ \{\ \mathit{ask} \mapsto \lambda x.\ \lambda k.\ k\ 1\ \}\\ &\quad (\mathsf{handler}_b\ \{\ \mathit{ask} \mapsto \lambda x.\ \lambda k.\ k\ 2\ \}\ (\mathsf{perform}_a\ \mathit{ask}\ ()))\end{aligned}$$

As a different approach, Zhang and Myers [2019] adopt implicit naming for the user language and internally elaborate programs to use explicit names. The elaboration forces operations to be handled by the closest, lexically enclosing handler, enabling modular reasoning in the presence of higher-order functions.

While existing studies [Biernacki et al. 2019; Zhang and Myers 2019] support selecting a specific handler, they both require a special binding mechanism for handlers names. Moreover, these studies treat handler names as *second-class* values. For example, consider the syntax of values and expressions in the calculus of Biernacki et al. [2019] (adapted to our notations):

$$\begin{aligned}v &::= x \mid \lambda x.\ e \mid \Lambda\alpha.\ e \mid \lambda\!\!\lambda a.\ e\\ e &::= v \mid e\ e \mid e\ [\sigma] \mid e\ a \mid \mathsf{perform}_a\ \mathit{op}\ v \mid \mathsf{handler}_a\ h\ e\end{aligned}$$

Notice that the syntax includes a new binder ($\lambda\!\!\lambda$) and application ($e\ a$) for handler names a. The special binding mechanism for names makes these systems deviate from standard lambda calculus, complicating both the meta-theory and implementation. As such, the syntax prohibits first-class use of handler names: we cannot return a name from a function (as in $\lambda\!\!\lambda a.\ a$), pass a name to a data type constructor (e.g., Just a), or create a list of names (e.g., $[a_1,\ a_2]$) and pick a handler from that list at runtime.

2.3 First-class Names

We propose a simpler approach to supporting named handlers. In our calculus, handler names are *first-class* values, and a handler uses a normal lambda binding to pass the handler name to an action. Performing an operation then takes the explicit handler name as a regular application. For example, the example with two *read* handlers is written as:

$$\begin{aligned}&\mathsf{handler}\ \{\ \mathit{ask} \mapsto \lambda z.\ \lambda k.\ k\ 1\ \}\ (\lambda x.\\ &\quad \mathsf{handler}\ \{\ \mathit{ask} \mapsto \lambda z.\ \lambda k.\ k\ 2\ \}\ (\lambda y.\ \mathsf{perform}\ \mathit{ask}\ x\ ()))\end{aligned}$$

The action being handled now receives the name of the handler as its argument (x and y), instead of a unit value. When performing *ask*, we specify the handler we want to use, in this case x.

During evaluation, a handler generates an internal frame handle_x marked by its (uniquely instantiated) concrete handler name x. A perform can then search for the matching handler in the evaluation context.

$$\begin{aligned}&\ \mathsf{handler}\ \{\ \mathit{ask} \mapsto \lambda z.\ \lambda k.\ k\ 1\ \}\ (\lambda x.\\ &\quad \mathsf{handler}\ \{\ \mathit{ask} \mapsto \lambda z.\ \lambda k.\ k\ 2\ \}\ (\lambda y.\ \mathsf{perform}\ \mathit{ask}\ x\ ()))\\ &\longmapsto^*\ \mathsf{handle}_x\ \{\ \mathit{ask} \mapsto \lambda z.\ \lambda k.\ k\ 1\ \}\\ &\quad \mathsf{handle}_y\ \{\ \mathit{ask} \mapsto \lambda z.\ \lambda k.\ k\ 2\ \}\ (\mathsf{perform}\ \mathit{ask}\ x\ ())\\ &\longmapsto^*\ 1\end{aligned}$$

The *generative* semantics for handlers is a generalization of the semantics designed by Xie et al. [2020a]. In their calculus, handler receives a unit-taking function representing a suspended computation. The idea of using unit-taking functions to represent computations has a close connection with call-by-push-value calculi [Levy 2006], which feature a strict value-computation distinction useful for modeling algebraic effect systems [Kammar and Pretnar 2017; Plotkin and Pretnar 2013]. In our calculus, we use unit-taking actions for unnamed handlers, and name-taking actions for named handlers. The generalized generative semantics is essential for preventing name escaping, as we will see in Section 3.2.

From a typing perspective, handler names are given an *evidence* type (ev). For instance, a named reader handler that always returns a specific constant x is defined as:

$$\begin{aligned}&\mathit{reader} \;:\; \forall \alpha\ \mu.\ \mathit{int} \to (\text{ev}\ \mathit{read} \to \langle \mathit{read} \mid \mu\rangle\ \alpha) \to \mu\ \alpha\\&\mathit{reader}\ x\ f \;=\; \text{handler}\ \{\ \mathit{ask} \mapsto \lambda().\ \lambda k.\ k\ x\ \}\ f\end{aligned}$$

We see that the name argument of the action f has type ev *read*, meaning that a read handler is available during evaluation of f. Other parts of the type signature are based on the standard, row-polymorphic effect system [Hillerström and Lindley 2016; Leijen 2005 2014; Xie et al. 2020a]. Here we see that action f has effect $\langle \mathit{read} \mid \mu\rangle$, which means it can perform the *read* effect and possibly more effects, denoted by the polymorphic effect variable μ. The handler construct discharges the *read* effect, hence the final effect is just μ. The empty effect is denoted by the empty row $\langle\rangle$ and is often omitted.

Now the differences between our work and previous work by Biernacki et al. [2019] and Zhang and Myers [2019] become clear. First, name bindings and evidence types in our system are treated as normal bindings and types. Second, names are plain variables (e.g., x, y), which can be used as first-class values. The latter means that we can easily construct a term like $\lambda x\!:\!\text{ev}\ \mathit{read}.\ x$ to pass names around, or build a list of reader evidences $[x_1,\ x_2]\ :\ [\text{ev}\ \mathit{read}]$ (where x_1 and x_2 are of type ev *read*) and pick one of them at runtime. This can be used to, for example, dispatch handlers according to configurations or specific applications.

2.4 Scoped Effects

Having handler names as first-class values is convenient, but it is also dangerous, as names can escape the scope of their handler. For instance, the following program fails to evaluate to a value.

$$\begin{array}{ll}& \mathit{reader}\ 1\ (\lambda x.\ (\mathit{reader}\ 2\ (\lambda y.\ (\lambda z.\ \text{perform}\ \mathit{ask}\ y\ ())))\ ())\\ \longmapsto^{*} & \text{handle}_x\ \{\ \mathit{ask} \mapsto \lambda y.\ \lambda k.\ k\ 1\ \}\\ & \quad(\text{handle}_y\ \{\ \mathit{ask} \mapsto \lambda y.\ \lambda k.\ k\ 2\ \}\ (\lambda z.\ \text{perform}\ \mathit{ask}\ y\ ())\ ())\\ \longmapsto^{*} & \text{handle}_x\ \{\ \mathit{ask} \mapsto \lambda y.\ \lambda k.\ k\ 1\ \}\ ((\lambda z.\ \text{perform}\ \mathit{ask}\ y\ ())\ ())\\ \longmapsto^{*} & \text{handle}_x\ \{\ \mathit{ask} \mapsto \lambda y.\ \lambda k.\ k\ 1\ \}\ (\text{perform}\ \mathit{ask}\ y\ ())\\ \not\longmapsto & \end{array}$$

Observe that handle_y returns a function that performs *ask* with name y. When this performing happens, however, the handler with name y is no longer present. This results in a failure of searching for a matching handler, which in turn makes the entire program get stuck.

Previous studies by Biernacki et al. [2019] and Zhang and Myers [2019] solve the name escaping issue in the following way. First, they expose handler names in the effect information of expressions. For example, Biernacki et al. [2019] would assign $\text{perform}_a\ \mathit{ask}\ v$ a type that mentions effect read_a, which implies the calculus must support a limited form of dependent typing. Second, they augment typing judgments with a separate environment for names. These have proven sufficient for ensuring well-scopedness of handler names, but from a practical point of view, the demand for limited dependent types and a separate name environment makes it difficult to implement handler

```
type ix  =  Ix int

vec{ push: string →^vec ix
   , find  : ix →^vec string }

withvec : ∀α μ. (() → ⟨vec | μ⟩ α) → μ α
withvec  =  handler [] {
  push ↦ λx k. v ← perform get ();
                 perform set (v ++ [x]);
                 k (Ix (length v))
, find ↦ λ(Ix i) k. v ← perform get ();
                      k (v[i])
}
```

(a) Unsafe interface

```
type ix η  =  Ix int  // index associated with scope η

vec{ push: string→^vec^η  ix η
   , find  : ix η   →^vec^η  string }  // scoped effect

withvec : ∀α μ. (∀η. () → ⟨vec^η | μ⟩ α) → μ α  // rank-2 type
withvec f  =  handler [] {
  push ↦ λx k. v ← perform get ();
                 perform set (v ++ [x]);
                 k (Ix (length v))
, find ↦ λ(Ix i) k. v ← perform get ();
                      k (v[i])
} f
```

(b) Safe interface with scoped effects

Fig. 1. A string vector resource using locally isolated state

names in an existing framework. Moreover, the approach cannot be taken when we like to use names are first-class values.

We propose to guarantee well-scopedness of names by *scoped* effects. As we will show in this section, scoped effects are useful on their own, and we treat naming and scoping as orthogonal concepts. In what follows, we take three steps to achieve our goal. First, we describe a novel way of using local state in a handler (Section 2.4.1), and motivate the need for scoping as an independent concept (Section 2.4.2). Next, we introduce scoped effects (Section 2.4.3), showing that they enable implementing a scope-safe resource interface. Note that the main focus of these sections is scoped effects themselves, hence handlers in these sections are all *unnamed*. Then, in Section 2.5, we combine scoped effects and *named* handlers as a solution to the name escaping problem.

2.4.1 Locally Isolated Handler State. Often a handler needs a form of local state. In the algebraic effects literature, an elegant solution is provided by *parameterized handlers* [Bauer and Pretnar 2015b; Leijen 2017; Pretnar 2010], which enable passing around a local parameter (i.e. state) when handling an operation or resuming a continuation. However, the threading of such handler parameters requires new evaluation rules for performing and handling, and increases the complexity of the semantics.

Here we present a new approach that requires no extensions to the core calculus. The idea is to use standard masking (discussed in Section 2.2) and a regular state handler (given in Section 2.1). More specifically, we handle any state-related effects using two handlers: an outer one as the regular state handler, and an inner one for the user-defined effect. The inner handler uses the state provided by the outer h^{st}, while wrapping its action around $\mathsf{mask}^{st\ \sigma}$ to make the state local. For convenience, we define a syntactic sugar ($\mathsf{handler}\ e\ h\ f$), denoting a handler h handling action f with a local state initialized to e[3]:

$$\begin{aligned}\mathsf{handler}\ e\ h\ f\ \triangleq\ &\mathsf{handler}\ h^{st}\ (\lambda\_.\\ &\quad x \leftarrow \mathsf{handler}\ h\ (\lambda\_.\ \mathsf{mask}^{st\ \sigma}\ (f\ ()));\ \lambda z.\ x)\ e\end{aligned}$$

With mask, the state used in its handler h is no longer exposed to f; it is only available to the handler operations.

[3]A small restriction on the encoding is that h cannot be a $st\ \sigma$ handler itself, or otherwise $mask^{st\ \sigma}$ would mask the wrong handler. The restriction is however not important in practice, as in that case h can store resources on its own without needing locally isolated state.

Using the above definition, we can define a handler for the *tick* effect, which counts the number of times its operation *tick* is performed:

$tick\ \{\ tick\ :\ ()\ \to^{tick}\ int\ \}$

$\text{handler}\ 0\ \{\ tick \mapsto \lambda x.\ \lambda k.\ i \leftarrow \text{perform}\ get\ ();$
$\qquad \text{perform}\ set\ (i+1);\ \ k\ i\ \}$

This is convenient in practice, and provides a full alternative to parameterized handlers. In a language with support for mutable state, we can further reduce the overhead caused by the monadic encoding by implementing the standard state handler using native state [Xie and Leijen 2020].

2.4.2 An Unsafe Resource Interface. While the use of mask guarantees local isolation of the state of a handler, the handler itself cannot yet isolate its own resources from other handlers. Consider the *vec* effect in Figure 1a, which is inspired by an example from Dreyer [2018]. The effect associates an abstract *index* (*ix*) with a string resource using locally isolated state[4]. In the definition of the *withvec* handler, we use a locally isolated list of strings, which is initially set to the empty list. The *push* operation appends a new string to the local list and returns its index, while *find* uses the index to look up the string again in the local state. Unfortunately, this interface is unsafe as indices are not bound to specific handlers. The following program shows how *find* may cause an out-of-bound error at runtime:

$withvec\ (\ \lambda\_.\ i \leftarrow \text{perform}\ push\ \text{"}hello\text{"};$
$\qquad withvec\ (\lambda\_.\ \text{perform}\ find\ i))\quad //\ out\ of\ bound$

The *find* operation is called with the index *i* obtained from the outer handler, but the operation is to be handled by the inner handler, whose state is the empty list. To avoid such unsafe lookups, we need to somehow associate indices with specific handlers.

2.4.3 A Safe Resource Interface. Fortunately, we already have the required expressive power in System F: we can manage the association between resources and handlers through *rank-2* polymorphic types [Jones 1996; McCracken 1984] together with *phantom types* [Hinze 2003; Leijen and Meijer 1999]. This is a well-understood technique used by Haskell's `runST` monad [Peyton Jones and Launchbury 1995] and other work on state isolation [Kammar et al. 2017; Launchbury and Sabry 1997; Timany et al. 2017].

To implement a safe *vec* effect, we use *scoped effects* with *rank-2 polymorphism*. First, we add a scoped type parameter η to *vec* effect, resulting in vec^{η} (Figure 1b).[5] We then assign handler in *withvec* a rank-2 type $\forall \alpha\ \mu.\ (\forall \eta.\ () \to \langle vec^{\eta} \mid \mu \rangle\ \alpha) \to \mu\ \alpha$. Here, the scoped type parameter η is *fully abstract* in the expression over which *withvec* is applied. That is, if μ is instantiated to some effect ϵ, and α to some type σ, we know $\eta \notin \text{ftv}(\epsilon,\ \sigma)$ by capture-avoiding substitution. Finally, we attach scope parameter η to the indices $ix\ \eta$ to associate an index to a specific handler scope.

Assuming the type *ix* is abstract (e.g., the constructor Ix is private), it is now guaranteed that the lookup $v[i]$ in *find* can never fail at runtime – the index *i* is always within the bounds of the local list. This is because each index $ix\ \eta$ is uniquely associated with the current handler instance vec^{η}, preventing us from passing to *find* an index returned from the *push* in some other instance:

$withvec\ (\lambda\_.\ i \leftarrow \text{perform}\ push\ \text{"}hello\text{"};$
$\qquad withvec\ (\lambda\_.\ \text{perform}\ find\ i))\quad //\ statically\ rejected$

Note that it is essential to attach a scoped type parameter η to not only the associated index ($ix\ \eta$) but also the effects (vec^{η}). If we kept *vec* unscoped, we could let resources escape the scope of the

[4]We use the keyword type to define a new type, and use $[\,]$, $+\!\!+$, and $v[i]$ to mean the empty list, list append, and list lookup.
[5]Note the difference between polymorphic effects, e.g., $st\ \alpha$, where the effect $st\ \alpha$ is *parameterized* by α, and scoped effects, e.g., vec^{η}, where the effect *vec* (without η) is *scoped* by η.

```
ref {  // scoped effect
      getref : () →^{ref^η} int
    , setref : int →^{ref^η} () }

makeref : ∀α. int →
              (∀η. ev ref^η → ⟨ref^η | μ⟩ α) → μ α
makeref i f =
  handler i {  // scoped and named handler
    getref ↦ λ(). λk. k (perform get ())
  , setref ↦ λx. λk. k (perform set x)
  } f
```

(a) Mutable references

```
heap {  // scoped effect
      newref : α →^{heap^η} (ev (ref α)^η) }
ref  {  // perform the heap effect
      getref : () →^{heap^η} int
    , setref : int →^{heap^η} () }

makeref : ∀α η. int → (ev ref^η → μ α) → μ α
makeref i f =
  handler i {  // named but unscoped handler
    getref ↦ λ(). λk. k (perform get ())
  , setref ↦ λx. λk. k (perform set x)
  } f
hp : ∀α. (∀η. () → ⟨heap^η | μ⟩ α) → μ α
hp f =
  handler {  // scoped but unnamed handler
    newref ↦ λx. λk. makeref x k
  } f
```

(b) First-class heap using umbrella effects

Fig. 2. Mutable references and first-class heap

handler by returning a lambda that performs *find* on an index captured by that lambda (as now the type of the lambda would not reflect the use of η anymore). If we scope *vec* as vec^η, any use of a resource must be reflected in the type of functions and data types, and the rank-2 polymorphic type of *withvec* prevents any possible resource escaping.

2.5 Combining Scoped Effects and Named Handlers

We have discussed the combination of scoped effects and unnamed handlers, showing that they allow for a safe implementation of resource interfaces. Now we look at the combination of scoped effects and named handlers, demonstrating that the former can be used to guarantee the well-scopedness of handler names.

In a calculus with scoped effects and named handlers, effect labels carry a scope parameter, and actions have a rank-2 type abstracting over their scope. For instance, the type of the named *reader* handler from Section 2.3 is redefined to:

$$\begin{aligned} &reader \;:\; \forall \alpha\, \mu.\; int \to (\forall \eta.\; \mathsf{ev}\; read^\eta \to \langle read^\eta \mid \mu\rangle\; \alpha) \to \mu\; \alpha \\ &reader \;=\; \lambda x.\, \lambda f.\, \mathsf{handler}\; \{\; ask \mapsto \lambda().\, \lambda k.\; k\; x \;\}\; f \end{aligned}$$

Now, the problematic example from Section 2.4, repeated below, is statically rejected, since the scope variable for name y would appear in the return type α of the handler.

```
// statically rejected
reader 1 (λx. (reader 2 (λy. λz. perform ask y ())) ())
```

Importantly, names in the combined system are still first-class values, and since they are guaranteed to be well-scoped by the use of rank-2 polymorphic types, we can prove that the combined system is *type sound*. This can be done without any special binding mechanism or dependent typing.

2.6 Scoping Named Handlers under Umbrella Effects

We have seen that the combination of named handlers and scoped effects gives us the well-scopedness guarantee of handler names. We now outline a different way of combining the two concepts, which enables us to express *dynamic instantiation* of named handlers.

In the existing type systems for algebraic effects [Biernacki et al. 2019; Leijen 2017; Zhang and Myers 2019], handlers manifest themselves in their type by discharging the effects being handled. A consequence of this design is that there can only be a static number of handlers at any time. As an example, consider an implementation of mutable reference cells based on scoped effects and named handlers (Figure 2a), and a program that uses those references:

$$\begin{aligned} &\mathit{tworef} \;:\; (\forall \eta_1\ \eta_2.\ \mathsf{ev}\ \mathit{ref}^{\eta_1} \to \mathsf{ev}\ \mathit{ref}^{\eta_2} \to \langle \mathit{ref}^{\eta_1} \mid \mathit{ref}^{\eta_2} \mid \mu\rangle\ \alpha) \to \mu\ \alpha \\ &\mathit{tworef}\ f \;=\; \mathit{makeref}\ 1\ (\lambda r_1.\ \mathit{makeref}\ 2\ (\lambda r_2.\ f\ r_1\ r_2)) \end{aligned}$$

Here, the existence of the two *makeref* handlers is directly reflected in the type of *tworef*, which discharges two distinct *ref* effects (ref^{η_1} and ref^{η_2}).

While *makeref* correctly implements the operations *getref* and *setref*, it does not allow us to use reference cells as flexibly as in a calculus with native support for heap references. For instance, we must use references under their own *ref* handler, which means we need to *statically* instantiate as many handlers as the number of distinct references required. Moreover, when references are implemented via scoped effects, different reference cells cannot be put in a homogeneous list as in $[r_1,\ r_2]$, since their types carry different scope variables (η_1 and η_2 respectively). Finally, there is no means to create fresh reference cells *dynamically* at runtime.

We propose a novel way of combining named handlers and scoped effects, where we can instantiate named handlers dynamically, and where different instances of a reference cell can share the same scope effect. The core idea is to scope all named handlers under a single scoped effect, which we call the *umbrella effect*. The notion of umbrella effects was first proposed by Leijen [2018], but not as a combination of named handlers and scoped effects, and without the well-scopedness guarantee for handler names.

2.6.1 A First-class Isolated Heap. It turns out umbrella effects are very expressive. Here we illustrate the expressiveness by implementing full *first-class isolated heaps* using scoped effects, and *dynamic mutable references* using named handlers. Figure 2b shows the implementation. First, we define the *heap* effect, which is a scoped effect with a single operation *newref* that returns a new name for a fresh reference cell. Second, we define the *ref* effect, whose *getref* and *setref* operations produce their umbrella effect heap^{η} rather than ref^{η}. This is key to enabling dynamic instantiation of references: it allows us to give a uniform effect type to all reference cells.

Having defined the two effects, we refine the signature (but not the implementation) of the *makeref* handler. Specifically, we move the universal quantification over the scope variable η outside of the action type, making ref^{η} connected to the heap effect heap^{η} that scopes over all dynamic references. Even though *makeref* still handles the operations of a particular reference, it no longer discharges the ref^{η} effect – all operations on references are instead reified under a single *umbrella* effect, namely heap^{η}.

Finally, we define the *hp* handler for the *heap* effect. We interpret the operation *newref* as a call to the *makeref* handler, with the action being the resumption k itself! This means a new named handler for *ref* is instantiated, and its name serves as the result of the call to *newref* – remember that an action of a named handler receives a name; using a resumption as an action is thus equivalent to passing a name to a resumption.

Example. Below, we give an example showing how the first-class heap works. Note that h^{heap^η} and h^{ref^η} denote the handler of *heap* and *ref* defined in Figure 2b.

```
handler h^{heap^η} (λ_. // heap
  r₁ ← perform newref 1;
  r₂ ← perform newref 2;
  perform getref r₁ () + perform getref r₂ ())

⟼* [handle_{r₁} 1 h^{ref^η}] ( // dynamically insert a new named handler (handle_{r₁})
    handle h^{heap^η} (
      r₂ ← perform newref 2;
      perform getref r₁ () + perform getref r₂ ()))

⟼* [handle_{r₁} 1 h^{ref^η}] (
    [handle_{r₂} 2 h^{ref^η}] ( // dynamically insert a new named handler (handle_{r₂})
    handle h^{heap^η} (
      perform getref r₁ () + perform getref r₂ ())))

⟼* 3
```

At each call to the operation *newref*, *heap* effectively pushes a new reference handler directly on top of h^{heap^η}. This corresponds to creating a new reference cell, denoted as a boxed area.

Note that the use of an umbrella effect is crucial in this example – without an umbrella effect, any newly pushed reference handler would *dynamically* change the effect type of the computation. Furthermore, the reification of the ref^η types under a single umbrella effect means we can use them homogeneously, for example putting them into a list, as in $[r_1,\ r_2]\ :\ [\text{ev}\ ref^\eta]$.

2.7 Desirable Properties

The aim of our work is to explore the design space of effect handlers with first-class names. As we saw in Section 2.4, naive named handlers suffer from the name escaping problem. To ensure well-scopedness of names with minimal effort, we develop an orthogonal concept called scoped effects, which are implemented via standard rank-2 polymorphism and are useful for enforcing safe usage of resources. We then combine named handlers and scoped effects in two different ways: by making every instance of a named handler scoped under its own effect, and by allowing multiple instances of a named handler to be scoped under an umbrella effect. While naive named handlers and scoped effects can be formalized independently (as we do in the appendix), in this paper we focus on the two combinations of the named handlers and scoped effects.

Given the novel design of our combined systems, it is important to guarantee that the systems are well-behaved. In this paper, we establish the following properties:

Well-scopedness of handler names. We show that names can never escape the scope of their handler. This property is subsumed by the *type soundness* theorem, which can be proved by showing the *preservation* and *progress* properties [Wright and Felleisen 1994]. Preservation guarantees that the well-typedness of expressions is preserved by reduction. Progress ensures that a well-typed expression always takes a step, which, in our context, means an operation performing always finds the corresponding named handler. One thing to note here is that type soundness of umbrella effects needs extra care. In particular, while the interface in Figure 2b is type-safe, general umbrella effects may result in accidental escaping of names. In Section 4.3, we discuss two type-theoretic restrictions on umbrella effects for ensuring type soundness, without losing the expressiveness required to encode first-class heaps.

$$
\begin{array}{llcl}
\text{Expression} & e & ::= & v \mid e\ e \mid e\ [\sigma] \\
 & & \mid & \mathsf{handle}^{\epsilon}_{m}\ h^{\ell^{\eta}}\ e \\
\text{Value} & v & ::= & x \mid \lambda^{\epsilon}(x : \sigma).\ e \mid \Lambda\alpha^{\kappa}.\ v \\
 & & \mid & \mathsf{handler}^{\epsilon}\ h^{\ell} \\
 & & \mid & \mathsf{perform}^{\epsilon}\ op\ \overline{\sigma} \\
 & & \mid & \mathsf{perform}^{\epsilon}\ op\ \overline{\sigma}\ v \\
 & & \mid & (m,\ h^{\ell^{\eta}}) \\
\text{Handler} & h & ::= & \{\ \overline{op_i \mapsto f_i}\ \} \\
 & & & \\
\text{Type} & \sigma & ::= & \alpha^{\kappa} \mid c^{\kappa}\ \overline{\sigma} \mid \sigma \to \epsilon\ \sigma \mid \forall\alpha^{\kappa}.\ \sigma \mid \mathsf{ev}\ \ell^{\eta} \\
\text{Effect row} & \epsilon & ::= & \langle\rangle \mid \langle \ell^{\eta} \mid \epsilon \rangle \\
\text{Kind} & \kappa & ::= & * \mid \kappa \to \kappa \mid \mathsf{lab} \mid \mathsf{eff} \mid \mathsf{S} \\
 & & & \\
\text{Term context} & \Gamma & ::= & \varnothing \mid \Gamma,\ x : \sigma \\
\text{Effect context} & \Sigma & ::= & \{\ \overline{\ell_i\ :\ sig^{\ell_i}}\ \} \\
\text{Effect signature} & sig^{\ell} & ::= & \{\ \overline{op_i\ :\ \forall\overline{\alpha_i}^{\kappa_i}.\ \sigma_i \to^{\ell^{\eta}} \sigma'_i}\ \}
\end{array}
$$

Fig. 3. Syntax of System $\mathsf{F}^{\epsilon+\mathsf{sn}}$

Uniqueness of names. We show that names can never be duplicated in evaluation contexts. In other words, the search for names is always deterministic. This property is not proved in the previous studies on named handlers: Biernacki et al. [2019] accept programs with duplicate names, arguing that duplication does no harm to type soundness; Zhang and Myers [2019] assume names (in their case, labels) are unique in the first place.

3 NAMED HANDLERS WITH SCOPED EFFECTS

In this section, we present System $\mathsf{F}^{\epsilon+\mathsf{sn}}$, a calculus of scoped effects and named handlers outlined in Section 2.5.

3.1 Syntax

Figure 3 defines the syntax of System $\mathsf{F}^{\epsilon+\mathsf{sn}}$. The system is explicitly typed; that is, every language construct is fully annotated with the effect, type, and kind information. The constructs highlighted in gray are internal forms generated during evaluation, and are not exposed to the user.

Expressions and Values. Expressions and values include variables, abstractions and applications for terms and types[6]. Type abstractions $\Lambda\alpha^{\kappa}.\ v$ require a value body. This is a common requirement for establishing type soundness of effectful calculi [Kammar and Pretnar 2017; Leijen 2017; Sekiyama and Igarashi 2019]; in our context, it is necessary for defining a type-erasure based semantics [Xie et al. 2020a]. To perform an effect operation op, we need to provide its type arguments $\overline{\sigma}$ to instantiate the type variables in the signature of op, and pass a handler name e_1 together with the operation argument e_2 via application. Thus, a fully applied operation looks like $\mathsf{perform}^{\epsilon}\ op\ \overline{\sigma}\ e_1\ e_2$; partially applied forms $\mathsf{perform}^{\epsilon}\ op\ \overline{\sigma}$ and $\mathsf{perform}^{\epsilon}\ op\ \overline{\sigma}\ v$ are both considered as values.

Handlers can be found in both expression and value categories. The value form $\mathsf{handler}^{\epsilon}\ h^{\ell}$ represents a named handler that, when given an *action* (thunkified expression), handles the operations in effect ℓ that are performed by the action. Here, we assume that h^{ℓ} has exactly one clause for each operation of effect ℓ. The expression form $\mathsf{handle}^{\epsilon}_{m}\ h^{\ell^{\eta}}\ e$ is an internal frame generated during

[6] We do not include constants (such as integers) in the formal systems, but we assume their existence in examples.

$$\frac{\Gamma \vdash_{\mathsf{val}} v : \sigma}{\Gamma \vdash v : \sigma \mid \epsilon}\ [\textsc{val}] \qquad \frac{x : \sigma \in \Gamma}{\Gamma \vdash_{\mathsf{val}} x : \sigma}\ [\textsc{var}] \qquad \frac{}{\Gamma \vdash_{\mathsf{val}} (m,\ h^{\ell^\eta}) : \mathsf{ev}\ \ell^\eta}\ [\textsc{ev}]$$

$$\frac{\Gamma, x : \sigma_1 \vdash e : \sigma_2 \mid \epsilon}{\Gamma \vdash_{\mathsf{val}} \lambda^\epsilon\,(x : \sigma_1).\ e : \sigma_1 \to \epsilon\ \sigma_2}\ [\textsc{abs}] \qquad \frac{\Gamma \vdash e_1 : \sigma_1 \to \epsilon\ \sigma \mid \epsilon \quad \Gamma \vdash e_2 : \sigma_1 \mid \epsilon}{\Gamma \vdash e_1\ e_2 : \sigma \mid \epsilon}\ [\textsc{app}]$$

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

$$\frac{\begin{array}{c} op_i : \forall\overline{\alpha}^{\overline{\kappa}}.\ \sigma_1 \to \ell^\eta\ \sigma_2 \in \Sigma(\ell) \quad \eta \in \overline{\alpha}^{\overline{\kappa}} \\ \Gamma \vdash_{\mathsf{val}} f_i : \forall\overline{\alpha}^{\overline{\kappa}}.\ \sigma_1 \to \epsilon\ ((\sigma_2 \to \epsilon\ \sigma) \to \epsilon\ \sigma) \quad \overline{\alpha}^{\overline{\kappa}} \notin \mathsf{ftv}(\sigma) \end{array}}{\Gamma \vdash_{\mathsf{ops}} \{\ op_1 \mapsto f_1,\ \ldots,\ op_n \mapsto f_n\ \} : \sigma \mid \ell \mid \epsilon}\ [\textsc{ops}]$$

$$\frac{op : \forall\overline{\alpha}^{\overline{\kappa}}.\ \sigma_1 \to \ell^\eta\ \sigma_2 \in \Sigma(\ell) \quad \eta \in \overline{\alpha}^{\overline{\kappa}} \quad \vdash_{\mathsf{wf}} \overline{\sigma} : \overline{\kappa}}{\Gamma \vdash_{\mathsf{val}} \mathsf{perform}^\epsilon\ op\ \overline{\sigma} : (\mathsf{ev}\ \ell^\eta \to \langle \ell^\eta \mid \epsilon\rangle\ \sigma_1 \to \langle \ell^\eta \mid \epsilon\rangle\ \sigma_2)[\overline{\alpha} := \overline{\sigma}]}\ [\textsc{perform}]$$

$$\frac{\Gamma \vdash_{\mathsf{ops}} h : \sigma \mid \ell \mid \epsilon \quad \eta \notin \mathsf{ftv}(\epsilon,\ \sigma)}{\Gamma \vdash_{\mathsf{val}} \mathsf{handler}^\epsilon\ h^\ell : (\forall\eta.\ \mathsf{ev}\ \ell^\eta \to \langle \ell^\eta \mid \epsilon\rangle\ \sigma) \to \epsilon\ \sigma}\ [\textsc{handler}]$$

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

(a) Typing

$$\frac{}{\epsilon \equiv \epsilon} \qquad \frac{\epsilon_1 \equiv \epsilon_2 \quad \epsilon_2 \equiv \epsilon_3}{\epsilon_1 \equiv \epsilon_3}$$

$$\frac{\epsilon_1 \equiv \epsilon_2}{\langle \ell_1^{\eta_1} \mid \ell_2^{\eta_2} \mid \epsilon_1\rangle \equiv \langle \ell_2^{\eta_2} \mid \ell_1^{\eta_1} \mid \epsilon_2\rangle} \qquad \frac{\epsilon_1 \equiv \epsilon_2}{\langle \ell^\eta \mid \epsilon_1\rangle \equiv \langle \ell^\eta \mid \epsilon_2\rangle}$$

(b) Equivalence of row-types

Fig. 4. Typing rules of System $\mathsf{F}^{\epsilon+\mathsf{sn}}$.

evaluation[7], and represents a specific instance of a handler. It associates the label ℓ with a scope η, and carries a marker m as an identifier of the handler. We call a pair $(m,\ h^{\ell^\eta})$ of a marker and a handler *evidence*, and assign it the type $\mathsf{ev}\ \ell^\eta$. Such an evidence pair serves as the *name* of handlers. Note that we could simply use markers as names, since they are sufficient for locating a handler. We couple it with a handler because this helps us demonstrate the correspondence between names and evidence in an evidence language [Xie et al. 2020a] (Section 5).

Types. Types include type variables α^k, type constructors $c^\kappa\ \overline{\sigma}$ (where c^κ is of kind κ), function types $\sigma \to \epsilon\ \sigma$, quantified types $\forall\alpha^\kappa.\ \sigma$, and the evidence type $\mathsf{ev}\ \ell^\eta$. Function types $\sigma_1 \to \epsilon\ \sigma_2$ consist of three components: an input type σ_1, an output type σ_2, and an effect type ϵ representing

[7]In Section 2.3, we used handle_x for better illustration. In the formalization, we use handle_m to stress that the generated handler names are *fresh*.

the effects of the function's body. An effect row is either empty $\langle\rangle$ (representing the *total* effect) or an extension $\langle l \mid \epsilon\rangle$ (meaning that ϵ is extended with effect label l).

To distinguish among regular value types (of kind $*$ and $\kappa \to \kappa$), effect labels ($\ell^\eta : \mathsf{lab}$), effect rows ($\epsilon : \mathsf{eff}$), and scopes (of kind S), we use a basic kind system to ensure well-formedness of types. We use μ to denote effect type variables α^{eff}, and η to denote scope type variables α^{S}. The kinding judgment $\vdash_{\mathsf{wf}}$ is standard, and is defined in the appendix.

3.2 Typing Rules

Figure 4a gives the typing rules of expressions, values, and handlers. The judgment for expressions $\Gamma \vdash e : \sigma \mid \epsilon$ means expression e has type σ under context Γ, and may perform operations associated with effect labels in ϵ. The judgment for values $\Gamma \vdash_{\mathsf{val}} v : \sigma$ lacks the effect component, reflecting the fact that values are effect-free. Finally, in the judgment $\Gamma \vdash_{\mathsf{ops}} h : \sigma \mid \ell \mid \epsilon$ for handlers, σ represents the type returned by h, ℓ represents the effect label being handled, and ϵ represents the effects to be handled by outer handlers.

To briefly go through the rules for expressions and values, rule VAL allows us to view a pure value as an effectful expression. Rule ABS concludes with a pure value while keeping the body's effects in the arrow type. Rule APP requires that the function, the argument, and the function's body have the same effect. Rule EV assigns evidence a type that carries an effect label ℓ and a scope variable η. Rule VAR, TABS and TAPP are completely standard.

Rule OPS imposes two requirements on the body of a handler. First, the types of the operation argument and the continuation of every handling function f_i agree with the type of the corresponding operation op_i. Second, the output types of those functions are all equal.

Rule PERFORM serves as the introduction rule for effects. We can see that the rule extends the original effect ϵ with an additional label ℓ^η, which comes from the signature of the operation being performed.

Rule HANDLER serves as the elimination rule for effects, and plays a crucial role in ensuring the type safety of named handlers. The rule derives the type $\sigma \mid \ell \mid \epsilon$ of the handler, and concludes with a rank-2 type, which tells us that the action it takes is polymorphic over a scope variable η. With $\eta \notin \mathrm{ftv}(\epsilon, \sigma)$, it is guaranteed that η cannot escape through the return type and effect. In the type of the action, the first component ev ℓ^η represents a handler name, and since ℓ^η is associated with η, it follows that this name cannot escape. In the effect of the action, the label ℓ^η represents the effect to be handled, and since a handled effect is not observable outside of the handler, we have a smaller effect ϵ as the return effect. The elimination of effects can also be observed in rule HANDLE, which takes care of an internal expression obtained by reducing handler.

Note that the representation of actions as functions are crucial in HANDLER. If actions were computations, we could not treat handlers as first-class values; more precisely, we could not give handler h a proper type as actions would require a computation type (in the sense of CBPV [Levy 2006]), which does not exist in our system. Furthermore, having actions as computations makes it impossible to assign them a rank-2 type, which means we would need some other means to ensure well-scopedness of names.

In addition to the typing rules, we need a set of rules for deciding whether two row types are equivalent or not (Figure 4b). Row equivalence is defined by reflexivity, transitivity, commutativity, and head equivalence. In calculi with row effects but unnamed handlers [Leijen 2017; Xie et al. 2020a], commutativity is restricted in that it only applies to distinct labels. In those calculi, an operation is always handled by the innermost handler, hence effects with the same label but different instantiations (e.g., *st int* and *st bool*) must be put in order. Here with named handlers, operations can be handled by an arbitrary handler specified by the user, so its label may be located anywhere in an effect row.

$$\text{Evaluation context} \quad \mathsf{E} \;::=\; \square \mid \mathsf{E}\; e \mid v\; \mathsf{E} \mid \mathsf{E}\;[\sigma] \mid \mathsf{handle}^{\epsilon}_{m}\; h^{\ell^{\eta}}\; \mathsf{E}$$

$$\begin{array}{lllll}
(\mathit{app}) & (\lambda^{\epsilon}(x:\sigma).\; e)\; v & \longrightarrow & e[x:=v] & \\
(\mathit{tapp}) & (\Lambda\alpha^{\kappa}.\; v)\; [\sigma] & \longrightarrow & v[\alpha:=\sigma] & \\
(\mathit{handler}) & (\mathsf{handler}^{\epsilon}\; h^{\ell})\; v & \longrightarrow & \mathsf{handle}^{\epsilon}_{m}\; h^{\ell^{\eta}}\; (v\;[\eta]\;(m, h^{\ell^{\eta}})) & \\
 & & & \text{where} \quad \eta,\; m \text{ fresh} & \\
(\mathit{return}) & \mathsf{handle}^{\epsilon}_{m}\; h^{\ell^{\eta}}\; v & \longrightarrow & v & \\
(\mathit{perform}) & \multicolumn{4}{l}{\mathsf{handle}^{\epsilon}_{m}\; h^{\ell^{\eta}}\; \mathsf{E}[\mathsf{perform}\; op\; \overline{\sigma}\; (m, h^{\ell^{\eta}})\; v]} \\
 & & \longrightarrow & f\;[\overline{\sigma}]\; v\; k \quad \text{iff } (op \mapsto f) \in h & \\
 & & & \text{where} \quad op : \forall\overline{\alpha}^{\overline{\kappa}}.\; \sigma_1 \to \ell^{\eta}\; \sigma_2 \;\in \Sigma(\ell) & \\
 & & & \qquad\qquad k \;=\; \lambda^{\epsilon}x:\sigma_2[\overline{\alpha}:=\overline{\sigma}].\; \mathsf{handle}^{\epsilon}_{m}\; h^{\ell^{\eta}}\; \mathsf{E}[x] &
\end{array}$$

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

Fig. 5. Operational Semantics of System $\mathsf{F}^{\epsilon+\mathsf{sn}}$

3.3 Operational Semantics

System $\mathsf{F}^{\epsilon+\mathsf{sn}}$ is equipped with a call-by-value, typed semantics defined in Figure 5. An evaluation context E is an expression template with a single hole $\square$ in it. The notation $\mathsf{E}[e]$ stands for an expression obtained by filling in the hole of E with expression e. Rule STEP defines one-step evaluation ($\longmapsto$) as a congruence of the small-step reduction ($\longrightarrow$).

Among the small-step rules, rule APP and rule TAPP are standard. Rule HANDLER is unique to our system: it generates a fresh scope variable η as well as a unique marker m. The scope variable is computationally irrelevant; it is only used to make the semantics fully typed. On the other hand, the marker plays an important role: it is used to identify a target handler in an evaluation context. After the reduction, the handler becomes a handle, whose action is passed the scope η and name $(m,\; h^{\ell^{\eta}})$.

Handling an action either results in a value via rule RETURN, or triggers evaluation of an operation clause via rule PERFORM. In the latter case, we search for the matching handler handle_m in the evaluation context, extract the implementation f of the performed operation, and apply f to the type instantiations $\overline{\sigma}$, the operation argument v, and the resumption k. Note that application of k is evaluated under the same handle_m frame again (meaning that handlers are *deep*), but f is not.

3.4 Type Soundness

The combination of naming and scoping leads to a sound type system. Following Wright and Felleisen [1994], we prove soundness through the preservation and progress theorems. Below is the statement of preservation, which can be shown in a relatively straightforward manner:

Theorem 3.1. (*Preservation of System* $\mathsf{F}^{\epsilon+\mathsf{sn}}$)
If $\varnothing \vdash e_1 : \sigma \mid \langle\rangle$ and $e_1 \longmapsto e_2$, then $\varnothing \vdash e_2 : \sigma \mid \langle\rangle$.

The progress theorem is trickier. For a well-typed expression, we know from its type that all effects are handled properly. However, the type information is not used at runtime; it is the marker that determines the handler associated with each operation. Then, how can we be sure that a particular marker exists in the evaluation context? Indeed, it turns out that the progress property does not hold for System $\mathsf{F}^{\epsilon+\mathsf{sn}}$ in general. For instance, the following expression is well-typed but does not take a step:

$\mathsf{handle}_{m_1}\ h^{\ell^\eta}\ (\mathsf{perform}\ op\ \eta\ (m_2,\ h^{\ell^\eta})\ ())\ \not\longrightarrow$

We find that the handler has marker m_1 for effect ℓ^η, but the operation requires marker m_2 for effect ℓ^η. The whole expression is judged well-typed by our type system, but it is stuck as there is no handler marked m_2 in the context.

On the other hand, the above program is not something that a user can write: it explicitly uses handle and evidence, which are *not* accessible to the user. Then, the reader may wonder: is it possible for a user to create an expression that causes failure of handler search? In particular, we are interested in user-written expressions without handle, and any expressions reduced from them during evaluation. For ease of reference, let us call such expressions *handle-safe* [Xie et al. 2020a]:

Definition 3.2. (*Handle-safe Expressions*)
A *handle-safe* expression is a well-typed, closed expression (with no term or type variables) that either (1) contains no handle term; or (2) is itself reduced from a handle-safe expression.

Note that handle-safe expressions still allow occurrences of handle, but only ones that are reduced from handler. Now we state our progress theorem as: if we start the evaluation from a handle-safe expression, we will never get stuck.

Theorem 3.3. (*Progress of Handle-safe System* $\mathsf{F}^{\epsilon+\mathsf{sn}}$)
If $\varnothing\ \vdash\ e_1\ :\ \sigma \mid \langle\rangle$ where e_1 is a handle-safe expression, then either e_1 is a value, or $e_1 \longmapsto e_2$ for some e_2.

Uniqueness of names. It might appear that rule (*perform*) renders the operational semantics non-deterministic, as there can potentially be multiple occurrences of m. For the operational semantics to be deterministic, all handlers must be unique in the evaluation context. This is not generally true, as we can easily construct two handle with the same m:

$\mathsf{handle}_m\ h^{\ell^\eta}\ (\mathsf{handle}_m\ h^{\ell^\eta}\ (\mathsf{perform}\ op\ \eta\ (m,\ h^{\ell^\eta})\ ()))$

However, the above example is again not a proper user program, as it uses handle and evidence directly. This makes us wonder whether we can avoid duplication of markers by considering only handle-safe expressions. It turns out that the answer is "yes": a handle construct produced from handler always has a freshly generated marker, and a marker can never be duplicated during evaluation:

Theorem 3.4. (*Uniqueness of Names for Handle-safe* $\mathsf{F}^{\epsilon+\mathsf{sn}}$)
For any handle-safe expression $\mathsf{E}_1[\mathsf{handle}_{m_1}\ h^{\ell_1^{\eta_1}}\ (\mathsf{E}_2[\mathsf{handle}_{m_2}\ h^{\ell_2^{\eta_2}}\ e])]$ in System $\mathsf{F}^{\epsilon+\mathsf{sn}}$, we have $m_1\ \neq\ m_2$.

4 NAMED HANDLERS UNDER SCOPED EFFECTS

We now formalize System $\mathsf{F}^{\epsilon+u}$, which combines named handlers and scoped effects through umbrella effects introduced in Section 2.6.

4.1 Syntax and Typing

The syntax of System $\mathsf{F}^{\epsilon+u}$ is basically the same as System $\mathsf{F}^{\epsilon+\mathsf{sn}}$. The only difference is that the typing of effect-related constructs depends on whether the effect label belongs to named handlers or scoped effects. Thus, we focus only on those constructs in this section. To distinguish between named handlers and scoped effects, we use label ℓ for effects with named handlers, and label l for scoped effects. For instance, in the case of the heap example in Figure 2b, we would have *ref* and *heap*.

$$\frac{op : \forall \overline{\alpha}^{\overline{\kappa}}.\, \sigma_1 \to l^{\eta}\ \sigma_2 \ \in \Sigma(l) \qquad \eta \in \overline{\alpha}^{\overline{\kappa}} \qquad \vdash_{\mathsf{wf}}\ \overline{\sigma} : \overline{\kappa}}{\Gamma \vdash_{\mathsf{val}}\ \mathsf{perform}^{\epsilon}\ op\ \overline{\sigma} : (\sigma_1 \to \langle l^{\eta} \mid \epsilon\rangle\ \sigma_2)[\overline{\alpha} := \overline{\sigma}]}\ [\textsc{u-perform}]$$

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

$$\frac{\Gamma \vdash_{\mathsf{ops}}\ h : \sigma \mid l \mid \epsilon \qquad \Gamma \vdash\ e : \sigma \mid \langle l^{\eta} \mid \epsilon\rangle}{\Gamma \vdash\ \mathsf{handle}^{\epsilon}\ h^{l^{\eta}}\ e : \sigma \mid \epsilon}\ [\textsc{u-handle}]$$

(a) Scoped effects (l)

$$\frac{op : \forall \overline{\alpha}^{\overline{\kappa}}.\, \sigma_1 \to l^{\eta}\ \sigma_2 \ \in \Sigma(\ell) \qquad \eta \in \overline{\alpha}^{\overline{\kappa}} \qquad \vdash_{\mathsf{wf}}\ \overline{\sigma} : \overline{\kappa}}{\Gamma \vdash_{\mathsf{val}}\ \mathsf{perform}^{\epsilon}\ op\ \overline{\sigma} : (\mathsf{ev}\ \ell^{\eta} \to \sigma_1 \to \langle l^{\eta} \mid \epsilon\rangle\ \sigma_2)[\overline{\alpha} := \overline{\sigma}]}\ [\textsc{n-perform}]$$

$$\frac{\Gamma \vdash_{\mathsf{ops}}\ h : \sigma \mid \ell \mid \epsilon}{\Gamma \vdash_{\mathsf{val}}\ \mathsf{handler}^{\epsilon}\ h^{\ell} : (\mathsf{ev}\ \ell^{\eta} \to \epsilon\ \sigma) \to \epsilon\ \sigma}\ [\textsc{n-handler}]$$

$$\frac{\Gamma \vdash_{\mathsf{ops}}\ h : \sigma \mid \ell \mid \epsilon \qquad \Gamma \vdash\ e : \sigma \mid \epsilon}{\Gamma \vdash\ \mathsf{handle}^{\epsilon}_{m}\ h^{\ell^{\eta}}\ e : \sigma \mid \epsilon}\ [\textsc{n-handle}]$$

(b) Named handlers (ℓ) under scoped effects (l)

Fig. 6. System $\mathsf{F}^{\epsilon+u}$

The typing rules of System $\mathsf{F}^{\epsilon+u}$ are given in Figure 6. The first half takes care of scoped effects, and the second half deals with named handlers. The separation of typing rules is unique to this system; recall that, in System $\mathsf{F}^{\epsilon+sn}$, scoped effects and named handlers are handled by a single typing rule, namely rule HANDLER in Figure 4.

Let us look at the rules for scoped umbrella effects (Figure 6a). To highlight the key ideas of umbrella effects, we design these rules for scoped effects with *unnamed* handlers; extending them to named handlers could be done easily. Since there are no handler names, operations are performed without an evidence, and are always handled by the innermost handler of effect l^{η} (rule U-PERFORM). Actions are polymorphic over the scope η (rule U-HANDLER) as in System $\mathsf{F}^{\epsilon+sn}$, but they now take a unit value instead of an evidence due to the absence of names. A handle construct simply eliminates effect l^{η} (rule U-HANDLE).

We next turn our attention to the rules for named handlers (Figure 6b). Observe that, when we perform an operation with an evidence for label ℓ (rule N-PERFORM), we produce its umbrella effect l that comes from the effect signature of ℓ. Since there is no ℓ effect, handlers do not discharge the effect of their action (rules N-HANDLER and N-HANDLE), but they keep the effect l^{η} scoped under η, which is the scope of its umbrella.

4.2 Operational Semantics

As with the typing rules, we have two sets of rules defining the operational semantics. The first three rules in Figure 7 are for handlers with scoped effects. Rule (*u-perform*) reduces a handler into a handle, while applying the action to a fresh scope variable and the unit value. Rule (*u-return*) simply returns a value. Rule (*u-perform*) searches for the handle frame that handles the raised

$$\begin{array}{llll}
(\textit{u-handler}) & (\mathsf{handler}^{\epsilon}\ h^{l})\ v & \longrightarrow & \mathsf{handle}^{\epsilon}\ h^{l^{\eta}}\ (v\ [\eta]\ ())\quad \text{where } \eta \text{ fresh} \\
(\textit{u-return}) & \mathsf{handle}^{\epsilon}\ h^{l^{\eta}}\ v & \longrightarrow & v \\
(\textit{u-perform}) & \mathsf{handle}^{\epsilon}\ h^{l^{\eta}}\ \mathsf{E}[\mathsf{perform}\ op\ \overline{\sigma}\ v] & & \\
 & & \longrightarrow & (f\ [\overline{\sigma}]\ v\ k)\quad \text{iff}\ \ op \notin \mathsf{bop}(\mathsf{E})\ \wedge\ (op \mapsto f)\ \in h^{l^{\eta}} \\
 & & & \text{where}\quad op : \forall \overline{\alpha}.\ \sigma_1 \to l^{\eta}\ \sigma_2\ \in \Sigma(l) \\
 & & & \qquad k\ =\ \lambda^{\epsilon} x\!:\!\sigma_2[\overline{\alpha} := \overline{\sigma}].\ \mathsf{handle}^{\epsilon}\ h^{l^{\eta}}\ \mathsf{E}[x]
\end{array}$$

(a) Scoped effects (l)

$$\begin{array}{llll}
(\textit{n-handler}) & (\mathsf{handler}^{\epsilon}\ h^{\ell})\ v & \longrightarrow & \mathsf{handle}^{\epsilon}_{m}\ h^{\ell^{\eta}}\ (v\ (m, h^{\ell^{\eta}}))\quad \text{where } m \text{ fresh} \\
(\textit{n-return}) & \mathsf{handle}^{\epsilon}_{m}\ h^{\ell^{\eta}}\ v & \longrightarrow & v \\
(\textit{n-perform}) & \mathsf{handle}^{\epsilon}_{m}\ h^{\ell^{\eta}}\ \mathsf{E}[\mathsf{perform}\ op\ \overline{\sigma}\ (m, h^{\ell^{\eta}})\ v] & & \\
 & & \longrightarrow & (f[\overline{\sigma}]\ v\ k)\quad \text{iff}\ (op \mapsto f)\ \in h^{\ell^{\eta}} \\
 & & & \text{where}\quad op : \forall \overline{\alpha}.\ \sigma_1 \to l^{\eta}\ \sigma_2\ \in \Sigma(\ell) \\
 & & & \qquad k\ =\ \lambda^{\epsilon} x\!:\!\sigma_2[\overline{\alpha} := \overline{\sigma}].\ \mathsf{handle}^{\epsilon}_{m}\ h^{\ell^{\eta}}\ \mathsf{E}[x]
\end{array}$$

(b) Named handlers (ℓ) under scoped effects (l)

Fig. 7. System $\mathsf{F}^{\epsilon+u}$: operational semantics

operation *op*. The condition $op \notin \mathsf{bop}(\mathsf{E})$ means that E has no handler for the operation *op*, i.e., the handler surrounding E is the innermost one.

The rest of the rules are for named handlers. Like (*u-perform*), rule (*n-perform*) reduces a handler into a handle, but unlike (*u-perform*), it applies the action to an evidence with a fresh marker *m*. Rules (*n-return*) and (*n-perform*) remain the same as the corresponding rules in System $\mathsf{F}^{\epsilon+\mathsf{sn}}$.

4.3 Type Soundness

Having walked through all the rules, we prove the meta theory of System $\mathsf{F}^{\epsilon+u}$. We first prove preservation:

Theorem 4.1. (*Preservation of System* $\mathsf{F}^{\epsilon+u}$)
If $\varnothing\ \vdash\ e_1\ :\ \sigma \mid \langle\rangle$ and $e_1 \longmapsto e_2$, then $\varnothing\ \vdash\ e_2\ :\ \sigma \mid \langle\rangle$.

As in System $\mathsf{F}^{\epsilon+\mathsf{sn}}$, proving progress is much more challenging. In particular, the system is unaware of any ℓ effect performed. On the other hand, scoping named handlers under an umbrella effect provides a form of safety guarantee, which implies that the first-class heap example in Section 2.6 is type-safe. To see why this is the case, observe that the effect ref^{η} is associated with η. This means the effect must be in the scope of the umbrella effect heap^{η}. Assuming the reference handler *makeref* is private, the heap handler $h^{\mathit{heap}^{\eta}}$ is the only place where a new reference handler may be generated. As performing operations in ref^{η} produces the heap^{η} effect, it must have $h^{\mathit{heap}^{\eta}}$ in the scope, which in turn ensures that $h^{\mathit{ref}^{\eta}}$ is in scope (since they are pushed right above $h^{\mathit{heap}^{\eta}}$).

Unfortunately, general umbrella effects may result in accidental name escaping. To illustrate the problem, let us assume that *heap* has another operation *bad*, and consider the following program[8]:

$$(\lambda f.\ f\ ())\ (\boxed{\mathsf{handle}_{r_1}\ 1\ h^{\mathit{ref}^{\eta}}}\ (\mathsf{handle}\ h^{\mathit{heap}^{\eta}}\ (\mathsf{perform}\ \mathit{bad}\ ();\ \mathsf{perform}\ \mathit{getref}\ r_1\ ()))$$

[8]While this example is judged ill-typed in System $\mathsf{F}^{\epsilon+u}$, it illustrates the basic idea of this problem. A well-typed but more involved example can be constructed by having $(\mathit{bad} \mapsto \lambda x\ k.\ (\lambda\_.\ k\ ()\ ()) \in h^{\mathit{heap}^{\eta}}$, and the program being $(\lambda f.\ f\ ())\ (\mathsf{handle}_{r_1}\ 1\ h^{\mathit{ref}^{\eta}}\ (\mathsf{handle}\ h^{\mathit{heap}^{\eta}}\ (\mathsf{perform}\ \mathit{bad}\ ();\ \mathsf{perform}\ \mathit{getref}\ r_1\ ();\ \lambda\_.\ ()))$.

where the *bad* operation in h^{heap^η} returns its resumption k, i.e., $(bad \mapsto \lambda x\ k.\ k\ \in h^{heap^\eta}$. In that case, the result would unwind the handlers.

$$\begin{array}{ll} & (\lambda f.\,f\ ())\ (\boxed{\mathsf{handle}_{r_1}\ 1\ h^{ref^\eta}}\ (\mathsf{handle}\ h^{heap^\eta}\ (\mathsf{perform}\ bad\ ();\ \mathsf{perform}\ getref\ r_1\ ()))\\ \longmapsto^* & (\lambda f.\,f\ ())\ (\boxed{\mathsf{handle}_{r_1}\ 1\ h^{ref^\eta}}\ k) \quad \text{where } k\ =\ \lambda x.\ \mathsf{handle}\ h^{heap^\eta}\ (x;\ \mathsf{perform}\ getref\ r_1\ ())\\ \longmapsto^* & (\lambda f.\,f\ ())\ k \end{array}$$

If we would then invoke k, we would still resume under the heap handler h^{heap^η} (since k captures h^{heap^η}), but the reference handler h^{ref^η} would no longer exist in the context.

$$\begin{array}{ll} \longmapsto^* & k\ ()\\ \longmapsto^* & \mathsf{handle}\ h^{heap^\eta}\ (\mathsf{perform}\ getref\ r_1\ ()) \end{array}$$

At that point, performing an operation in r_1 would get stuck.

To establish the progress property in System $\mathsf{F}^{\epsilon+u}$, we equip the type system with two restrictions drawn from the previous discussion on type-safe heaps (Section 2.6). These restrictions are: (1) an ℓ handler can only be used inside a handler for its umbrella effect l (like *makeref* in *hp*); and (2) for an umbrella effect, a resumption cannot be returned by a handler (so *bad* is statically rejected). We believe the restrictions are reasonable, since the restricted system is still expressive enough to encode first-class heaps, as *heap* satisfies both restrictions. With these restrictions, we can prove progress for handle-safe expressions with general umbrella effects. For space reasons, we give the full specification of the restrictions and its soundness proof in the appendix.

Theorem 4.2. (*Progress of Handle-safe System* $\mathsf{F}^{\epsilon+u}$)
If $\varnothing\ \vdash\ e_1\ :\ \sigma\ |\ \langle\rangle$ where e_1 is a handle-safe expression in restricted System $\mathsf{F}^{\epsilon+u}$, then either e_1 is a value, or $e_1 \longmapsto e_2$ for some e_2.

Again, names can never be duplicated during evaluation.

Theorem 4.3. (*Uniqueness of Names for Handle-safe* $\mathsf{F}^{\epsilon+u}$)
For any handle-safe expression $\mathsf{E}_1[\mathsf{handle}_{m_1}\ h^{\ell_1^{\eta_1}}\ (\mathsf{E}_2[\mathsf{handle}_{m_2}\ h^{\ell_2^{\eta_2}}\ e])]$ in System $\mathsf{F}^{\epsilon+u}$, we have $m_1\ \neq\ m_2$.

5 IMPLEMENTATION

We have implemented named-and-scoped handlers and named-under-umbrella effect handlers in the Koka compiler [Koka 2019]. In this section, we describe how Koka compiles named handlers, and what programs we can write using them.

5.1 Compiling Named Handlers

Koka is a programming language with full support for algebraic effects and handlers. Its compiler compiles via standard C code using Perceus style reference counting for memory management [Reinking et al. 2020]. To support effect handlers and first-class resumptions in C, the compiler uses two transformations. The first one targets an evidence calculus F^{ev} [Xie et al. 2020a], where every function receives the current *evidence vector*, making the search for the innermost handler explicit. The second one targets a polymorphic lambda calculus à la System F, using a standard multi-prompt delimited control monad [Gunter et al. 1995] to yield to a specific handler while capturing the resumption.

To see how Koka compiles unnamed handlers, let us look at the monadic translation of effect constructs in F^{ev}. The semantics of multi-prompt control is defined as follows, where every prompt is identified with a unique *marker* m and can be yielded to directly.

$$\begin{array}{lll} \mathsf{prompt}\ m\ v & \longrightarrow & v\\ \mathsf{prompt}\ m\ \mathsf{E}[\mathsf{yield}\ m\,f\ \ v] & \longrightarrow & f\ \ v\ (\lambda x.\ \mathsf{prompt}\ m\ \mathsf{E}[x]) \end{array}$$

After the evidence transformation, where every function receives the current evidence vector (denoted by w), the compiler translates unnamed handlers into a multi-prompt monad, which can be directly implemented in C:

$$\begin{array}{lll}\text{handler } h^{\ell}\ v\ w & \rightsquigarrow & \mathit{prompt}\ m\ (v\ ()\ (\mathit{insert}\ (m, h^{\ell})\ w)) \quad \text{with fresh } m\\ \text{perform } \mathit{op}^{\ell}\ v\ w & \rightsquigarrow & \mathit{yield}\ m\, f\ v \quad \text{iff } (m, h^{\ell})\ =\ \mathit{find}^{\ell}\ w\ \wedge (\mathit{op} \mapsto f) \in h^{\ell}\end{array}$$

Here, we see that a handler installs a prompt with a fresh marker m. The pair of a marker and a handler, (m, h), serves as the evidence of handler h, and is inserted into the current evidence vector w, which is passed to every function as a last argument. Performing an operation finds the evidence for its effect in the evidence vector, and uses the marker m from the evidence to yield to the corresponding prompt m.

The reason for calling the name of handlers *evidence* is apparent now: the representation (m, h) of a handler name is exactly the evidence that is used internally. This also means that we can directly translate all our variants of named handlers to the existing evidence calculus in Koka. The only difference from the implementation of unnamed handlers is that we (1) leave out insertion into the evidence vector in the handler rule, and (2) use handler names directly instead of searching the evidence vector in the perform rule:

$$\begin{array}{lll}\text{handler } h^{\ell}\ v\ w & \rightsquigarrow & \mathit{prompt}\ m\ (v\ (m, h^{\ell})\ w) \quad \text{with fresh } m\\ \text{perform } \mathit{op}\ (m, h^{\ell})\ v\ w & \rightsquigarrow & \mathit{yield}\ m\, f\ v \quad \text{iff } (\mathit{op} \mapsto f)\ \in h^{\ell}\end{array}$$

Now, the action v in the handler transitions is passed the evidence (m, h^{ℓ}) directly (as the name of the handler) instead of a unit argument. The other argument to v, namely the evidence vector w, is unchanged, as it is only used for regular, unnamed handlers. Correspondingly, the perform rule does not look up the evidence in the evidence vector as for regular handlers, but instead gets it directly (as the name of the handler). This is of course also the point where things can go wrong: if a named handler escapes its scope, the *yield* to m will find no matching prompt m in the evaluation context.

As we can imagine from the above description, there were very few changes that needed to be made to either the Koka runtime system or compiler, as all internal translations already use "names" (as evidence). The implementation is also consistent with the formalization presented in this paper, except with regard to the following points:

- In addition to named handlers with scoping, the Koka implementation also supports named but unscoped handlers. To ensure type safety, Koka inserts an exception effect that is raised if a specific handler is not found at runtime.
- The Koka implementation does not impose the two restrictions for umbrella effects (Section 4.3). Therefore, any umbrella operations induce an exception effect, which is raised if an umbrella handler escapes its scope. We feel adding the exception effect is a reasonable implementation tradeoff, but we may in the future add static checks to umbrella handler definitions to avoid this, and we see no fundamental challenges in adding those checks. Note however that the current treatment is already quite strict; for example, even in a pure language like Haskell, any demanded value may raise an exception or not terminate.

5.2 Examples

5.2.1 Koka Syntax. We now show examples of Koka programs that use named handlers. To help the reader understand the examples, we briefly introduce some of the syntax of Koka (see the Koka manual [Koka 2020] for a full description). Here is how to write the reader effect from Section 2.1:

```
effect reader {
  fun ask() : int
}
```

```
named effect file {
  fun read-line() : string
}

fun file(fname : path,
         action: file → ⟨exn,fsys|e⟩ a
        ) : ⟨exn,fsys|e⟩ a
{
  var ls := read-text-file(fname).lines
  with f = named handler {
    fun read-line() {
      match(ls) {
        Nil        { "" }
        Cons(x,xx) { ls := xx; x }
      }
    }
  }
  action(f)
}

fun main() {
  with f1 = file("foo.txt".path)
  with f2 = file("bar.txt".path)
  println( f1.read-line() + "," +
           f2.read-line() )
}
```

```
named scoped effect file⟨s⟩ {
  fun read-line() : string
}

fun file(fname : path,
         action: forall⟨s⟩ file⟨s⟩ → ⟨scope⟨s⟩,fsys|e⟩ a
        ) : ⟨fsys|e⟩ a
{
  var ls := read-text-file(fname).lines
  with f = named handler {
    fun read-line() {
      match(ls) {
        Nil        { "" }
        Cons(x,xx) { ls := xx; x }
      }
    }
  }
  action(f)
}

public fun main() {
  with f1 = file("foo.txt".path)
  with f2 = file("bar.txt".path)
  println( f1.read-line() + "," +
           f2.read-line() )
}
```

Fig. 8. Files as named handlers in Koka: plain named handler on the left, and named-and-scoped on the right.

```
fun main() {
  with handler {
    fun ask(){ 1 }
  }
  ask().println
}
```

The block starting with `effect reader` declares a new effect type `reader` with a single `ask` operation. The left-associative dot syntax in `ask().println` is used to chain function applications, and is equivalent to `println(ask())`. In Koka, we can call operations directly as regular functions, without using the perform keyword.

The `with` keyword is not specific to handlers; it is just convenient sugar to wrap the statements following it into the body of an anonymous function argument. It is defined as:

$$\begin{array}{lll} \text{with } f(e_1, \ldots, e_i) & \rightsquigarrow & f(e_1, \ldots, e_i,\ \text{fn}()\{\ \langle \mathit{body} \rangle\ \}) \\ \langle \mathit{body} \rangle & & \end{array}$$

There is a binding variant as well:

$$\begin{array}{lll} \text{with } x\ =\ f(e_1, \ldots, e_i) & \rightsquigarrow & f(e_1, \ldots, e_i,\ \text{fn}(x)\{\ \langle \mathit{body} \rangle\ \}) \\ \langle \mathit{body} \rangle & & \end{array}$$

The first variant is often used for unnamed handlers. In the reader example above, the body of `main` desugars to `(handler{...})(fn(){ println(ask()) })`. The expression `handler { ... }` returns a function that receives an action, which, in this case, is a computation that prints the result of $ask()$. The second variant, on the other hand, is useful for named handlers, as we will see shortly.

Finally, the implementation of the ask operation is written as `fun ask(){ 1 }` in the handler. The `fun` keyword is used for operations that are tail-resumptive; such operations implicitly resume with the final result. For the more general case, Koka provides a separate keyword `control`, which allows explicit resumptions as in our formalism. In the reader example, we use `fun` to simplify the program, but we can also write `control ask(){ resume(1) }` and obtain the same result.

5.2.2 Files as Named Handlers. Figure 8 shows how to model multiple opened files using named handlers. The two programs implement the same example; the difference is that the one on the left uses non-scoped named handlers, while the right one uses scoped named handlers. In both programs, the handler definition in `file` opens a file using library functions (causing the `fsys` effect), and stores its content as a list of lines using locally isolated state (declared with the `var` keyword). Each time the operation `read-line` is performed, the handler either returns the first line of the file, or returns the empty string when it reaches the end of the file. The `main` function uses the binding `with` to bind $\mathtt{f}_1$ and $\mathtt{f}_2$ to two `file` handlers.

Note the difference in the type signatures of the `file` handling function between the two programs. The named but unscoped handler on the left has an `exn` effect, as its operations can raise an exception if they are performed outside of their handlers scope. In contrast, the named and scoped handler on the right has no `exn` effect, as it guarantees well-scopedness through the rank-2 `s` scope parameter.

By looking at the type of the action in the scoped handler program, we further find a difference from the formal systems. In our formalization, we would use type ev file^η for the handler name and effect type file^η for the body effect. In the Koka implementation, on the other hand, we use `file⟨s⟩` for the handler name and a generic effect type $\mathit{scope}\langle\sigma\rangle$ for the body effect. We believe that the latter is more intuitive for programmers, but it works effectively the same since `file⟨s⟩` is internally an alias for evidence.

The two implementations in Figure 8 behave differently when a name escapes its scope. Consider the following use of the file handler.

```
fun wrong-escape() {
  with f = file("test.txt".path)
  fn(){ f.read-line() }
}
```

The program returns an anonymous function that captures the handler name `f`, which thus escapes its scope. The scoped handler implementation rejects this program outright with a static type error (as the action is no longer polymorphic in the scoped type parameter). In contrast, the unscoped handler implementation accepts the program but will raise an exception when applying the returned function.

5.2.3 First-class heap. Figure 9 shows a complete encoding of a first-class heap with dynamic mutable reference cells using umbrella effects, corresponding to the example in Figure 2b (Section 2.6.1). The implementation here is a bit more general, as `ref` is now a polymorphic resource. Notice also the rank-2 type for the `action` in the `hp` handler.

In the example, the `heap` effect is declared as a scoped but unnamed effect using the `scoped` keyword. Since it is scoped, the type of `heap` takes a scoped type variable `s` (as in `heap⟨s⟩`), and the handlers for `heap` are rank-2 polymorphic in the scoped type parameter. The `ref` effect is declared as a named effect but uses `in` keyword; this makes it a named effect under the umbrella `heap` effect and also modifies the signature of its operations to be in the `heap⟨s⟩` effect (instead of the `ref` effect). Finally, the heap handler implements the `new-ref` operation using `control`, so it can pass the resumption function `resume` as a first-class parameter to the `make-ref` handler for references.

6 RELATED WORK

Algebraic effects and handlers. The algebraic account of effects was first given by Plotkin and Power [2003], and later extended by Plotkin and Pretnar [2009] with handlers. In the subsequent years, we have seen a number of programming languages dedicated to effect handlers, including

```
scoped effect heap⟨s⟩ {
  control new-ref( init : a ) : ref⟨s,a⟩   /* a → ⟨heap⟨s⟩|e⟩ ref⟨s,a⟩ */
}

named effect ref⟨s,a⟩ in heap⟨s⟩ {
  fun get() : a                /* (ref⟨s,a⟩)   → ⟨heap⟨s⟩,exn⟩ a */
  fun set( value : a ) : ()    /* (ref⟨s,a⟩,a) → ⟨heap⟨s⟩,exn⟩ a */
}

fun make-ref(init,action) {
  var s := init
  with r = named handler {
    fun get() { s }
    fun set(x){ s := x }
  }
  action(r)
}

fun hp(action : forall⟨s⟩ () → ⟨heap⟨s⟩|e⟩ a): e a {
  with handler {
    control new-ref(init){ make-ref(init,resume) }
  }
  action()
}

fun main() {
  with hp
  val r₁ = new-ref(1)            /* ref⟨s,int⟩ */
  val r₂ = new-ref(2)            /* ref⟨s,int⟩ */
  println( r₁.get() + r₂.get() )
}
```

Fig. 9. First-class Heap in Koka

Eff [Pretnar 2015], Koka [Leijen 2017], Frank [Lindley et al. 2017], Links [Hillerström and Lindley 2016], Multicore OCaml [Dolan et al. 2017], and Effekt [Brachthäuser et al. 2020]. Recent work by Wu et al. [2014] introduces scoped syntax to control the interaction between effects, but it is fundamentally different from the scoped effects in our systems. Our systems are syntactically similar to the effect system of Xie et al. [2020a], which is based on System F_ω. The difference is that we have named handlers and scoped effects as additional features. The concept of umbrella effects comes from the work by Leijen [2018]. The novelty of our work is that we formalize umbrella effects as a combination of named handlers and scoped effects.

Type systems for named handlers. The type-safe treatment of named handlers was first considered by Bauer and Pretnar [2014]. They design a type system for an old version of the Eff language [Pretnar 2015], which features *effect instances*. Effect instances correspond to handler names, and can be used as first-class values. However, the formalized language does not support dynamic creation of effect instances, which is possible in our umbrella effect calculus. Also, the type system relies on a form of dependent typing, since it mentions effect instances in effect types.

More recently, Biernacki et al. [2019] and Zhang and Myers [2019] independently solve the challenge with typing named handlers. As we discussed earlier, they treat names as second-class values, and ensure well-scopedness of names by annotating every operation type with a handler name and augmenting every typing judgment with a name context. Our approach is more powerful and principled: we treat handler names as first class, and solve the scoping issue using standard rank-2 types.

Control operators with prompt tags. The notion of named handlers is closely related to multi-prompt delimited control operators [Gunter et al. 1995; Kiselyov 2012; Sitaram 1993], which allow one to specify the association between the control operator and the delimiter through *prompt tags*. The connection implies that prompt tags suffer from the same problem with handler names: without special care, prompt tags may escape their scope in the course of evaluation. However, none of the existing type systems for multi-prompt control operators statically ensures well-scopedness of prompt tags [Gunter et al. 1995; Kiselyov 2012; Takikawa et al. 2013].

Rank-2 polymorphism and encapsulation. The concept of scoped effects has a close connection with the monadic encapsulation of Haskell, more precisely, the `runST` function in the ST monad [Peyton Jones and Launchbury 1995]. `runST` has a rank-2 polymorphic type, which guarantees that references cannot escape the scope of the monad [Launchbury and Peyton Jones 1994]. Timany et al. [2017] present a logical relations model of a higher-order functional programming language featuring a Haskell-style ST monad type with `runST`, and prove that programs encapsulated by `runST` are independent of state. It has also been proved that programs encapsulated by `runST` are independent of state [Timany et al. 2017].

Reference cells as algebraic effects. An algebraic-effect-based implementation of reference cells has previously described by Kiselyov and Sivaramakrishnan [2017] as an application of *dynamic effects*. The implementation is in OCaml, and uses a library for multi-prompt delimited control operators [Kiselyov 2012]. As OCaml does not have effect typing, scoping of handler names is not statically enforced.

7 CONCLUSION

We explored the design space of named effect handlers, where names are first-class and well-scoped. The first property is obtained by using regular lambdas to bind names, while the latter is enforced by assigning handlers a rank-2 type. We look forward to investigating new programming techniques enabled by named effect handlers.

REFERENCES

Andrej Bauer, and Matija Pretnar. 2014. An Effect System for Algebraic Effects and Handlers. *Logical Methods in Computer Science* 10 (4).

Andrej Bauer, and Matija Pretnar. 2015a. Programming with Algebraic Effects and Handlers. *J. Log. Algebr. Meth. Program.* 84 (1): 108–123. doi:10.1016/j.jlamp.2014.02.001.

Andrej Bauer, and Matija Pretnar. 2015b. Programming with Algebraic Effects and Handlers. *Journal of Logical and Algebraic Methods in Programming* 84 (1). Elsevier: 108–123.

Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. Dec. 2017. Handle with Care: Relational Interpretation of Algebraic Effects and Handlers. *Proc. ACM Program. Lang.* 2 (POPL'17 issue): 8:1–8:30. doi:10.1145/3158096.

Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. Dec. 2019. Binders by Day, Labels by Night: Effect Instances via Lexically Scoped Handlers. *Proc. ACM Program. Lang.* 4 (POPL). Association for Computing Machinery, New York, NY, USA. doi:10.1145/3371116.

Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. 2020. Effects as Capabilities: Effect Handlers and Lightweight Effect Polymorphism. *Proc. ACM Program. Lang.* 5. ACM.

Lukas Convent, Sam Lindley, Conor McBride, and Craig McLaughlin. Jan. 2020. Doo Bee Doo Bee Doo. *In the Journal of Functional Programming*, January.

Stephen Dolan, Spiros Eliopoulos, Daniel Hillerström, Anil Madhavapeddy, KC Sivaramakrishnan, and Leo White. 2017. Effectively Tackling the Awkward Squad. In *ML Workshop*.

Derek Dreyer. 2018. The Type Soundness Theorem That You Really Want to Prove (and Now You Can). Milner Award Lecture, POPL 2018.

Yannick Forster, Ohad Kammar, Sam Lindley, and Matija Pretnar. 2019. On the Expressive Power of User-Defined Effects: Effect Handlers, Monadic Reflection, Delimited Control. *Journal of Functional Programming* 29. Cambridge University Press: 15. doi:10.1017/S0956796819000121.

Carl A. Gunter, Didier Rémy, and Jon G. Riecke. 1995. A Generalization of Exceptions and Control in ML-like Languages. In *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture*, 12–23. FPCA '95. ACM.

Daniel Hillerström, and Sam Lindley. 2016. Liberating Effects with Rows and Handlers. In *Proceedings of the 1st International Workshop on Type-Driven Development*, 15–27. TyDe 2016. Nara, Japan. doi:10.1145/2976022.2976033.

Ralf Hinze. 2003. Fun with Phantom Types. In *The Fun of Programming, Cornerstones of Computing*, edited by eremy Gibbons and Oege de Moor, 245–262. Palgrave Macmillan.

Mark P. Jones. 1996. Using Parameterized Signatures to Express Modular Structure. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 68–78. POPL '96. St. Petersburg Beach, Florida, USA. doi:10.1145/237721.237731.

Ohad Kammar, P. B. Levy, S. K. Moss, and Sam Staton. Jun. 2017. A Monad for Full Ground Reference Cells. In *2017 32nd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, 1–12. doi:10.1109/LICS.2017.8005109.

Ohad Kammar, and Matija Pretnar. Jan. 2017. No Value Restriction Is Needed for Algebraic Effects and Handlers. *Journal of Functional Programming* 27 (1). Cambridge University Press. doi:10.1017/S0956796816000320.

Oleg Kiselyov. 2012. Delimited Control in OCaml, Abstractly and Concretely. *Theoretical Computer Science* 435. Elsevier: 56–76.

Oleg Kiselyov, and KC Sivaramakrishnan. Dec. 2017. Eff Directly in OCaml. In *ML Workshop 2016*. `http://kcsrk.info/papers/caml-eff17.pdf`. Extended version.

Koka. 2019. `https://github.com/koka-lang/koka`.

Koka. 2020. `https://koka-lang.github.io/koka/doc/book.html`.

John Launchbury, and Simon L. Peyton Jones. 1994. Lazy Functional State Threads. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, 24–35. PLDI '94. Association for Computing Machinery, New York, NY, USA. doi:10.1145/178243.178246.

John Launchbury, and Amr Sabry. 1997. Monadic State: Axiomatization and Type Safety. In *Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming*, 227–238. ICFP '97. Amsterdam, The Netherlands. doi:10.1145/258948.258970.

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

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

Daan Leijen. Aug. 2016. *Algebraic Effects for Functional Programming*. MSR-TR-2016-29. Microsoft Research technical report.

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

Daan Leijen. 2018. First Class Dynamic Effect Handlers: Or, Polymorphic Heaps with Dynamic Effect Handlers. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Type-Driven Development*, 51–64. TyDe 2018. St. Louis, MO, USA.

Daan Leijen, and Erik Meijer. 1999. Domain Specific Embedded Compilers. In *In Proceedings of the 2nd Conference on Domain Specific Languages*, 109–122. Atlanta.

Paul Blain Levy. 2006. Call-by-Push-Value: Decomposing Call-by-Value and Call-by-Name. *Higher-Order and Symbolic Computation* 19 (4). Springer: 377–414.

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

McCracken. 1984. The Typechecking of Programs with Implicit Type Structure. In *Lecture Notes in Computer Science*, volume 173. Semantics of Data Types.

Simon L Peyton Jones, and John Launchbury. 1995. State in Haskell. *Lisp and Symbolic Comp.* 8 (4): 293–341. doi:10.1007/BF01018827.

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

Gordon D. Plotkin, and Matija Pretnar. Mar. 2009. Handlers of Algebraic Effects. In *18th European Symposium on Programming Languages and Systems*, 80–94. ESOP'09. York, UK. doi:10.1007/978-3-642-00590-9_7.

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

Matija Pretnar. Jan. 2010. Logic and Handling of Algebraic Effects. Phdthesis, University of Edinburgh.

Matija Pretnar. Dec. 2015. An Introduction to Algebraic Effects and Handlers. Invited Tutorial Paper. *Electron. Notes Theor. Comput. Sci.* 319 (C). Elsevier Science Publishers: 19–35. doi:10.1016/j.entcs.2015.12.003.

Alex Reinking, Ningning Xie, Leonardo de Moura, and Daan Leijen. Nov. 2020. *Perceus: Garbage Free Reference Counting with Reuse*. MSR-TR-2020-42. Microsoft.

Taro Sekiyama, and Atsushi Igarashi. 2019. Handling Polymorphic Algebraic Effects. In *European Symposium on Programming*, 353–380. Springer.

Dorai Sitaram. 1993. Handling Control. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, 147–155.

Asumu Takikawa, T Stephen Strickland, and Sam Tobin-Hochstadt. 2013. Constraining Delimited Control with Contracts. In *European Symposium on Programming*, 229–248. Springer.

Amin Timany, Léo Stefanesco, Morten Krogh-Jespersen, and Lars Birkedal. Dec. 2017. A Logical Relation for Monadic Encapsulation of State: Proving Contextual Equivalences in the Presence of RunST. *Proc. ACM Program. Lang.* 2 (POPL). doi:10.1145/3158152.

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

Nicolas Wu, Tom Schrijvers, and Ralf Hinze. 2014. Effect Handlers in Scope. In *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell*, 1–12. Haskell '14. Göthenburg, Sweden. doi:10.1145/2633357.2633358.

Ningning Xie, Jonathan Brachthäuser, Phillip Schuster, Daniel Hillerström, and Daan Leijen. Aug. 2020a. Effect Handlers, Evidently. In *Proceedings of the 25th ACM SIGPLAN International Conference on Functional Programming (ICFP'2020)*. ICFP '20. Jersey City, NJ.

Ningning Xie, Jonathan Brachthäuser, Phillip Schuster, Daniel Hillerström, and Daan Leijen. Aug. 2020b. *Effect Handlers, Evidently*. MSR-TR-2020-23. Microsoft Research technical report.

Ningning Xie, and Daan Leijen. Aug. 2020. Effect Handlers in Haskell, Evidently. In *Proceedings of the 2020 ACM SIGPLAN Symposium on Haskell*. Haskell'20. Jersey City, NJ.

Yizhou Zhang, and Andrew C. Myers. Jan. 2019. Abstraction-Safe Effect Handlers via Tunneling. *Proc. ACM Program. Lang.* 3 (POPL). ACM. doi:10.1145/3290318.

$$\frac{}{\vdash_{\mathsf{wf}}\ \alpha^{\kappa} : \kappa}\ [\textsc{wf-var}]$$

$$\frac{}{\vdash_{\mathsf{wf}}\ c^{\kappa} : \kappa}\ [\textsc{wf-con}]$$

$$\frac{\vdash_{\mathsf{wf}}\ \sigma_1 : \kappa_1 \rightarrow \kappa_2 \qquad \vdash_{\mathsf{wf}}\ \sigma_2 : \kappa_1}{\vdash_{\mathsf{wf}}\ \sigma_1\ \sigma_2 : \kappa_2}\ [\textsc{wf-app}]$$

$$\frac{\vdash_{\mathsf{wf}}\ \sigma_1 : * \qquad \vdash_{\mathsf{wf}}\ \sigma_2 : * \qquad \vdash_{\mathsf{wf}}\ \epsilon : \mathsf{eff}}{\vdash_{\mathsf{wf}}\ \sigma_1 \rightarrow \epsilon\ \sigma_2}\ [\textsc{wf-arrow}]$$

$$\frac{\vdash_{\mathsf{wf}}\ \sigma : *}{\vdash_{\mathsf{wf}}\ \forall\alpha^{\kappa}.\,\sigma : *}\ [\textsc{wf-forall}]$$

$$\frac{}{\vdash_{\mathsf{wf}}\ \mathsf{ev}\ \ell^{\eta} : *}\ [\textsc{wf-total}]$$

$$\frac{}{\vdash_{\mathsf{wf}}\ \ell^{\eta} : \mathsf{lab}}\ [\textsc{wf-total}]$$

$$\frac{}{\vdash_{\mathsf{wf}}\ \langle\rangle : \mathsf{eff}}\ [\textsc{wf-total}]$$

$$\frac{\vdash_{\mathsf{wf}}\ \ell^{\eta} : \mathsf{lab} \qquad \vdash_{\mathsf{wf}}\ \epsilon : \mathsf{eff}}{\vdash_{\mathsf{wf}}\ \langle \ell^{\eta} \mid \epsilon\rangle : \mathsf{eff}}\ [\textsc{wf-row}]$$

Fig. 10. Well-kindedness for System $\mathsf{F}^{\epsilon+\mathsf{sn}}$.

APPENDIX

A WELL-KINDEDNESS IN SYSTEM $\mathsf{F}^{\epsilon+\mathsf{sn}}$

To distinguish between value types, effect labels, effect rows, and scopes, we define a set of kinding rules for each system. In Figure 10, we present the kinding rules for System $\mathsf{F}^{\epsilon+\mathsf{sn}}$. We define similar rules for other systems as well.

B PLAIN NAMED EFFECT HANDLER CALCULUS

This section introduces System $\mathsf{F}^{\epsilon+\mathsf{n}}$, which has plain *named* handlers without scoping, as outlined in Section 2.3.

B.1 Typing Rules

The typing rules of System $\mathsf{F}^{\epsilon+\mathsf{n}}$ are given in Figure 11a. As can be seen from rule PERFORM, performing an operation with label ℓ requires an evidence of effect ℓ. Dually, N-HANDLER creates the initial evidence and passes it to its action.

Figure 11b gives the equivalence of row-types in System $\mathsf{F}^{\epsilon+\mathsf{n}}$.

$$\frac{op \,:\, \forall \overline{\alpha}^{\overline{\kappa}}.\, \sigma_1 \rightarrow \ell\ \sigma_2 \ \in \Sigma(\ell) \qquad \vdash_{\mathsf{wf}}\ \overline{\sigma} \,:\, \overline{\kappa}}{\Gamma \vdash_{\mathsf{val}}\ \mathsf{perform}^{\epsilon}\ op\, \overline{\sigma} \,:\, (\mathsf{ev}\ \ell \rightarrow \langle \ell \mid \epsilon\rangle\ \sigma_1 \rightarrow \langle \ell \mid \epsilon \rangle\ \sigma_2)[\overline{\alpha} := \overline{\sigma}]}\ [\textsc{perform}]$$

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

$$\frac{\Gamma \vdash_{\mathsf{ops}}\ h \,:\, \sigma \mid \ell \mid \epsilon \qquad \Gamma \vdash\ e \,:\, \sigma \mid \langle \ell \mid \epsilon \rangle}{\Gamma \vdash\ \mathsf{handle}^{\epsilon}_{m}\ h^{\ell}\ e \,:\, \sigma \mid \epsilon}\ [\textsc{handle}]$$

(a) Typing

$$\frac{}{\epsilon \equiv \epsilon} \qquad \frac{\epsilon_1 \equiv \epsilon_2 \quad \epsilon_2 \equiv \epsilon_3}{\epsilon_1 \equiv \epsilon_3} \qquad \frac{\epsilon_1 \equiv \epsilon_2}{\langle \ell_1 \mid \ell_2 \mid \epsilon_1 \rangle \equiv \langle \ell_2 \mid \ell_1 \mid \epsilon_2 \rangle} \qquad \frac{\epsilon_1 \equiv \epsilon_2}{\langle \ell \mid \epsilon_1 \rangle \equiv \langle \ell \mid \epsilon_2 \rangle}$$

(b) Equivalence of row-types

$$\begin{array}{lllll}
(\mathit{handler}) & (\mathsf{handler}^{\epsilon}\ h^{\ell})\ v & \longrightarrow & \mathsf{handle}^{\epsilon}_{m}\ h^{\ell} \cdot v\ (m, h^{\ell}) \quad \text{where } m \text{ fresh} \\
(\mathit{return}) & \mathsf{handle}^{\epsilon}_{m}\ h^{\ell} \cdot v & \longrightarrow & v \\
(\mathit{perform}) & \mathsf{handle}^{\epsilon}_{m}\ h^{\ell} \cdot \mathsf{E} \cdot \mathsf{perform}\ op\ \overline{\sigma}\ (m, h^{\ell})\ v & \longrightarrow & f[\overline{\sigma}]\ v\ k \quad \text{iff } (op \mapsto f) \in h \\
 & & & \text{where } \ op \,:\, \forall \overline{\alpha}.\, \sigma_1 \rightarrow \ell\ \sigma_2 \ \in \Sigma(l) \\
 & & & \qquad\quad k \ = \ \lambda^{\epsilon} x \,{:}\, \sigma_2[\overline{\alpha} := \overline{\sigma}].\ \mathsf{handle}^{\epsilon}\ h \cdot \mathsf{E} \cdot x
\end{array}$$

(c) Operational Semantics

Fig. 11. System $\mathsf{F}^{\epsilon+\mathsf{n}}$: Named handlers.

B.2 Operational Semantics

The operational semantics in Figure 11c shows how names are generated and used in System $\mathsf{F}^{\epsilon+\mathsf{n}}$. A handler (rule (*handler*)) creates a unique marker m, and passes an evidence $(m,\ h^{\ell})$ (i.e., the name) to the action v. The evidence is used by perform (rule (*perform*)) to find the matching handler $\mathsf{handle}^{\epsilon}_{m}$ in the evaluation context.

B.3 Preservation and Uniqueness

Let us now discuss the meta theory of System $\mathsf{F}^{\epsilon+\mathsf{n}}$. System $\mathsf{F}^{\epsilon+\mathsf{n}}$ enjoys the preservation property.

Theorem B.1. (*Preservation*)
If $\varnothing \vdash e_1 : \sigma \mid \langle\rangle$ and $e_1 \longmapsto e_2$, then $\varnothing \vdash e_2 : \sigma \mid \langle\rangle$.

However, $\mathsf{F}^{\epsilon+\mathsf{n}}$ does not have the *progress* property. That is, a well-typed expression may get stuck during evaluation. Recall the example shown in the beginning of Section 2.4:

$$\begin{array}{l}
\mathit{reader}\ 1\ (\lambda x.\ (\mathit{reader}\ 2\ (\lambda y.\ (\lambda z.\ \mathsf{perform}\ \mathit{ask}\ y\ ()\ )))\ ()) \\
\longmapsto^{*} \mathsf{handle}_{x}\ \{\ \mathit{ask} \mapsto \lambda y.\ \lambda k.\ k\ 1\ \} \\
\qquad (\mathsf{handle}_{y}\ \{\ \mathit{ask} \mapsto \lambda y.\ \lambda k.\ k\ 2\ \}\ (\lambda z.\ \mathsf{perform}\ \mathit{ask}\ y\ ())\ ()) \\
\longmapsto^{*} \mathsf{handle}_{x}\ \{\ \mathit{ask} \mapsto \lambda y.\ \lambda k.\ k\ 1\ \}\ ((\lambda z.\ \mathsf{perform}\ \mathit{ask}\ y\ ())\ ()) \\
\longmapsto^{*} \mathsf{handle}_{x}\ \{\ \mathit{ask} \mapsto \lambda y.\ \lambda k.\ k\ 1\ \}\ (\mathsf{perform}\ \mathit{ask}\ y\ ()) \\
\not\longmapsto
\end{array}$$

On the other hand, for handle-safe expressions, we can prove that names can never be duplicated in evaluation contexts.

$$\frac{op : \forall \overline{\alpha}^{\overline{\kappa}}.\, \sigma_1 \rightarrow \ell^{\eta}\ \sigma_2 \ \in \Sigma(\ell) \qquad \eta\ \in \overline{\alpha}^{\overline{\kappa}} \qquad \vdash_{\mathsf{wf}}\ \overline{\sigma}\ :\ \overline{\kappa}}{\Gamma\ \vdash_{\mathsf{val}}\ \mathsf{perform}^{\epsilon}\ op\, \overline{\sigma}\ :\ (\sigma_1 \rightarrow \langle \ell^{\eta} \mid \epsilon\rangle\ \sigma_2)[\overline{\alpha} := \overline{\sigma}]}\ [\textsc{perform}]$$

$$\frac{\Gamma\ \vdash_{\mathsf{ops}}\ h\ :\ \sigma \mid \ell \mid \epsilon \qquad \eta \notin \mathsf{fv}(\epsilon,\ \sigma)}{\Gamma\ \vdash_{\mathsf{val}}\ \mathsf{handler}^{\epsilon}\ h^{\ell}\ :\ (\forall \eta.\ () \rightarrow \langle \ell^{\eta} \mid \epsilon\rangle\ \sigma) \rightarrow \epsilon\ \sigma}\ [\textsc{handler}]$$

$$\frac{\Gamma\ \vdash_{\mathsf{ops}}\ h\ :\ \sigma \mid \ell \mid \epsilon \qquad \Gamma\ \vdash\ e\ :\ \sigma \mid \langle \ell^{\eta} \mid \epsilon\rangle}{\Gamma\ \vdash\ \mathsf{handle}^{\epsilon}\ h^{\ell^{\eta}}\ e\ :\ \sigma \mid \epsilon}\ [\textsc{handle}]$$

(a) Typing

$$\frac{}{\epsilon \equiv \epsilon} \qquad \frac{\epsilon_1 \equiv \epsilon_2 \qquad \epsilon_2 \equiv \epsilon_3}{\epsilon_1 \equiv \epsilon_3} \qquad \frac{\ell_1 \neq l_2 \qquad \epsilon_1 \equiv \epsilon_2}{\langle \ell_1^{\eta_1} \mid \ell_2^{\eta_2} \mid \epsilon_1\rangle \equiv \langle \ell_2^{\eta_2} \mid \ell_1^{\eta_1} \mid \epsilon_2\rangle} \frac{\epsilon_1 \equiv \epsilon_2}{\langle \ell^{\eta} \mid \epsilon_1\rangle \equiv \langle \ell^{\eta} \mid \epsilon_2\rangle}$$

(b) Equivalence of row-types

$$\begin{array}{lllll}
(\mathit{handler}) & (\mathsf{handler}^{\epsilon}\ h^{\ell})\ v & \longrightarrow & \mathsf{handle}^{\epsilon}\ h^{\ell^{\eta}} \cdot v\ [\eta]\ () \quad \text{where } \eta \text{ fresh} \\
(\mathit{return}) & \mathsf{handle}^{\epsilon}\ h^{\ell^{\eta}} \cdot v & \longrightarrow & v \\
(\mathit{perform}) & \mathsf{handle}^{\epsilon}\ h^{\ell^{\eta}} \cdot \mathsf{E} \cdot \mathsf{perform}\ op\ \overline{\sigma}\ v & \longrightarrow & f\,[\overline{\sigma}]\ v\ k \quad \text{iff } op \notin \mathsf{bop}(\mathsf{E})\ \wedge\ (op \mapsto f)\ \in h \\
& & & \text{where}\ \ op\ :\ \forall \overline{\alpha}.\ \sigma_1 \rightarrow \ell^{\eta}\ \sigma_2\ \in \Sigma(\ell) \\
& & & \qquad k\ =\ \lambda^{\epsilon} x\,{:}\,\sigma_2[\overline{\alpha} := \overline{\sigma}].\ \mathsf{handle}^{\epsilon}\ h^{\ell^{\eta}} \cdot \mathsf{E} \cdot x
\end{array}$$

(c) Operational Semantics

Fig. 12. $\mathsf{F}^{\epsilon+\mathsf{s}}$: scoped effects

Theorem B.2. (*Uniqueness of Handlers for Handle-safe System* $\mathsf{F}^{\epsilon+\mathsf{n}}$)
For any handle-safe expression $\mathsf{E}_1\ [\mathsf{handle}^{\epsilon_1}_{m_1}\ h^{\ell 1}\ (\mathsf{E}_2\ [\mathsf{handle}^{\epsilon_2}_{m_2}\ h^{\ell 2}\ e_0])]$, in System $\mathsf{F}^{\epsilon+\mathsf{n}}$, we have $m_1\ \neq m_2$.

C PLAIN SCOPED EFFECT CALCULUS

This section introduces System $\mathsf{F}^{\epsilon+\mathsf{s}}$, which has plain *scoped effect* without named handlers, as outlined in Section 2.4.

C.1 Typing Rules

Figure 12a shows the typing rules. Rule HANDLER is the key to ensuring well-scopedness of effects. It derives the type $\sigma \mid \ell \mid \epsilon$ of the handler h, and concludes with a rank-2 polymorphic type with $\eta \notin \mathsf{ftv}(\epsilon, \sigma)$, i.e., such that η cannot escape through either ϵ or σ.

Scoped effects also require some modifications to the row equivalence rules, which we present in Figure 12b. In particular, row commutativity only swaps distinct labels; it does not swap same labels with distinct scope variables. The design is crucial to maintaining the dynamic untyped semantics of effect handlers: after erasing all types, we can still evaluate the program and obtain the same result. This may not hold if row equivalence would depend on scope variables, e.g.:

$$\begin{array}{l}\mathsf{handler}^{\epsilon}\ h_1^{\ell}\ (\Lambda\eta_1.\ \lambda^{\langle \ell_1^{\eta_1} \mid \epsilon\rangle}\ \_. \\ \quad (\mathsf{handler}^{\langle \ell^{\eta_1} \mid \epsilon\rangle}\ h_2^{\ell}\ (\Lambda\eta_2.\ \lambda^{\langle \ell^{\eta_2} \mid \ell^{\eta_1} \mid \epsilon\rangle}\ \_.\ \mathsf{perform}\ op\ \eta_1\ ())))\end{array}$$

This program is rejected by the type system: the inner action performs an effect with scope η_1, while being surrounded by an intervening handler with scope η_2. This leads to a mismatch between the actual effect ($\langle \ell^{\eta_1} \mid \ell^{\eta_2} \mid\ \epsilon\rangle$) and the expected effect ($\langle \ell^{\eta_2} \mid \ell^{\eta_1} \mid \epsilon\rangle$) of the action, to which the commutativity rule is not applicable. If row commutativity would take scoped variables into account and allow the swap, then static typing would conclude that the operation is handled by the first handler (η_1), but under the scope-erasure semantics the innermost handler (η_2) would actually handle the operation! Note however that we can still use η_1 to handle the operation under the current rules, by using mask explicitly:

$$\begin{array}{l} \mathsf{handler}^{\epsilon}\ h_1^{\ell}\ (\Lambda\eta_1.\ \lambda^{\langle \ell_1^{\eta_1} \mid \epsilon\rangle}\ \_. \\ \quad (\mathsf{handler}^{\langle \ell^{\eta_1} \mid \epsilon\rangle}\ h_2^{\ell}\ (\Lambda\eta_2.\ \lambda^{\langle \ell^{\eta_2} \mid \ell^{\eta_1} \mid \epsilon\rangle}\ \_. \\ \qquad \mathsf{mask}^{\ell^{\eta_2}}\ (\mathsf{perform}\ op\ \eta_1\ ()))))) \end{array}$$

C.2 Operational Semantics

Scopes play an essential role during typing, but not during evaluation – they are computationally irrelevant. Thus, if we erase all scope variables, abstractions and applications, we can treat handlers as normal unscoped handlers. In this work, we take a different approach: we give direct operational rules for scoped effects.

We define the direct operational semantics of System $\mathsf{F}^{\epsilon+s}$ in Figure 12c. The key rule is (*handler*). As we saw in the typing rule of handler, the handled action v requires a scope variable. Hence, we create a fresh scope variable η, pass it to v, and continue with handle. Another important rule is (*perform*).

One thing to note here is that, while scopes cannot escape into the return type or effect of handler, they may still appear free in the return value.

$$\begin{array}{l} \mathsf{handler}^{\epsilon}\ \{op \mapsto \Lambda\eta.\ \lambda\_.\ (\lambda x.\ (\Lambda\eta_1.\ 1)\ \eta)\ \}^{\ell} \\ \quad (\Lambda\eta.\ \lambda^{\langle \ell^{\eta} \mid \epsilon\rangle}\ \_.\ \mathsf{perform}\ op\ \eta\ ()) \\ \longmapsto^{*} (\lambda x.\ (\Lambda\eta_1.\ 1)\ \eta) \quad \text{where } \eta \text{ fresh} \end{array}$$

Nevertheless, the scope variable η cannot be *used*, that is, no expression can return values parameterized by η, or perform operations in η, since it is impossible to perform an operation without leaking its scope into the effect type. The above example type-checks as it eliminates the scope variable via an unused type application.

C.3 Type Soundness

We can establish the type soundness of System $\mathsf{F}^{\epsilon+s}$ by proving preservation and progress.

Theorem C.1. (*Preservation of System* $\mathsf{F}^{\epsilon+s}$)
If $\varnothing \vdash e_1 : \sigma \mid \langle\rangle$ and $e_1 \longmapsto e_2$, then $\varnothing \vdash e_2 : \sigma \mid \langle\rangle$.

Theorem C.2. (*Progress of System* $\mathsf{F}^{\epsilon+s}$)
If $\varnothing \vdash e_1 : \sigma \mid \langle\rangle$ then either e_1 is a value, or $e_1 \longmapsto e_2$.

D MASKING EFFECTS

Masking is a natural extension of our system. It is also called *inject* [Leijen 2016] or *lift* [Biernacki et al. 2017] in the literature, but we prefer *mask* [Convent et al. 2020] as it conveys an operational meaning that one makes a specific handler invisible to an operation. In contrast, *inject* and *lift* take a type-theoretic view, reflecting the fact that the effect type is extended in the conclusion of the typing rule.

Since named handlers provide an elegant alternative to masking, here we consider masking mainly for unnamed handlers (Section 2.1), including scoped effects with unnamed handlers (System $\mathsf{F}^{\epsilon+s}$ and umbrella effects in System $\mathsf{F}^{\epsilon+u}$).

$$\begin{array}{llll} \text{Expression} & e & ::= & \ldots \mid \mathsf{mask}^l\ e \quad \text{(effect masking)} \\ \text{Evaluation Context} & \mathsf{E} & ::= & \ldots \mid \mathsf{mask}^l\ \mathsf{E} \end{array}$$

$$\frac{\Gamma \vdash e : \sigma \mid \epsilon}{\Gamma \vdash \mathsf{mask}^l\ e : \sigma \mid \langle l \mid \epsilon \rangle}\ [\textsc{mask}]$$

$$\begin{array}{llcl} (\textit{mask}) & \mathsf{mask}^l\ v & \longrightarrow & v \\ (\textit{perform}) & \mathsf{handle}^\epsilon\ h^l \cdot \mathsf{E} \cdot \mathsf{perform}\ op\ \overline{\sigma}\ v & \longrightarrow & f\ [\overline{\sigma}]\ v\ k \\ & \quad\text{iff } 0\text{-}\mathsf{free}^l(\mathsf{E})\ \wedge\ (op \mapsto f)\ \in h & & \end{array}$$

$$\frac{}{0\text{-}\mathsf{free}^l(\square)} \qquad \frac{n\text{-}\mathsf{free}^l(\mathsf{E})}{n\text{-}\mathsf{free}^l(v\ \mathsf{E})} \qquad \frac{n\text{-}\mathsf{free}^l(\mathsf{E})}{n\text{-}\mathsf{free}^l(\mathsf{E}\ e)}$$

$$\frac{(n{+}1)\text{-}\mathsf{free}(\mathsf{E})}{n\text{-}\mathsf{free}^l(\mathsf{handle}^l \cdot \mathsf{E})} \qquad \frac{n\text{-}\mathsf{free}(\mathsf{E}) \quad \text{iff } l \neq l'}{n\text{-}\mathsf{free}^l(\mathsf{handle}^{l'} \cdot \mathsf{E})}$$

$$\frac{n\text{-}\mathsf{free}^l(\mathsf{E})}{(n{+}1)\text{-}\mathsf{free}^l(\mathsf{mask}^l \cdot \mathsf{E})} \qquad \frac{n\text{-}\mathsf{free}^l(\mathsf{E}) \quad l \neq l'}{n\text{-}\mathsf{free}^l(\mathsf{mask}^{l'} \cdot \mathsf{E})}$$

Fig. 13. Effect Masking

Figure 13 defines the type and operational rules for mask (which can be extended straightforwardly to scoped effects). The extension is fairly standard. The (*mask*) rule is an identity, but the rules for (*perform*) searches for the innermost hanlder frame for the corresponding effect using the notion of *n-free* contexts [Biernacki et al. 2017], instead of bop. The notation n-free^l(E) means that an operation of label l would only be handled by $n + 1$-th handle frame of label l outside E. Note that *mask* causes its innermost handler to be ignored. So in rule PERFORM, we require 0-free^l(E), which says that an operation of label l would be handled by the first handle frame of label l outside E.

E TYPE-THEORETICALLY SOUND UMBRELLA EFFECTS

This section introduces restricted System $\mathsf{F}^{\epsilon+u}$ (Section 4.3), which statically ensures that names scoped under umbrella effects cannot escape for general umbrella effects.

E.1 Typing Rules

Figure 14a shows the rules of the scoped umbrella effects. Rules U-PERFORM, U-HANDLER and U-HANDLE are basically the same as the corresponding rules in System $\mathsf{F}^{\epsilon+u}$, but indirectly uses a new judgment $\vdash^l_{\text{ops}}$ in rule U-OPS. This judgment relies on two new concepts: the *resume effect* r^η, and the *umbrella witness* umb $\eta\ \langle r^\eta \mid \epsilon\rangle\ \sigma$, highlighted in gray.

The resume effect r^η is assigned to the resumption argument k. As it is polymorphic in η, it effectively prevents k from escaping the scope of the operation clause through either ϵ or σ; it must be used directly in the operation clause. This guarantee is needed for type soundness.

The umbrella witness umb $\eta\ \langle r^\eta \mid \epsilon\rangle\ \sigma$ is introduced as an alternative to scoped effects. Since we are using a specific η of the umbrella effect, we cannot prevent name escaping by abstracting over η as in System $\mathsf{F}^{\epsilon+\mathsf{s}}$ and System $\mathsf{F}^{\epsilon+\mathsf{sn}}$. With the umbrella witness, we can guarantee $\eta \notin \mathrm{ftv}(\epsilon, \sigma)$, and rule out programs such as (handler h^ℓ umb^η $(\lambda ev.\ ev)$).

$$\frac{op : \forall \overline{\alpha}^{\overline{\kappa}}.\ \sigma_1 \rightarrow l^{\eta}\ \sigma_2 \ \in \Sigma(l) \qquad \eta \in \overline{\alpha}^{\overline{\kappa}} \qquad \vdash_{\mathsf{wf}} \overline{\sigma} : \overline{\kappa}}{\Gamma \vdash_{\mathsf{val}} \mathsf{perform}^{\epsilon}\ op\ \overline{\sigma} : (\sigma_1 \rightarrow \langle l^{\eta} \mid \epsilon \rangle\ \sigma_2)[\overline{\alpha} := \overline{\sigma}]}\ [\textsc{u-perform}]$$

$$\frac{\Gamma \vdash^{l}_{\mathsf{ops}}\ h : \sigma \mid l \mid \epsilon}{\Gamma \vdash_{\mathsf{val}} \mathsf{handler}^{\epsilon}\ h^{l} : (\forall \eta.\ () \rightarrow \langle l^{\eta} \mid \epsilon \rangle\ \sigma) \rightarrow \epsilon\ \sigma}\ [\textsc{u-handler}]$$

$$\frac{\Gamma \vdash^{l}_{\mathsf{ops}}\ h : \sigma \mid l \mid \epsilon \qquad \Gamma \vdash e : \sigma \mid \langle l^{\eta} \mid \epsilon \rangle}{\Gamma \vdash \mathsf{handle}^{\epsilon}\ h^{l^{\eta}}\ e : \sigma \mid \epsilon}\ [\textsc{u-handle}]$$

$$\frac{\begin{array}{l} op_i : \forall \overline{\alpha}_i.\ \sigma_1 \rightarrow l^{\eta}\ \sigma_2 \ \in \Sigma(l) \\ \Gamma \vdash_{\mathsf{val}} f_i : \forall \overline{\alpha}_i.\ \mathsf{umb}\ \eta\ \langle r^{\eta} \mid \epsilon \rangle\ \sigma \rightarrow \sigma_1 \rightarrow \langle r^{\eta} \mid \epsilon \rangle\ (\sigma_2 \rightarrow \langle r^{\eta} \mid \epsilon \rangle\ \sigma) \rightarrow \langle r^{\eta} \mid \epsilon \rangle\ \sigma \end{array}}{\Gamma \vdash^{l}_{\mathsf{ops}} \{\ op_1 \rightarrow f_1, \ldots, op_n \rightarrow f_n\ \} : \sigma \mid l \mid \epsilon}\ [\textsc{u-ops}]$$

(a) Typing: scoped effects with static scoped resumptions

$$\frac{op : \forall \overline{\alpha}^{\overline{\kappa}}.\ \sigma_1 \rightarrow l^{\eta}\ \sigma_2 \ \in \Sigma(\ell) \qquad \eta \in \overline{\alpha}^{\overline{\kappa}} \qquad \vdash_{\mathsf{wf}} \overline{\sigma} : \overline{\kappa}}{\Gamma \vdash_{\mathsf{val}} \mathsf{perform}^{\epsilon}\ op\ \overline{\sigma} : (\mathsf{ev}\ \ell^{\eta} \rightarrow \sigma_1 \rightarrow \langle l^{\eta} \mid \epsilon \rangle\ \sigma_2)[\overline{\alpha} := \overline{\sigma}]}\ [\textsc{n-perform}]$$

$$\frac{\Gamma \vdash^{\ell}_{\mathsf{ops}}\ h : \sigma \mid \ell \mid \epsilon}{\Gamma \vdash_{\mathsf{val}} \mathsf{handler}^{\epsilon}\ h^{\ell} : \mathsf{umb}\ \eta\ \epsilon\ \sigma \rightarrow (\mathsf{ev}\ \ell^{\eta} \rightarrow \epsilon\ \sigma) \rightarrow \epsilon\ \sigma}\ [\textsc{n-handler}]$$

$$\frac{\Gamma \vdash^{\ell}_{\mathsf{ops}}\ h : \sigma \mid \ell \mid \epsilon \qquad \Gamma \vdash e : \sigma \mid \epsilon}{\Gamma \vdash \mathsf{handle}^{\epsilon}_{m}\ h^{\ell^{\eta}}\ e : \sigma \mid \epsilon}\ [\textsc{n-handle}]$$

$$\frac{\begin{array}{l} op_i : \forall \overline{\alpha}_i.\ \sigma_1 \rightarrow l^{\eta}\ \sigma_2 \ \in \Sigma(\ell) \\ \Gamma \vdash_{\mathsf{val}} f_i : \forall \overline{\alpha}_i.\ \langle r^{\eta} \mid \epsilon \rangle\ (\sigma_2 \rightarrow \langle r^{\eta} \mid \epsilon \rangle\ \sigma) \rightarrow \langle r^{\eta} \mid \epsilon \rangle\ \sigma \end{array}}{\Gamma \vdash^{\ell}_{\mathsf{ops}} \{\ op_1 \rightarrow f_1, \ldots, op_n \rightarrow f_n\ \} : \sigma \mid \ell \mid \epsilon}\ [\textsc{n-ops}]$$

(b) Typing: named handlers under scoped effects

Fig. 14. System $\mathsf{F}^{\epsilon+u}$

Figure 14b defines the typing rules for named handlers. The rules are the similar to the corresponding rules in System $\mathsf{F}^{\epsilon+u}$ except for rule N-HANDLER, where handler requires an umbrella witness. Again we must prevent any resumption of a named handler from escaping, as it may unwind through other named handlers above it. We use the judgment $\vdash^{\ell}_{\mathsf{ops}}$, which is similar to $\vdash^{l}_{\mathsf{ops}}$ with the resume effect, but without the umbrella witness.

E.2 Operational Semantics

Now we turn to the operational semantics (Figure 15). Note that the gray parts are specific to restricted System $\mathsf{F}^{\epsilon+u}$.

Rule (*u-perform*) and rule (*n-perform*) define the behavior of the resume effect. Given the original resumption k, the new resumption k' uses $\mathsf{mask}^{r^{\eta}}$ (Section D) to add the resume effect. Evaluation of f then has r^{η} in its return effect. Therefore, we enclose it by handle with an empty handler to eliminate the resume effect. Moreover, rule (*u-perform*) generates an umbrella witness umb^{η}, and passes it as the first argument to the operation implementation f. This witness is used by rule

$$
\begin{array}{llcl}
(\textit{u-handler}) & (\mathsf{handler}^{\epsilon}\ h^{l})\ v & \longrightarrow & \mathsf{handle}^{\epsilon}\ h^{l^{\eta}} \cdot v\ [\eta]\ () \qquad \text{where } \eta \text{ fresh} \\
(\textit{u-return}) & \mathsf{handle}^{\epsilon}\ h^{l^{\eta}} \cdot v & \longrightarrow & v \\
(\textit{u-perform}) & \multicolumn{3}{l}{\mathsf{handle}^{\epsilon}\ h^{l^{\eta}} \cdot \mathsf{E} \cdot \mathsf{perform}\ op\ \overline{\sigma}\ v} \\
 & & \longrightarrow & \mathsf{handle}^{\epsilon}\ \{\ \}^{r^{\eta}}\ (f\ [\overline{\sigma}]\ \mathit{umb}^{\eta}\ v\ k') \qquad \text{iff } 0\text{-free}^{l}(\mathsf{E}) \ \wedge\ (op \mapsto f)\ \in h \\
 & & & \text{where} \quad op\ :\ \forall \overline{\alpha}.\ \sigma_1 \rightarrow l^{\eta}\ \sigma_2\ \in \Sigma(l) \\
 & & & \qquad\qquad k\ =\ \lambda^{\epsilon} x : \sigma_2[\overline{\alpha} := \overline{\sigma}].\ \mathsf{handle}^{\epsilon}\ h \cdot \mathsf{E} \cdot x \\
 & & & \qquad\qquad k'\ =\ \lambda^{\langle r^{\eta} \mid \epsilon \rangle} x : \sigma_2[\overline{\alpha} := \overline{\sigma}].\ \mathsf{mask}^{r^{\eta}}\ k\ x
\end{array}
$$

(a) Scoped effects with static scoped resumptions

$$
\begin{array}{llcl}
(\textit{n-handler}) & (\mathsf{handler}^{\epsilon}\ h^{\ell})\ \mathit{umb}^{\eta}\ v & \longrightarrow & \mathsf{handle}^{\epsilon}_{m}\ h^{\ell^{\eta}} \cdot v\ (m, h^{\ell^{\eta}}) \qquad \text{where } m \text{ fresh} \\
(\textit{n-nreturn}) & \mathsf{handle}^{\epsilon}_{m}\ h^{\ell^{\eta}} \cdot v & \longrightarrow & v \\
(\textit{n-perform}) & \multicolumn{3}{l}{\mathsf{handle}^{\epsilon}_{m}\ h^{\ell^{\eta}} \cdot \mathsf{E} \cdot \mathsf{perform}\ op\ \overline{\sigma}\ (m, h)\ v} \\
 & & \longrightarrow & \mathsf{handle}^{\epsilon}\ \{\ \}^{r^{\eta}}\ (f[\overline{\sigma}]\ v\ k) \qquad \text{iff } (op \mapsto f)\ \in h \\
 & & & \text{where} \quad op\ :\ \forall \overline{\alpha}.\ \sigma_1 \rightarrow l^{\eta}\ \sigma_2\ \in \Sigma(\ell) \\
 & & & \qquad\qquad k\ =\ \lambda^{\epsilon} x : \sigma_2[\overline{\alpha} := \overline{\sigma}].\ \mathsf{handle}^{\epsilon}_{m}\ h \cdot \mathsf{E} \cdot x \\
 & & & \qquad\qquad k'\ =\ \lambda^{\langle r^{\eta} \mid \epsilon \rangle} x : \sigma_2[\overline{\alpha} := \overline{\sigma}].\ \mathsf{mask}^{r^{\eta}}\ k\ x
\end{array}
$$

(b) Named handlers under scoped effects

Fig. 15. System $\mathsf{F}^{\epsilon+u}$: operational semantics

($\textit{n-handler}$), which associates the label ℓ of the handler with the scope η of the witness. Note that, just like scopes, all umbrella witnesses can be replaced by an arbitrary value (e.g., unit) after type checking.

E.3 Type Soundness

Preservation of restricted System $\mathsf{F}^{\epsilon+u}$ is similar to that of System $\mathsf{F}^{\epsilon+u}$.

Theorem E.1. (*Preservation of Restricted System* $\mathsf{F}^{\epsilon+u}$)
If $\varnothing\ \vdash\ e_1\ :\ \sigma \mid \langle\rangle$ and $e_1 \longmapsto e_2$, then $\varnothing\ \vdash\ e_2\ :\ \sigma \mid \langle\rangle$.

Proving progress is much more challenging. We have seen that System $\mathsf{F}^{\epsilon+u}$ is more than simply putting System $\mathsf{F}^{\epsilon+s}$ and $\mathsf{F}^{\epsilon+n}$ together. In particular, the rules in this system incorporate umbrella witnesses and resume effects. The reason why we have these two features is that they provide *one* way to ensure well-scopedness of dynamically created names. Specifically, for dynamically created names to be well-scoped, we must ensure that evidence cannot leak into the handler's return type, and umbrella handlers cannot let evidence escape (e.g., *bad* operation). We have found that these two features are sufficient to prove progress for handle-safe System $\mathsf{F}^{\epsilon+u}$; any other possible formalizations must include features of similar forms. We give a detailed proof in Section F.7.

The statement of progress of handle-safe restricted System $\mathsf{F}^{\epsilon+u}$ has been stated as Theorem 4.2 in the paper.

F PROOFS

The proofs of all calculi are based on the proofs for System F^{ϵ} in [Xie et al. 2020b, Section C.1], as many syntactic constructs such as variables and applications are the same as that system. Thus, for extensions we often only discuss the new cases, i.e., cases related to the new algebraic effect constructs.

$$\boxed{\Gamma \vdash_{ec} \mathsf{E} : \sigma \rightarrow \sigma' \mid \epsilon}$$

$$\frac{}{\Gamma \vdash_{ec} \square : \sigma \rightarrow \sigma \mid \epsilon}\ [\textsc{cempty}]$$

$$\frac{\Gamma \vdash e : \sigma_2 \mid \epsilon \qquad \Gamma \vdash_{ec} \mathsf{E} : \sigma_1 \rightarrow (\sigma_2 \rightarrow \epsilon\ \sigma_3) \mid \epsilon}{\Gamma \vdash_{ec} \mathsf{E}\ e : \sigma_1 \rightarrow \sigma_3 \mid \epsilon}\ [\textsc{capp1}]$$

$$\frac{\Gamma \vdash_{val} v : \sigma_2 \rightarrow \epsilon\ \sigma_3 \qquad \Gamma \vdash_{ec} \mathsf{E} : \sigma_1 \rightarrow \sigma_2 \mid \epsilon}{\Gamma \vdash_{ec} v\ \mathsf{E} : \sigma_1 \rightarrow \sigma_3 \mid \epsilon}\ [\textsc{capp2}]$$

$$\frac{\Gamma \vdash_{ec} \mathsf{E} : \sigma_1 \rightarrow \forall \alpha.\ \sigma_2 \mid \epsilon}{\Gamma \vdash_{ec} \mathsf{E}\ [\sigma] : \sigma_1 \rightarrow \sigma_2\ [\alpha := \sigma] \mid \epsilon}\ [\textsc{ctapp}]$$

$$\frac{\Gamma \vdash_{ops} h^{\ell} : \sigma \mid \ell \mid \epsilon \qquad \Gamma \vdash_{ec} \mathsf{E} : \sigma_1 \rightarrow \sigma \mid \langle \ell \mid \epsilon \rangle}{\Gamma \vdash_{ec} \mathsf{handle}^{\epsilon}\ h^{\ell}\ \mathsf{E} : \sigma_1 \rightarrow \sigma \mid \epsilon}\ [\textsc{chandle}]$$

Fig. 16. Evaluation context typing in System F^{ϵ}

F.1 Evaluation Context Typing

In this section, we discuss the typing rules for evaluation contexts in System F^{ϵ}, while the typing rules for evaluation contexts in other systems are straightforward extensions. Lemmas from F^{ϵ} regarding evaluation context typing [Xie et al. 2020b, Section C.1.2] can be easily generalized to all extensions, since all evaluation contexts share similar definitions.

F.1.1 Evaluation Context Typing. Figure 16 presents the evaluation context typing in System F^{ϵ}. The corresponding rules for all other systems are straightforward extensions of this system, with slightly different CHANDLE.

In System $\mathsf{F}^{\epsilon+\mathsf{n}}$

$$\frac{\Gamma \vdash_{ops} h : \sigma \mid \ell \mid \epsilon \qquad \Gamma \vdash_{ec} \mathsf{E} : \sigma_1 \rightarrow \sigma \mid \langle \ell \mid \epsilon \rangle}{\Gamma \vdash_{ec} \mathsf{handle}^{\epsilon}_{m}\ h^{\ell}\ \mathsf{E} : \sigma_1 \rightarrow \sigma \mid \epsilon}\ [\textsc{chandle}]$$

In System $\mathsf{F}^{\epsilon+\mathsf{s}}$

$$\frac{\Gamma \vdash_{ops} h : \sigma \mid \ell \mid \epsilon \qquad \Gamma \vdash_{ec} \mathsf{E} : \sigma_1 \rightarrow \sigma \mid \langle \ell^{\eta} \mid \epsilon \rangle}{\Gamma \vdash_{ec} \mathsf{handle}^{\epsilon}\ h^{\ell^{\eta}}\ \mathsf{E} : \sigma_1 \rightarrow \sigma \mid \epsilon}\ [\textsc{chandle}]$$

In System $\mathsf{F}^{\epsilon+\mathsf{sn}}$

$$\frac{\Gamma \vdash_{ops} h : \sigma \mid \ell \mid \epsilon \qquad \Gamma \vdash_{ec} \mathsf{E} : \sigma_1 \rightarrow \sigma \mid \langle \ell^{\eta} \mid \epsilon \rangle}{\Gamma \vdash_{ec} \mathsf{handle}^{\epsilon}_{m}\ h^{\ell^{\eta}}\ \mathsf{E} : \sigma_1 \rightarrow \sigma \mid \epsilon}\ [\textsc{chandle}]$$

In System $\mathsf{F}^{\epsilon+u}$

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathsf{ops}} h : \sigma \mid l \mid \epsilon \\ \Gamma \vdash_{\mathsf{ec}} \mathsf{E} : \sigma_1 \rightarrow \sigma \mid \langle l^{\eta} \mid \epsilon \rangle\end{array}}{\Gamma \vdash_{\mathsf{ec}} \mathsf{handle}^{\epsilon}\ h^{l^{\eta}}\ \mathsf{E} : \sigma_1 \rightarrow \sigma \mid \epsilon}\ [\textsc{chandle-scoped}]$$

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathsf{ops}} h : \sigma \mid \ell \mid \epsilon \\ \Gamma \vdash_{\mathsf{ec}} \mathsf{E} : \sigma_1 \rightarrow \sigma \mid \epsilon\end{array}}{\Gamma \vdash_{\mathsf{ec}} \mathsf{handle}^{\epsilon}_{m}\ h^{\ell^{\eta}}\ \mathsf{E} : \sigma_1 \rightarrow \sigma \mid \epsilon}\ [\textsc{chandle-named}]$$

F.1.2 Notations.

Definition F.1. (*Extraction of Labels*)
We define the extraction of labels as $\lceil \mathsf{E} \rceil^{\ell}(\epsilon)$. When given a context E whose effect is ϵ, the extraction extends ϵ with labels handled by the handlers in E. Note that the ℓ in the notation does not represent any specific label.

$$\begin{array}{lcl}\lceil \square \rceil^{\ell}(\epsilon) & = & \epsilon \\ \lceil \mathsf{E}\ e \rceil^{\ell}(\epsilon) & = & \lceil \mathsf{E} \rceil^{\ell}(\epsilon) \\ \lceil v\ \mathsf{E} \rceil^{\ell}(\epsilon) & = & \lceil \mathsf{E} \rceil^{\ell}(\epsilon) \\ \lceil \mathsf{handle}^{\epsilon}\ h^{\ell}\ \mathsf{E} \rceil^{\ell}(\epsilon) & = & \lceil \mathsf{E} \rceil^{\ell}(\langle \ell \mid \epsilon \rangle)\end{array}$$

In System $\mathsf{F}^{\epsilon+\mathsf{s}}$

$$\lceil \mathsf{handle}^{\epsilon}\ h^{\ell^{\eta}}\ \mathsf{E} \rceil^{\ell}(\epsilon) = \lceil \mathsf{E} \rceil^{\ell}(\langle \ell^{\eta} \mid \epsilon \rangle)$$

In System $\mathsf{F}^{\epsilon+\mathsf{n}}$

$$\lceil \mathsf{handle}^{\epsilon}\ h^{\ell}\ \mathsf{E} \rceil^{\ell}(\epsilon) = \lceil \mathsf{E} \rceil^{\ell}(\langle \ell \mid \epsilon \rangle)$$

In System $\mathsf{F}^{\epsilon+\mathsf{sn}}$

$$\lceil \mathsf{handle}^{\epsilon}\ h^{\ell^{\eta}}\ \mathsf{E} \rceil^{\ell}(\epsilon) = \lceil \mathsf{E} \rceil^{\ell}(\langle \ell^{\eta} \mid \epsilon \rangle)$$

In System $\mathsf{F}^{\epsilon+u}$

$$\begin{array}{lcl}\lceil \mathsf{handle}^{\epsilon}\ h^{\ell^{\eta}}\ \mathsf{E} \rceil^{\ell}(\epsilon) & = & \lceil \mathsf{E} \rceil^{\ell}(\langle \ell^{\eta} \mid \epsilon \rangle) \\ \lceil \mathsf{handle}^{\epsilon}\ h^{\ell}\ \mathsf{E} \rceil^{\ell}(\epsilon) & = & \lceil \mathsf{E} \rceil^{\ell}(\epsilon)\end{array}$$

Definition F.2. (*Dot Notation*)
For conciseness, we often use the *dot notation* to compose and decompose evaluation contexts.

$$\begin{array}{lclcl}\mathsf{E} \cdot e & \triangleq & \mathsf{E}[e] & & \\ \square\ e \cdot \mathsf{E} & \triangleq & \mathsf{E}\ e & & \\ v\ \square \cdot \mathsf{E} & \triangleq & v \cdot \mathsf{E} & \triangleq & v\ \mathsf{E} \\ \mathsf{handle}\ h\ \square \cdot \mathsf{E} & \triangleq & \mathsf{handle}\ h \cdot \mathsf{E} & \triangleq & \mathsf{handle}\ h\ \mathsf{E}\end{array}$$

F.1.3 Lemmas.

Lemma F.3. (*Evaluation context typing*)
If $\varnothing \vdash_{\mathsf{ec}} \mathsf{E} : \sigma_1 \rightarrow \sigma_2 \mid \epsilon$ and $\varnothing \vdash e : \sigma_1 \mid \langle \lceil \mathsf{E} \rceil^{\ell}(\epsilon) \rangle$,
then $\varnothing \vdash \mathsf{E}[e] : \sigma_2 \mid \epsilon$.

Lemma F.4. (*Effect corresponds to the evaluation context*)
If $\varnothing \vdash \mathsf{E}[e] : \sigma \mid \epsilon$, then there exists σ_1 such that
$\varnothing \vdash_{\mathsf{ec}} \mathsf{E} : \sigma_1 \rightarrow \sigma \mid \epsilon$, and $\varnothing \vdash e : \sigma_1 \mid \langle \lceil \mathsf{E} \rceil^{\ell} \mid \epsilon \rangle$.

F.2 Values are Effect-free

The following lemma holds for all systems:

Lemma F.5. (*Values can have any effect*)
If $\Gamma \vdash v : \sigma \mid \epsilon_1$, then $\Gamma \vdash v : \sigma \mid \epsilon_2$.

Proof. (*of Lemma F.5*) Follows directly by VAL. □

F.3 Proofs for System $\mathsf{F}^{\epsilon+n}$

F.3.1 Preservation.

Lemma F.6. (*Small Step Preservation*)
If $\varnothing \vdash e_1 : \sigma \mid \epsilon$ and $e_1 \longrightarrow e_2$, then $\varnothing \vdash e_2 : \sigma \mid \epsilon$.

Proof. (*of Lemma F.6*) By induction on reduction. Here we show the cases involving reduction of named handlers.

case $(\mathsf{handler}^{\epsilon}\ h^{\ell})\ v \longrightarrow \mathsf{handle}^{\epsilon}\ h^{\ell}\ (v\ (m,\ h^{\ell}))$ with η fresh.

$\varnothing \vdash (\mathsf{handler}^{\epsilon}\ h)\ v : \sigma \mid \epsilon$	given
$\varnothing \vdash \mathsf{handler}^{\epsilon}\ h : (\mathsf{ev}\ \ell \to \langle \ell \mid \epsilon\rangle\ \sigma) \to \epsilon\ \sigma \mid \epsilon$	APP
$\varnothing \vdash v : \mathsf{ev}\ \ell \to \langle \ell \mid \epsilon\rangle\ \sigma \mid \epsilon$	above
$\varnothing \vdash_{\mathsf{val}} \mathsf{handler}^{\epsilon}\ h^{\ell} : (\mathsf{ev}\ \ell \to \langle \ell \mid \epsilon\rangle\ \sigma) \to \epsilon\ \sigma$	VAL
$\varnothing \vdash_{\mathsf{ops}} h : \sigma \mid \ell \mid \epsilon$	HANDLER
$\varnothing \vdash v : \mathsf{ev}\ \ell \to \langle \ell \mid \epsilon\rangle\ \sigma \mid \langle \ell \mid \epsilon\rangle$	Lemma F.5
$\varnothing \vdash v\ (m,\ h^{\ell}) : \sigma \mid \langle \ell \mid \epsilon\rangle$	APP
$\varnothing \vdash \mathsf{handle}^{\epsilon}_{m}\ h^{\ell}\ (v\ (m,\ h^{\ell})) : \sigma \mid \epsilon$	HANDLE

case $\mathsf{handle}^{\epsilon}_{m}\ h \cdot \mathsf{E} \cdot \mathsf{perform}\ op\ \overline{\sigma}\ (m,\ h)\ v \longrightarrow f\ [\overline{\sigma}]\ v\ k.$

$op \mapsto f\ \in h$	given
$op : \forall \overline{\alpha}.\ \sigma_1 \to \ell\ \sigma_2\ \in \Sigma(\ell)$	given
$k = \lambda^{\epsilon}\ x : \sigma_2[\overline{\alpha} := \overline{\sigma}].\ \mathsf{handle}^{\epsilon}\ h \cdot \mathsf{E} \cdot x$	given
$\varnothing \vdash \mathsf{handle}^{\epsilon}\ h \cdot \mathsf{E} \cdot \mathsf{perform}\ op\ \overline{\sigma}\ (m,\ h)\ v : \sigma \mid \epsilon$	given
$\varnothing \vdash_{\mathsf{ops}} h : \sigma \mid \ell \mid \epsilon$	rule HANDLE
$\varnothing \vdash_{\mathsf{val}} f : \forall \overline{\alpha}.\ \sigma_1 \to \epsilon\ (\sigma_2 \to \epsilon\ \sigma) \to \epsilon\ \sigma$	OPS
$\varnothing \vdash f : \forall \overline{\alpha}.\ \sigma_1 \to \epsilon\ (\sigma_2 \to \epsilon\ \sigma) \to \epsilon\ \sigma \mid \epsilon$	VAL
$\varnothing \vdash f\ [\overline{\sigma}] : \sigma_1[\overline{\alpha} := \overline{\sigma}] \to \epsilon\ (\sigma_2[\overline{\alpha} := \overline{\sigma}] \to \epsilon\ \sigma) \to \epsilon\ \sigma \mid \epsilon$	TAPP
$\varnothing \vdash \mathsf{perform}\ op\ \overline{\sigma}\ (m,\ h)\ v : \sigma_2[\overline{\alpha} := \overline{\sigma}] \mid \lceil \mathsf{handle}^{\epsilon}\ h\ \mathsf{E} \rceil^{\ell}(\epsilon)$	Lemma F.4
$\varnothing \vdash_{\mathsf{ec}} \mathsf{handle}^{\epsilon}\ h \cdot \mathsf{E} : \sigma_2[\overline{\alpha} := \overline{\sigma}] \to \sigma \mid \epsilon$	above
$\varnothing \vdash v : \sigma_1[\overline{\alpha} := \overline{\sigma}] \mid \lceil \mathsf{handle}^{\epsilon}\ h\ \mathsf{E} \rceil^{\ell}(\epsilon)$	APP and TAPP
$\varnothing \vdash v : \sigma_1[\overline{\alpha} := \overline{\sigma}] \mid \epsilon$	Lemma F.5
$\varnothing \vdash f\ [\overline{\sigma}]\ v : (\sigma_2[\overline{\alpha} := \overline{\sigma}] \to \epsilon\ \sigma) \to \epsilon\ \sigma \mid \epsilon$	APP
$x : \sigma_2[\overline{\alpha} := \overline{\sigma}] \vdash_{\mathsf{val}} x : \sigma_2[\overline{\alpha} := \overline{\sigma}]$	VAR
$x : \sigma_2[\overline{\alpha} := \overline{\sigma}] \vdash x : \sigma_2[\overline{\alpha} := \overline{\sigma}] \mid \epsilon$	VAL
$x : \sigma_2[\overline{\alpha} := \overline{\sigma}] \vdash_{\mathsf{ec}} \mathsf{handle}^{\epsilon} \cdot \mathsf{E} : \sigma_2[\overline{\alpha} := \overline{\sigma}] \to \sigma \mid \epsilon$	weakening
$x : \sigma_2[\overline{\alpha} := \overline{\sigma}] \vdash \mathsf{handle}^{\epsilon}\ h \cdot \mathsf{E} \cdot x : \sigma \mid \epsilon$	Lemma F.3
$\varnothing \vdash_{\mathsf{val}} \lambda^{\epsilon}\ x : \sigma_2[\overline{\alpha} := \overline{\sigma}].\ \mathsf{handle}^{\epsilon}\ h \cdot \mathsf{E} \cdot x : \sigma_2[\overline{\alpha} := \overline{\sigma}] \to \epsilon\ \sigma$	ABS
$\varnothing \vdash \lambda^{\epsilon}\ x : \sigma_2[\overline{\alpha} := \overline{\sigma}].\ \mathsf{handle}^{\epsilon}\ h \cdot \mathsf{E} \cdot x : \sigma_2[\overline{\alpha} := \overline{\sigma}] \to \epsilon\ \sigma \mid \epsilon$	
$\varnothing \vdash f\ [\overline{\sigma}]\ v\ k : \sigma \mid \epsilon$	APP

□

Proof. (*Of Theorem B.1*)

$e_1 \;=\; \mathsf{E}[e_1']$	(*step*)
$e_1' \longrightarrow e_2'$	above
$e_2 \;=\; \mathsf{E}[e_2']$	above
$\varnothing \;\vdash\; \mathsf{E}[e_1'] \;:\; \sigma \mid \langle\rangle$	given
$\varnothing \;\vdash\; e_1 \;:\; \sigma_1 \mid \lceil \mathsf{E} \rceil^{\ell}(\langle\rangle)$	Lemma F.4
$\varnothing \;\vdash\; \mathsf{E} \;:\; \sigma_1 \rightarrow \sigma \mid \langle\rangle$	above
$\varnothing \;\vdash\; e_2 \;:\; \sigma_1 \mid \lceil \mathsf{E} \rceil^{\ell}(\langle\rangle)$	Lemma F.6
$\varnothing \;\vdash\; \mathsf{E}[e_2] \;:\; \sigma \mid \langle\rangle$	Lemma F.3

□

F.3.2 Uniqueness of Names.

Proof. (*of Theorem B.2*) First, we observe that for handle-safe expressions, if

$$e \;=\; \mathsf{E}_1 \cdot \mathsf{handle}^{\epsilon_1}_{m_1}\; h^{\ell 1} \cdot \mathsf{E}_2 \cdot \mathsf{handle}^{\epsilon_2}_{m_2}\; h^{\ell 2} \cdot e_0$$

then we have $\epsilon_1 \;=\; \epsilon_2$.

We proceed by proving the goal by contradiction. Suppose we have $m_1 = m_2$. That means

$$e \;=\; \mathsf{E}_1 \cdot \mathsf{handle}^{\epsilon_1}_{m_1}\; h^{\ell 1} \cdot \mathsf{E}_2 \cdot \mathsf{handle}^{\epsilon_1}_{m_1}\; h^{\ell 2} \cdot e_0.$$

By Lemma F.4, we have

$$\varnothing \;\vdash\; \mathsf{handle}^{\epsilon_1}_{m_1}\; h^{\ell 2} \cdot e_0 : \lceil \mathsf{E}_1 \cdot \mathsf{handle}^{\epsilon_1}_{m_1}\; h^{\ell 1} \cdot \mathsf{E}_2 \rceil^{\ell}(\epsilon_1)$$

By typing rule HANDLE, we have

$$\epsilon_1 \;=\; \lceil \mathsf{E}_1 \cdot \mathsf{handle}^{\epsilon_1}_{m_1}\; h^{\ell 1} \cdot \mathsf{E}_2 \rceil^{\ell}(\epsilon_1)$$

However, according to Definition F.1,

$$\lceil \mathsf{E}_1 \cdot \mathsf{handle}^{\epsilon_1}_{m_1}\; h^{\ell 1} \cdot \mathsf{E}_2 \rceil^{\ell}(\epsilon_1)$$

has at least one more ℓ_1 than ϵ_1. Thus the contradiction. □

F.4 Proofs for System $\mathsf{F}^{\epsilon+s}$

F.4.1 Preservation.

Lemma F.7. (*Small Step Preservation*)
If $\varnothing \ \vdash \ e_1 \ : \sigma \mid \epsilon$ and $e_1 \longrightarrow e_2$, then $\varnothing \ \vdash \ e_2 \ : \sigma \mid \epsilon$.

Proof. (*of Lemma F.7*) By induction on reduction. Again, we discuss the reduction of handlers.
case $(\mathsf{handler}^{\epsilon}\ h^{\ell})\ v \longrightarrow \mathsf{handle}^{\epsilon}\ h^{\ell^{\eta}}\ (v\ [\eta]\ ())$ with η fresh.

$\varnothing \ \vdash \ (\mathsf{handler}^{\epsilon}\ h)\ v \ : \sigma \mid \epsilon$	given
$\varnothing \ \vdash \ \mathsf{handler}^{\epsilon}\ h \ : (\forall \eta.\ () \rightarrow \langle \ell^{\eta} \mid \epsilon \rangle\ \sigma) \rightarrow \epsilon\ \sigma \mid \epsilon$	APP
$\varnothing \ \vdash \ v \ : \forall \eta.\ () \rightarrow \langle \ell^{\eta} \mid \epsilon \rangle\ \sigma \mid \epsilon$	above
$\varnothing \ \vdash_{\mathsf{val}} \ \mathsf{handler}^{\epsilon}\ h^{\ell} \ : (\forall \eta.\ () \rightarrow \langle \ell^{\eta} \mid \epsilon \rangle\ \sigma) \rightarrow \epsilon\ \sigma$	VAL
$\varnothing \ \vdash_{\mathsf{ops}} \ h \ : \sigma \mid \ell \mid \epsilon$	HANDLER
$\varnothing \ \vdash \ v \ : \forall \eta.\ () \rightarrow \langle \ell^{\eta} \mid \epsilon \rangle\ \sigma \mid \langle \ell \mid \epsilon \rangle$	Lemma F.5
$\varnothing \ \vdash \ v\ [\eta]\ () \ : \sigma \mid \langle \ell^{\eta} \mid \epsilon \rangle$	TAPP and APP
$\varnothing \ \vdash \ \mathsf{handle}^{\epsilon}\ h^{\ell^{\eta}}\ (v\ [\eta]\ ()) \ : \sigma \mid \langle \epsilon \rangle$	HANDLE

□

Proof. (*of Theorem C.1*) Same as Theorem B.1 with Lemma F.7. □

F.4.2 Progress.

Lemma F.8. (*Progress with effects*)
If $\varnothing \ \vdash \ e_1 \ : \sigma \mid \epsilon$ then either e_1 is a value, or $e_1 \longmapsto e_2$, or $e_1 \ = \ \mathsf{E}[\mathsf{perform}\ op\ \overline{\sigma}\ v]$, where $op \ : \forall \overline{\alpha}.\ \sigma_1 \rightarrow \ell^{\eta}\ \sigma_2$
and $op \notin \mathsf{bop}(\mathsf{E})$.

Proof. (*of Lemma F.8*) By induction on typing. The proof structure is the same as the progress lemma for F^{ϵ}. Here we discuss cases specific to System $\mathsf{F}^{\epsilon+s}$.
case $e_1 \ = \ e_3\ e_4$ where both e_3 and e_4 are values. We do case analysis on the form of e_3.
subcase $e_3 \ = \ \mathsf{handler}^{\epsilon}\ h$. Then by (*handler*) and (*step*) we have $\mathsf{handler}^{\epsilon}\ h\ e_4 \longrightarrow \mathsf{handle}^{\epsilon}\ h\ (e_4\ [\eta]\ ())$ with η fresh.
case $e_1 \ = \ e_3\ [\sigma_1]$ where e_3 is a value. We do case analysis on the form of e_3.

$\varnothing \ \vdash \ e_3\ [\sigma_1] \ : \sigma_2[\alpha := \sigma_1] \ \mid \epsilon$	given
$\varnothing \ \vdash \ e_3 \ : \forall \alpha.\ \sigma_2 \mid \epsilon$	APP

subcase $e_3 \ = \ \mathsf{handler}^{\epsilon}\ h$. This is impossible because it does not have a polymorphic type. □

Proof. (*of Theorem C.2*) By applying Lemma F.8, we know that either e_1 is a value, or $e_1 \longmapsto e_2$, or $e_1 \ = \ \mathsf{E}[\mathsf{perform}\ op\ \overline{\sigma}\ v]$, where $op \ : \forall \overline{\alpha}.\ \sigma_1 \rightarrow \sigma_2 \ \in \Sigma(\ell)$ and $op \notin \mathsf{bop}(\mathsf{E})$. For the first two cases, we have proved the goal. For the last case, we prove the goal by contradiction.

$\varnothing \ \vdash \ \mathsf{E}[\mathsf{perform}\ op\ \overline{\sigma}\ v] \ : \sigma \mid \langle\rangle$	given
$\varnothing \ \vdash \ \mathsf{perform}\ op\ \overline{\sigma}\ v \ : \sigma_1 \mid \lceil \mathsf{E} \rceil^{\ell}(\langle\rangle)$	Lemma F.4
$l \ \in \lceil \mathsf{E} \rceil^{\ell}(\langle\rangle)$	PERFORM
$op \notin \mathsf{bop}(\mathsf{E}))$	given
$l \notin \lceil \mathsf{E} \rceil^{\ell}(\langle\rangle)$	Follows
Contradiction	

□

F.5 Proofs for System $\mathsf{F}^{\epsilon+\mathsf{sn}}$

F.5.1 Preservation.

Lemma F.9. (*Small Step Preservation*)
If $\varnothing \vdash e_1 : \sigma \mid \epsilon$ and $e_1 \longrightarrow e_2$, then $\varnothing \vdash e_2 : \sigma \mid \epsilon$.

Proof. (*of Lemma F.9*) By induction on reduction. We detail the interesting cases.
case $(\mathsf{handler}^{\epsilon}\ h^{\ell})\ v \longrightarrow \mathsf{handle}^{\epsilon}_{m}\ h^{\ell^{\eta}}\ (v\ [\eta]\ (m,\ h^{\ell^{\eta}}))$ with $\eta,\ m$ fresh.

$\varnothing \vdash (\mathsf{handler}^{\epsilon}\ h)\ v : \sigma \mid \epsilon$	given
$\varnothing \vdash \mathsf{handler}^{\epsilon}\ h : (\forall\eta.\ \mathsf{ev}\ \ell^{\eta} \to \langle \ell^{\eta} \mid \epsilon\rangle\ \sigma) \to \epsilon\ \sigma \mid \epsilon$	APP
$\varnothing \vdash v : \forall\eta.\ \mathsf{ev}\ \ell^{\eta} \to \langle \ell^{\eta} \mid \epsilon\rangle\ \sigma \mid \epsilon$	above
$\varnothing \vdash_{\mathsf{val}} \mathsf{handler}^{\epsilon}\ h^{\ell} : (\forall\eta.\ \mathsf{ev}\ \ell^{\eta} \to \langle \ell^{\eta} \mid \epsilon\rangle\ \sigma) \to \epsilon\ \sigma$	VAL
$\varnothing \vdash_{\mathsf{ops}} h : \sigma \mid \ell \mid \epsilon$	HANDLER
$\varnothing \vdash v : \forall\eta.\ \mathsf{ev}\ \ell^{\eta} \to \langle \ell^{\eta} \mid \epsilon\rangle\ \sigma \mid \langle \ell^{\eta} \mid \epsilon\rangle$	Lemma F.5
$\varnothing \vdash v\ [\eta]\ (m,\ h) : \sigma \mid \langle \ell^{\eta} \mid \epsilon\rangle$	TAPP and APP
$\varnothing \vdash \mathsf{handle}^{\epsilon}_{m}\ h^{\ell^{\eta}}\ (v\ [\eta]\ (m,\ h)) : \sigma \mid \langle\epsilon\rangle$	HANDLE

case $\mathsf{handle}^{\epsilon}_{m}\ h \cdot \mathsf{E} \cdot \mathsf{perform}\ op\ \overline{\sigma}\ (m,\ h)\ v \longrightarrow f\ [\overline{\sigma}]\ v\ k$.

$op \mapsto f\ \in h$	given
$op : \forall\overline{\alpha}.\ \sigma_1 \to \ell^{\eta}\ \sigma_2\ \in \Sigma(\ell)$	given
$k = \lambda^{\epsilon}\ x : \sigma_2[\overline{\alpha} := \overline{\sigma}].\ \mathsf{handle}^{\epsilon}\ h^{\ell^{\eta}} \cdot \mathsf{E} \cdot x$	given
$\varnothing \vdash \mathsf{handle}^{\epsilon}\ h \cdot \mathsf{E} \cdot \mathsf{perform}\ op\ \overline{\sigma}\ (m,\ h)\ v : \sigma \mid \epsilon$	given
$\varnothing \vdash_{\mathsf{ops}} h : \sigma \mid \ell \mid \epsilon$	HANDLE
$\varnothing \vdash_{\mathsf{val}} f : \forall\overline{\alpha}.\ \sigma_1 \to \epsilon\ (\sigma_2 \to \epsilon\ \sigma) \to \epsilon\ \sigma$	OPS
$\varnothing \vdash f : \forall\overline{\alpha}.\ \sigma_1 \to \epsilon\ (\sigma_2 \to \epsilon\ \sigma) \to \epsilon\ \sigma \mid \epsilon$	VAL
$\varnothing \vdash f\ [\overline{\sigma}] : \sigma_1[\overline{\alpha} := \overline{\sigma}] \to \epsilon\ (\sigma_2[\overline{\alpha} := \overline{\sigma}] \to \epsilon\ \sigma) \to \epsilon\ \sigma \mid \epsilon$	TAPP
$\varnothing \vdash \mathsf{perform}\ op\ \overline{\sigma}\ v : \sigma_2[\overline{\alpha} := \overline{\sigma}] \mid \lceil \mathsf{handle}^{\epsilon}\ h\ \mathsf{E}\rceil^{\ell}(\epsilon)$	Lemma F.4
$\varnothing \vdash_{\mathsf{ec}} \mathsf{handle}^{\epsilon}\ h \cdot \mathsf{E} : \sigma_2[\overline{\alpha} := \overline{\sigma}] \to \sigma \mid \epsilon$	above
$\varnothing \vdash v : \sigma_1[\overline{\alpha} := \overline{\sigma}] \mid \lceil \mathsf{handle}^{\epsilon}\ h\ \mathsf{E}\rceil^{\ell}(\epsilon)$	APP and TAPP
$\varnothing \vdash v : \sigma_1[\overline{\alpha} := \overline{\sigma}] \mid \epsilon$	Lemma F.5
$\varnothing \vdash f\ [\overline{\sigma}]\ v : (\sigma_2[\overline{\alpha} := \overline{\sigma}] \to \epsilon\ \sigma) \to \epsilon\ \sigma \mid \epsilon$	APP
$x : \sigma_2[\overline{\alpha} := \overline{\sigma}] \vdash_{\mathsf{val}} x : \sigma_2[\overline{\alpha} := \overline{\sigma}]$	VAR
$x : \sigma_2[\overline{\alpha} := \overline{\sigma}] \vdash x : \sigma_2[\overline{\alpha} := \overline{\sigma}] \mid \epsilon$	VAL
$x : \sigma_2[\overline{\alpha} := \overline{\sigma}] \vdash_{\mathsf{ec}} \mathsf{handle}^{\epsilon} \cdot \mathsf{E} : \sigma_2[\overline{\alpha} := \overline{\sigma}] \to \sigma \mid \epsilon$	weakening
$x : \sigma_2[\overline{\alpha} := \overline{\sigma}] \vdash \mathsf{handle}^{\epsilon}\ h \cdot \mathsf{E} \cdot x : \sigma \mid \epsilon$	Lemma F.3
$\varnothing \vdash_{\mathsf{val}} \lambda^{\epsilon}\ x : \sigma_2[\overline{\alpha} := \overline{\sigma}].\ \mathsf{handle}^{\epsilon}\ h \cdot \mathsf{E} \cdot x : \sigma_2[\overline{\alpha} := \overline{\sigma}] \to \epsilon\ \sigma$	ABS
$\varnothing \vdash \lambda^{\epsilon}\ x : \sigma_2[\overline{\alpha} := \overline{\sigma}].\ \mathsf{handle}^{\epsilon}\ h \cdot \mathsf{E} \cdot x : \sigma_2[\overline{\alpha} := \overline{\sigma}] \to \epsilon\ \sigma \mid \epsilon$	VAL
$\varnothing \vdash f\ [\overline{\sigma}]\ v\ k : \sigma \mid \epsilon$	APP

□

Proof. (*of Theorem 3.1*) Same as Theorem B.1 with Lemma F.9. □

F.5.2 Progress.

Lemma F.10. (*Progress with effects*)
If $\varnothing \vdash e_1 : \sigma \mid \epsilon$ then either e_1 is a value, or $e_1 \longmapsto e_2$, or $e_1 = \mathsf{E}[\mathsf{perform}\ op\ \overline{\sigma}\ (m,\ h^{\ell^{\eta}})\ v]$, where $op : \forall\overline{\alpha}.\ \sigma_1 \to \ell^{\eta}\ \sigma_2\ \in \Sigma(\ell)$, and E has no $\mathsf{handle}_{m}\ h^{\ell^{\eta}}$.

Proof. (*of Lemma F.10*) By induction on typing. The proof structure is the same as the progress lemma for F^{ϵ}. Here we discuss cases specific to System $\mathsf{F}^{\epsilon+\mathsf{sn}}$.

case $e_1 = e_3\ e_4$ where both e_3 and e_4 are values. We do case analysis on the form of e_3.

subcase $e_3 = \mathsf{handler}^{\epsilon}\ h$. Then by (*handler*) and (*step*) we have
$\mathsf{handler}^{\epsilon}\ h\ e_4 \longrightarrow \mathsf{handle}^{\epsilon}\ h\ (e_4\ [\eta]\ (m,\ h))$ with $\eta,\ m$ fresh.

case $e_1 = e_3\ [\sigma_1]$ where e_3 is a value. We do case analysis on the form of e_3.

$\varnothing \vdash e_3\ [\sigma_1] : \sigma_2[\alpha := \sigma_1] \mid \epsilon$	given
$\varnothing \vdash e_3 : \forall\alpha.\ \sigma_2 \mid \epsilon$	APP

subcase $e_3 = \mathsf{handler}^{\epsilon}\ h$. This is impossible because it does not have a polymorphic type.

case $e_1 = \mathsf{handle}^{\epsilon}_m\ h^{\ell^{\eta}}\ e$.

$\varnothing \vdash \mathsf{handle}^{\epsilon}\ h^{\ell^{\eta}}\ e : \sigma \mid \epsilon$	given
$\varnothing \vdash e : \sigma \mid \langle \ell^{\eta} \mid \epsilon \rangle$	HANDLE

By I.H., we know that either e is a value, or $e \longmapsto e_3$, or $e = \mathsf{E}_0[\mathsf{perform}\ op\ \overline{\sigma}\ (m_1,\ h_1^{\ell 1^{\eta_1}})\ v]$.

- $e \longmapsto e_3$. Then we know $\mathsf{handle}^{\epsilon}\ h\ e \longmapsto \mathsf{handle}^{\epsilon}\ h\ e_3$ by STEP and the goal holds.
- $e = \mathsf{E}_0[\mathsf{perform}\ op\ \overline{\sigma}\ (m_1,\ h_1^{\ell^{\eta}})\ v]$, and E_0 has no $\mathsf{handle}_{m_1}\ h_1^{\ell^{\eta}}$. We discuss whether $\mathsf{handle}_m\ h^{\ell^{\eta}}$ handles this perform.
  - Handle. Then by (*perform*) and (*step*) we have
    $\mathsf{handle}^{\epsilon}_m\ h \cdot \mathsf{E}_0 \cdot \mathsf{perform}\ op\ \overline{\sigma}\ (m,\ h)\ v \longmapsto f\ \overline{\sigma}\ v\ k$.
  - Not handle. Let $\mathsf{E} = \mathsf{handle}^{\epsilon}_m\ h\ \mathsf{E}_0$, then we have
    $e_1 = \mathsf{E}[\mathsf{perform}\ op\ \overline{\sigma}\ (m_1,\ h_1^{\ell 1^{\eta_1}})\ v]$.
- e is a value. Then by (*return*) and (*step*) we have $\mathsf{handle}^{\epsilon}\ h\ e \longmapsto e$.

□

Proof. (*of Theorem 3.3*) We first observe that for handle-safe expressions, when (*handler*) generates m, each m is specific to some scope η and some label ℓ.

By applying Lemma F.10, we know that either e_1 is a value, or $e_1 \longmapsto e_2$, or
$e_1 = \mathsf{E}[\mathsf{perform}\ op\ \overline{\sigma}\ (m,\ h^{\ell^{\eta}})\ v]$, where $op : \forall\overline{\alpha}.\ \sigma_1 \to^{\ell^{\eta}} \sigma_2 \in \Sigma(\ell)$, and E has no $\mathsf{handle}_m\ h^{\ell^{\eta}}$. For the first two cases, we have proved the goal. For the last case, we prove the goal by contradiction.

$\varnothing \vdash \mathsf{E}[\mathsf{perform}\ op\ \overline{\sigma}\ (m,\ h^{\ell^{\eta}})\ v] : \sigma \mid \langle\rangle$	given
$\varnothing \vdash \mathsf{perform}\ op\ \overline{\sigma}\ (m,\ h^{\ell^{\eta}})\ v : \sigma_1 \mid \lceil \mathsf{E} \rceil^{\ell}(\langle\rangle)$	Lemma F.4
$\ell^{\eta} \in \lceil \mathsf{E} \rceil^{\ell}(\langle\rangle)$	PERFORM
E has no $\mathsf{handle}_m\ h^{\ell^{\eta}}$	Given

From rule (*handler*), we know that every η has a one-to-one correspondence. Therefore, E cannot have $\mathsf{handle}_{m_1}\ h^{\ell^{\eta}}$ for some other m_1.

Now given E has no $\mathsf{handle}_m\ h^{\ell^{\eta}}$, then $\ell^{\eta} \in \lceil \mathsf{E} \rceil^{\ell}(\langle\rangle)$ is impossible. Thus we have a contradiction.

□

F.5.3 Uniqueness of Names.

Proof. (*of Theorem 3.4*) The same as Theorem B.2. □

F.6 Proofs for System $\mathsf{F}^{\epsilon+u}$

F.6.1 Preservation.

Lemma F.11. (*Small Step Preservation*)
If $\varnothing \vdash e_1 : \sigma \mid \epsilon$ and $e_1 \longrightarrow e_2$, then $\varnothing \vdash e_2 : \sigma \mid \epsilon$.

Proof. (*of Lemma F.11*) By induction on reduction. As before, we focus on the cases involving handlers.

case $\mathsf{handle}^{\epsilon}\ h^{l^{\eta}} \cdot \mathsf{E} \cdot \mathsf{perform}\ op\ [\overline{\sigma}]\ v \longrightarrow \mathsf{handle}^{\epsilon}\ (f\ [\overline{\sigma}]\ v\ k)$

$op \notin \mathsf{bop}(\mathsf{E})\ \wedge\ (op \mapsto f)\ \in h$	given
$op : \forall\overline{\alpha}.\ \sigma_1 \rightarrow l^{\eta}\ \sigma_2\ \in \Sigma(l)$	given
$k\ =\ \lambda^{\epsilon}\ x : \sigma_2[\overline{\alpha} :=\overline{\sigma}].\ \mathsf{handle}^{\epsilon}\ h \cdot \mathsf{E} \cdot x$	given
$\varnothing \vdash \mathsf{handle}^{\epsilon}\ h \cdot \mathsf{E} \cdot \mathsf{perform}\ op\ \overline{\sigma}\ v : \sigma \mid \epsilon$	given
$\varnothing \vdash_{\mathsf{ops}} h : \sigma \mid l \mid \epsilon$	U-HANDLE
$\varnothing \vdash_{\mathsf{val}} f : \forall\overline{\alpha}_i. \rightarrow \sigma_1 \rightarrow \langle r^{\eta} \mid \epsilon\rangle\ (\sigma_2 \rightarrow \langle r^{\eta} \mid \epsilon\rangle\ \sigma) \rightarrow \langle r^{\eta} \mid \epsilon\rangle\ \sigma$	U-OPS
$\varnothing \vdash_{\mathsf{val}} f : \forall\overline{\alpha}_i. \rightarrow \sigma_1 \rightarrow \langle r^{\eta} \mid \epsilon\rangle\ (\sigma_2 \rightarrow \langle r^{\eta} \mid \epsilon\rangle\ \sigma) \rightarrow \langle r^{\eta} \mid \epsilon\rangle\ \sigma \mid \langle r^{\eta} \mid \epsilon\rangle$	VAL
$\varnothing \vdash f\ [\overline{\sigma}] : \sigma_1[\overline{\alpha} :=\overline{\sigma}] \rightarrow \langle r^{\eta} \mid \epsilon\rangle\ (\sigma_2[\overline{\alpha} :=\overline{\sigma}] \rightarrow \langle r^{\eta} \mid \epsilon\rangle\ \sigma) \rightarrow \langle r^{\eta} \mid \epsilon\rangle\ \sigma \mid \langle r^{\eta} \mid \epsilon\rangle$	TAPP, APP
$\varnothing \vdash \mathsf{perform}\ op\ \overline{\sigma}\ v : \sigma_2[\overline{\alpha} :=\overline{\sigma}] \mid \lceil \mathsf{handle}^{\epsilon}\ h\ \mathsf{E}\rceil^{\ell}(\epsilon)$	Lemma F.4
$\varnothing \vdash_{\mathsf{ec}} \mathsf{handle}^{\epsilon}\ h \cdot \mathsf{E} : \sigma_2[\overline{\alpha} :=\overline{\sigma}] \rightarrow \sigma \mid \epsilon$	above
$\varnothing \vdash v : \sigma_1[\overline{\alpha}\ := \overline{\sigma}] \mid \langle \lceil \mathsf{handle}^{\epsilon}\ h\ \mathsf{E}\rceil^{\ell} \mid \epsilon\rangle$	APP and TAPP
$\varnothing \vdash v : \sigma_1[\overline{\alpha}\ := \overline{\sigma}] \mid \langle r^{\eta} \mid \epsilon\rangle$	Lemma F.5
$x : \sigma_2[\overline{\alpha} :=\overline{\sigma}] \vdash_{\mathsf{val}} x : \sigma_2[\overline{\alpha} :=\overline{\sigma}]$	VAR
$x : \sigma_2[\overline{\alpha} :=\overline{\sigma}] \vdash x : \sigma_2[\overline{\alpha} :=\overline{\sigma}] \mid \epsilon$	VAL
$x : \sigma_2[\overline{\alpha} :=\overline{\sigma}] \vdash_{\mathsf{ec}} \mathsf{handle}^{\epsilon} \cdot \mathsf{E} : \sigma_2[\overline{\alpha} :=\overline{\sigma}] \rightarrow \sigma \mid \epsilon$	weakening
$x : \sigma_2[\overline{\alpha} :=\overline{\sigma}] \vdash \mathsf{handle}^{\epsilon}\ h \cdot \mathsf{E} \cdot x : \sigma \mid \epsilon$	Lemma F.3
$\varnothing \vdash_{\mathsf{val}} \lambda^{\epsilon}\ x : \sigma_2[\overline{\alpha} :=\overline{\sigma}].\ \mathsf{handle}^{\epsilon}\ h \cdot \mathsf{E} \cdot x\ \sigma_2[\overline{\alpha} :=\overline{\sigma}] \rightarrow \epsilon\ \sigma$	ABS
$\varnothing \vdash \lambda^{\epsilon}\ x : \sigma_2[\overline{\alpha} :=\overline{\sigma}].\ \mathsf{handle}^{\epsilon}\ h \cdot \mathsf{E} \cdot x : \sigma_2[\overline{\alpha} :=\overline{\sigma}] \rightarrow \epsilon\ \sigma \mid \epsilon$	VAL
$\varnothing \vdash f\ [\overline{\sigma}]\ v\ k : \sigma \mid \langle r^{\eta} \mid \epsilon\rangle$ a	APP
$\varnothing \vdash \mathsf{handle}^{\epsilon}\ \{\ \}^{r^{\eta}} \cdot f\ [\overline{\sigma}]\ v\ k : \sigma \mid \epsilon$	U-HANDLE

case $(\mathsf{handler}^{\epsilon}\ h^{\ell})\ v \longrightarrow \mathsf{handle}^{\epsilon}_{m}\ h^{\ell^{\eta}} \cdot v\ (m, h^{\ell^{\eta}})$.

$\varnothing \vdash (\mathsf{handler}^{\epsilon}\ h)\ v : \sigma \mid \epsilon$	given
$\varnothing \vdash \mathsf{handler}^{\epsilon}\ h : (\mathsf{ev}\ \ell^{\eta} \rightarrow \epsilon\ \sigma) \rightarrow \epsilon\ \sigma \mid \epsilon$	APP
$\varnothing \vdash v : \mathsf{ev}\ \ell^{\eta} \rightarrow \epsilon\ \sigma \mid \epsilon$	above
$\varnothing \vdash_{\mathsf{val}} \mathsf{handler}^{\epsilon}\ h : (\mathsf{ev}\ \ell^{\eta} \rightarrow \epsilon\ \sigma) \rightarrow \epsilon\ \sigma \mid \epsilon$	VAL
$\varnothing \vdash_{\mathsf{ops}} h : \sigma \mid \ell \mid \epsilon$	N-HANDLER
$\varnothing \vdash v\ (m,\ h^{\ell^{\eta}}) : \sigma \mid \epsilon$	APP
$\varnothing \vdash \mathsf{handle}^{\epsilon}\ h\ (v\ ())\ : \sigma \mid \langle\epsilon\rangle$	N-HANDLE

□

Proof. (*of Theorem 4.1*) Same as Theorem B.1 with Lemma F.11. □

F.6.2 Uniqueness of Names.

Proof. (*of Theorem 4.3*) The same as Theorem B.2. □

F.7 Proofs for Restricted System $\mathsf{F}^{\epsilon+u}$

F.7.1 *Preservation.*

Lemma F.12. (*Small Step Preservation*)
If $\varnothing \vdash e_1 : \sigma \mid \epsilon$ and $e_1 \longrightarrow e_2$, then $\varnothing \vdash e_2 : \sigma \mid \epsilon$.

Proof. (*of Lemma F.12*) By induction on reduction. We discuss the new cases.
case $\mathsf{handle}^{\epsilon}\ h^{l^{\eta}} \cdot \mathsf{E} \cdot \mathsf{perform}\ op\ [\overline{\sigma}]\ v \longrightarrow \mathsf{handle}^{\epsilon}\ \{\ \}^{r^{\eta}}\ (f\ [\overline{\sigma}]\ umb^{\eta}\ v\ k')$

$op \notin \mathsf{bop}(\mathsf{E}) \ \wedge\ (op \mapsto f)\ \in h$	given
$op : \forall \overline{\alpha}.\ \sigma_1 \rightarrow l^{\eta}\ \sigma_2\ \in \Sigma(l)$	given
$k\ =\ \lambda^{\epsilon}\ x : \sigma_2[\overline{\alpha} := \overline{\sigma}].\ \mathsf{handle}^{\epsilon}\ h \cdot \mathsf{E} \cdot x$	given
$k'\ =\ \lambda^{\langle r^{\eta}\ \mid \epsilon\rangle} x : \sigma_2[\overline{\alpha} := \overline{\sigma}].\ \mathsf{mask}^{r^{\eta}}\ k\ x$	given
$\varnothing \vdash \mathsf{handle}^{\epsilon}\ h \cdot \mathsf{E} \cdot \mathsf{perform}\ op\ \overline{\sigma}\ v : \sigma \mid \epsilon$	given
$\varnothing \vdash_{\mathsf{ops}}\ h : \sigma \mid l \mid \epsilon$	U-HANDLE
$\varnothing \vdash_{\mathsf{val}}\ f : \forall \overline{\alpha}_i.\ \mathsf{umb}\ \eta\ \langle r^{\eta} \mid \epsilon\rangle\ \sigma \rightarrow \sigma_1 \rightarrow \langle r^{\eta} \mid \epsilon\rangle\ (\sigma_2 \rightarrow \langle r^{\eta} \mid \epsilon\rangle\ \sigma) \rightarrow \langle r^{\eta} \mid \epsilon\rangle\ \sigma$	U-OPS
$\varnothing \vdash_{\mathsf{val}}\ f : \forall \overline{\alpha}_i.\ \mathsf{umb}\ \eta\ \langle r^{\eta} \mid \epsilon\rangle\ \sigma \rightarrow \sigma_1 \rightarrow \langle r^{\eta} \mid \epsilon\rangle\ (\sigma_2 \rightarrow \langle r^{\eta} \mid \epsilon\rangle\ \sigma) \rightarrow \langle r^{\eta} \mid \epsilon\rangle\ \sigma \mid \langle r^{\eta} \mid \epsilon\rangle$	VAL
$\varnothing \vdash f\ [\overline{\sigma}]\ umb^{\eta} : \sigma_1[\overline{\alpha} := \overline{\sigma}] \rightarrow \langle r^{\eta} \mid \epsilon\rangle\ (\sigma_2[\overline{\alpha} := \overline{\sigma}] \rightarrow \langle r^{\eta} \mid \epsilon\rangle\ \sigma) \rightarrow \langle r^{\eta} \mid \epsilon\rangle\ \sigma \mid \langle r^{\eta} \mid \epsilon\rangle$	TAPP, APP
$\varnothing \vdash \mathsf{perform}\ op\ \overline{\sigma}\ v : \sigma_2[\overline{\alpha} := \overline{\sigma}] \mid \lceil \mathsf{handle}^{\epsilon}\ h\ \mathsf{E} \rceil^{\ell}(\epsilon)$	Lemma F.4
$\varnothing \vdash_{\mathsf{ec}}\ \mathsf{handle}^{\epsilon}\ h \cdot \mathsf{E} : \sigma_2[\overline{\alpha} := \overline{\sigma}] \rightarrow \sigma \mid \epsilon$	above
$\varnothing \vdash v : \sigma_1[\overline{\alpha}\ := \overline{\sigma}] \mid \langle \lceil \mathsf{handle}^{\epsilon}\ h\ \mathsf{E} \rceil^{\ell} \mid \epsilon\rangle$	APP and TAPP
$\varnothing \vdash v : \sigma_1[\overline{\alpha}\ := \overline{\sigma}] \mid \langle r^{\eta} \mid \epsilon\rangle$	Lemma F.5
$x : \sigma_2[\overline{\alpha} := \overline{\sigma}] \vdash_{\mathsf{val}}\ x : \sigma_2[\overline{\alpha} := \overline{\sigma}]$	VAR
$x : \sigma_2[\overline{\alpha} := \overline{\sigma}] \vdash x : \sigma_2[\overline{\alpha} := \overline{\sigma}] \mid \epsilon$	VAL
$x : \sigma_2[\overline{\alpha} := \overline{\sigma}] \vdash_{\mathsf{ec}}\ \mathsf{handle}^{\epsilon} \cdot \mathsf{E} : \sigma_2[\overline{\alpha} := \overline{\sigma}] \rightarrow \sigma \mid \epsilon$	weakening
$x : \sigma_2[\overline{\alpha} := \overline{\sigma}] \vdash \mathsf{handle}^{\epsilon}\ h \cdot \mathsf{E} \cdot x : \sigma \mid \epsilon$	Lemma F.3
$\varnothing \vdash_{\mathsf{val}}\ \lambda^{\epsilon}\ x : \sigma_2[\overline{\alpha} := \overline{\sigma}].\ \mathsf{handle}^{\epsilon}\ h \cdot \mathsf{E} \cdot x : \sigma_2[\overline{\alpha} := \overline{\sigma}] \rightarrow \epsilon\ \sigma$	ABS
$\varnothing \vdash \lambda^{\epsilon}\ x : \sigma_2[\overline{\alpha} := \overline{\sigma}].\ \mathsf{handle}^{\epsilon}\ h \cdot \mathsf{E} \cdot x : \sigma_2[\overline{\alpha} := \overline{\sigma}] \rightarrow \epsilon\ \sigma \mid \epsilon$	VAL
$\varnothing \vdash \lambda^{\langle r^{\eta}\ \mid \epsilon\rangle} x : \sigma_2[\overline{\alpha} := \overline{\sigma}].\ \mathsf{mask}^{r^{\eta}}\ k\ x : \sigma_2[\overline{\alpha} := \overline{\sigma}] \rightarrow \langle r^{\eta}\ \mid \epsilon\rangle\ \sigma \mid \langle r^{\eta} \mid \epsilon\rangle$	VAL
$\varnothing \vdash f\ [\overline{\sigma}]\ umb^{\eta}\ v\ k' : \sigma \mid \langle r^{\eta} \mid \epsilon\rangle\ \mathrm{a}$	APP
$\varnothing \vdash \mathsf{handle}^{\epsilon}\ \{\ \}^{r^{\eta}} \cdot f\ [\overline{\sigma}]\ umb^{\eta}\ v\ k : \sigma \mid \epsilon$	U-HANDLE

case $(\mathsf{handler}^{\epsilon}\ h^{\ell})\ umb^{\eta}\ v \longrightarrow \mathsf{handle}^{\epsilon}_{m}\ h^{\ell^{\eta}} \cdot v\ (m, h^{\ell^{\eta}})$.

$\varnothing \vdash (\mathsf{handler}^{\epsilon}\ h)\ v : \sigma \mid \epsilon$	given
$\varnothing \vdash \mathsf{handler}^{\epsilon}\ h : \mathsf{umb}\ \eta\ \epsilon\ \sigma \rightarrow (\mathsf{ev}\ \ell^{\eta} \rightarrow \epsilon\ \sigma) \rightarrow \epsilon\ \sigma \mid \epsilon$	APP
$\varnothing \vdash v : \mathsf{ev}\ \ell^{\eta} \rightarrow \epsilon\ \sigma \mid \epsilon$	above
$\varnothing \vdash_{\mathsf{val}}\ \mathsf{handler}^{\epsilon}\ h : \mathsf{umb}\ \eta\ \epsilon\ \sigma \rightarrow (\mathsf{ev}\ \ell^{\eta} \rightarrow \epsilon\ \sigma) \rightarrow \epsilon\ \sigma \mid \epsilon$	VAL
$\varnothing \vdash_{\mathsf{ops}}\ h : \sigma \mid \ell \mid \epsilon$	N-HANDLER
$\varnothing \vdash v\ (m,\ h^{\ell^{\eta}}) : \sigma \mid \epsilon$	APP
$\varnothing \vdash \mathsf{handle}^{\epsilon}\ h\ (v\ ())\ : \sigma \mid \langle\epsilon\rangle$	N-HANDLE

case $\mathsf{mask}^{r^{\eta}}\ v \longrightarrow v$. Follows directly by VAL. □

Proof. (*of Theorem E.1*) Same as Theorem B.1 with Lemma F.12. □

F.7.2 *Progress.*

Lemma F.13. (*Progress with effects*)
If $\varnothing \vdash e_1 : \sigma \mid \epsilon$, then either
(1) e_1 is a value; or
(2) $e_1 \longmapsto e_2$; or

(3) $e_1 \;=\; \mathsf{E}[\mathsf{perform}\ op\,\overline{\sigma}\ v]$, where $op : \forall\overline{\alpha}.\ \sigma_1 \rightarrow l^{\eta}\ \sigma_2 \;\in \Sigma(l)$, and $op \notin \mathsf{bop}(\mathsf{E})$.
(4) $e_1 \;=\; \mathsf{E}[\mathsf{perform}\ op\,\overline{\sigma}\ (m,\ h^{\ell^{\eta}})\ v]$, where $op : \forall\overline{\alpha}.\ \sigma_1 \rightarrow l^{\eta}\ \sigma_2 \;\in \Sigma(\ell)$, and E has no $\mathsf{handle}_m\ h^{\ell^{\eta}}$.

Proof. (*of Lemma F.13*) By induction on typing. The proof structure is similar to Lemma F.8 and Lemma F.10, but with two cases where e_1 does not take a step. □

Lemma F.14. (*Evidence does not Escape*)
If $\varnothing \vdash e : \sigma \mid \epsilon$, where e is a handle-safe System $\mathsf{F}^{\epsilon+u}$ expression, and $e \;=\; \mathsf{handle}_m\ h^{\ell^{\eta}} \cdot v$, then according to rule (*n-return*) we have $e \longrightarrow v$, and v does not contain evidence $(m,\ h^{\ell^{\eta}})$ that can later be used.

Proof. (*of Lemma F.14*) We case analyze the shape of v.

- v is x, handler h^{l}, perform$^{\epsilon}$ $op\,\overline{\sigma}$, or handler h^{ℓ}. In this case, the goal holds trivially.
- v = perform$^{\epsilon}$ $op\,\overline{\sigma}\ v$. In this case, we care about whether v is $(m,\ h^{\ell^{\eta}})$. If it is not, then the goal holds. If it is, that means the return type of handle contains η. However, as a handle-safe expression, this handle must have been reduced from handler. With the umbrella witness, we know that its return type cannot contain η. So contradiction.
- v is an evidence. Using similar reasoning as above, we know that v cannot be $(m,\ h^{\ell^{\eta}})$.
- v is a term abstraction or type abstraction. Then there are two cases to be discussed in the body.
  - As reasoned before, the body of v cannot directly contain evidence $(m,\ h^{\ell^{\eta}})$ that can later be used. Otherwise, the scope η would appear in the return type, which is not allowed.
  - There is a special case where v may contain an evidence, and does not expose the scope in the return type. That is, the body of v contains handle $h^{l} \cdot$ perform$^{\epsilon}$ $op\,\overline{\sigma}\ (m,\ h^{\ell^{\eta}})$. Here, we perform on the evidence, but use the umbrella handler handle h^{l} to eliminate the effect in the result row, and thus pretend that the evidence does not escape. An important observation is that, for handle-safe expressions, resumptions are the only expressions that can contain handle and potentially cause the problem. Recall also that, with the resume effect, we can never return the resumption inside an abstraction, because the resume effect r^{η} will then appear in the return type, which is not allowed by the umbrella witness. Thus this case is also impossible.

  □

Proof. (*of Theorem 4.2*) By applying Lemma F.13, we know that either e_1 is a value, or $e_1 \longmapsto e_2$, or $e_1 \;=\; \mathsf{E}[\mathsf{perform}\ op\,\overline{\sigma}\ v]$, where $op : \forall\overline{\alpha}.\ \sigma_1 \rightarrow \sigma_2 \;\in \Sigma(l)$, and $op \notin \mathsf{bop}(\mathsf{E})$, or $e_1 \;=\; \mathsf{E}[\mathsf{perform}\ op\,\overline{\sigma}\ (m,\ h^{\ell^{\eta}})$
where $op : \forall\overline{\alpha}.\ \sigma_1 \rightarrow l^{\eta}\ \sigma_2 \;\in \Sigma(\ell)$, and E has no $\mathsf{handle}_m\ h^{\ell^{\eta}}$.

For the first two cases, we have proved the goal. For the third case, we prove the goal by contradiction.

$\varnothing \vdash \mathsf{E}[\mathsf{perform}\ op\,\overline{\sigma}\ v] : \sigma \mid \langle\rangle$	given
$l \in \lceil \mathsf{E} \rceil^{\ell}(\langle\rangle)$	Lemma F.4
$op \notin \mathsf{bop}(\mathsf{E})$	given
$l \notin \lceil \mathsf{E} \rceil^{\ell}(\langle\rangle)$	follows
Contradiction	

The fourth case means that the evidence $(m,\ h^{\ell^{\eta}})$ escapes its handler $\mathsf{handle}_m\ h^{\ell^{\eta}}$, which is impossible by Lemma F.14. □