First-Class Names for Effect Handlers

NINGNING XIE, University of Cambridge, UK
YOUYOU CONG, Tokyo Institute of Technology, Japan
KAZUKI IKEMORI, Tokyo Institute of Technology, Japan
DAAN LEIJEN, Microsoft Research, USA

Algebraic effects and handlers are a promising technique for incorporating composable computational effects into functional programming languages. Effect handlers enable concisely programming with different effects, but they do not offer a convenient way to program with different instances of the same effect. As a solution to this inconvenience, previous studies have introduced *named effect handlers*, which allow the programmer to distinguish among different effect instances. However, existing formalizations of named handlers are both involved and restrictive, as they employ non-standard mechanisms to prevent the escaping of handler names.

In this paper, we propose a simple and flexible design of named handlers. Specifically, we treat handler names as first-class values, and prevent their escaping while staying within the ordinary λ-calculus. Such a design is enabled by combining named handlers with *scoped effects*, a novel variation of effects that maintain a scope via rank-2 polymorphism. We formalize two combinations of named handlers and scoped effects, and implement them in the Koka programming language. We also present practical applications of named handlers, including a neural network and a unification algorithm.

CCS Concepts: • **Software and its engineering** → **Control structures**; **Polymorphism**; • **Theory of computation** → **Type theory**.

Additional Key Words and Phrases: Algebraic Effects, Effect Handlers, Scoping

ACM Reference Format:
Ningning Xie, Youyou Cong, Kazuki Ikemori, and Daan Leijen. 2022. First-Class Names for Effect Handlers. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 126 (October 2022), 30 pages. https://doi.org/10.1145/3563289

1 INTRODUCTION

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

The question of how to represent computational effects has been studied for decades in the programming languages community. *Algebraic effects and handlers* [Plotkin and Power 2003; Plotkin and Pretnar 2013] are one of the solutions to this problem. Since their introduction, effect handlers have been applied to diverse domains, such as Web programming [Hillerström and Lindley 2016], reactive programming [Bračevac et al. 2018], and probabilistic programming [Bingham et al. 2019].

A key advantage of effect handlers is that they can be freely composed. For example, if we have a program that uses both mutable state and exceptions, we can simply run it under two handlers: one for state and the other for exceptions. When an effectful operation is performed, it is automatically handled by the innermost handler for that effect.

Authors' addresses: Ningning Xie, University of Cambridge, Cambridge, UK, ningning.xie@cl.cam.ac.uk; Youyou Cong, Tokyo Institute of Technology, Tokyo, Japan, cong@c.titech.ac.jp; Kazuki Ikemori, Tokyo Institute of Technology, Tokyo, Japan, ikemori.k.aa@m.titech.ac.jp; Daan Leijen, Microsoft Research, Redmond, WA, USA, daan@microsoft.com.

On the other hand, plain effect handlers do not provide a convenient way to program with *different instances* of the *same effect*. For example, if we wish to use multiple states or multiple files, we typically need to insert code for skipping intervening handlers [Biernacki et al. 2017; Convent et al. 2020; Leijen 2014; Wu et al. 2014]. This is cumbersome and fragile.

As a generalization of effect handlers, Biernacki et al. [2019] and Zhang and Myers [2019] formalize a *named* variant of handlers. With named handlers, the programmer can distinguish among different effect instances by passing a handler name as an argument to an operation. However, named handlers increase the complexity of the calculus, because, without special care, handler names can easily *escape* the scope of their handler during evaluation. Previous studies address this challenge by introducing names as *second-class* values with special syntactic constructs, in combination with a richer type system (e.g., a form of *region typing*) to prevent name escaping. This restricts the expressiveness of the calculus, and makes it hard to incorporate named handlers into existing frameworks.

In this paper, we study a new design of named handlers, where handler names are *first-class* values, and where well-scopedness of handler names is guaranteed through *higher-rank polymorphism*. We realize this by combining named handlers with *scoped effects*, a novel variation of effects for safe management of resources. To explore the design space of named handlers, we present two combinations of named handlers and scoped effects, and discuss their trade-offs. We believe that our work will provide new insights into the design of named handlers, and more generally, increase the expressive power of effect handlers. Our specific contributions are summarized as follows:

- We demonstrate the practicality of named handlers through various examples (Section 2). In particular, we use a file handling program to show how named handlers allow us to interact with resources (Section 2.3.1).
- We present a novel technique of combining a regular handler (called *umbrella handler*) and named handlers, which enables *dynamic* creation of named handlers using finite types. We illustrate this by implementing a first-class heap with dynamic mutable references as named handlers (Section 2.3.2).
- We establish the notion of scoped effects as a solution to the name escaping problem (Section 3.2.2). We maintain scopes using rank-2 polymorphism [Jones 1996; McCracken 1984], a well-understood technique found in work on monadic encapsulation [Peyton Jones and Launchbury 1995; Timany et al. 2017]. As an application of scoped effects, we show an implementation of safe resource interface.
- We formalize two combinations of named handlers and scoped effects (Sections 4 and 5), where handler names are first class. Both combinations are a higher-order polymorphically typed lambda calculus in the style of System F_ω, extended with row-polymorphic algebraic effects [Hillerström and Lindley 2016; Leijen 2014]. Their difference is in the expressiveness and complexity; in particular, one of the combinations has simple typing rules but cannot express umbrella handler, whereas the other combination can fully express our heap example but has more complex typing rules.
- We implemented named handlers and scoped effects in Koka [Koka 2019], a programming language with native support for algebraic effects and handlers (Section 6). We show how our formalization of named handlers allows straightforward integration into the Koka compiler. We further present two larger applications of first-class named handlers: a neural network and a unification algorithm (Section 7). The implementation and samples are available as an artifact [Xie et al. 2022].

In the appendix of the supplementary technical report [Xie et al. 2021], we provide the full typing rules and soundness proofs of the two systems discussed in Sections 4 and 5. We also present the specification of two systems that are not detailed in the paper, one featuring only named handlers, and the other featuring only scoped effects.

2 PROGRAMMING WITH NAMED HANDLERS

In this section, we demonstrate how named handlers are useful in practice, using various examples enabled by our design of named handlers in Koka. We begin with a brief introduction to algebraic effects and handlers (Section 2.1), and motivate the need for named handlers (Section 2.2). We then present two applications of named handlers: file handling (Section 2.3.1) and first-class heap (Section 2.3.2). We show that the two applications can both be implemented in a fully functional way, thanks to the expressiveness of effect handlers (Section 2.4).

2.1 Effect Handlers in Koka

Algebraic effects [Plotkin and Power 2003] and handlers [Plotkin and Pretnar 2013] are an abstraction of user-defined effects. Intuitively, the abstraction can be understood as a generalization of exceptions and their handlers. That is, the programmer can declare their own effect, and specify its behavior using a handler.

Let us illustrate the idea of algebraic effects by implementing a read-only state in the Koka language. The first step is to write the signature of the `read` effect:

```
effect read
  ask() : int
```

The keyword `effect` defines a new effect signature, consisting of an effect *label* `read` and a control *operation* `ask`. The type of `ask` tells us that, when given a unit argument, it returns an integer and produces a `read` effect.

We next specify the behavior of the `read` effect by implementing a handler for this effect[1]:

```
fun read(x, action)
  (handler
     ask(){ resume(x) }
  )(action)
```

The `handler` expression takes a set of operation definitions, and returns a function that takes as its argument a thunked computation `action` to be handled. In our example, the handler defines the interpretation of `ask()` as resuming with `x`. Here, the `resume` keyword represents the delimited continuation from the `ask` operation up to the handler; passing `x` to `resume` thus means that every call to `ask()` in the handled computation `action` returns `x` as its result. Hence, we can view the operation `ask` as a statically typed, but dynamically bound function.

We can now write a program that uses the `ask` operation:

```
fun main()
  read(1, fn(){ ask().println })  // prints 1
```

Using algebraic effects and handlers, one can implement a wide range of computational effects, such as exceptions, nondeterminism, async/await style interleaved computations, backtracking, etc [Leijen 2017; Pretnar 2015] . In fact, it has been shown that algebraic effects and handlers are equally expressive as monads in an untyped setting [Forster et al. 2019].

The Dot Syntax. In the above program, we have an expression `ask().println` that uses the *dot* syntax. The expression is equivalent to `println(ask())`, i.e., it passes the expression `ask()` as the first argument to `println`. Koka provides the dot syntax as a convenient way of writing function applications. Formally, the syntax is defined as follows:

[1]In both Koka and our formalization, types and values are in separate name spaces. Therefore, in our example, we can define a function `read` that handles the `read` effect type. For clarity, we use a teal color for all types in this paper.

```
e.f(e1,...,en)            ⇝ f(e,e1,...,en)
```

The dots associate to the left, making it easy to compose multiple functions. For instance, we can write `[1,2,3].map(inc).sum.println` instead of `println(sum(map([1,2,3],inc)))`.

With-statements. Recall that handlers take a thunked computation. If we want to handle a computation that is not already a function (e.g., `ask.println`), we must wrap it around a lambda abstraction (e.g., `fn(){ ask().println }`). Koka provides the `with` statements as syntactic sugar to avoid writing such abstractions explicitly. There are two variants of the `with` statements, both of which allow the programmer to apply a function over the rest of the scope without manually needing to wrap it with a lambda abstraction:

```
with f(e1, ..., en) body        ⇝  f(e1, ..., en, fn(){ body })
with x <- f(e1, ..., en) body   ⇝  f(e1, ..., en, fn(x){ body })
```

Using `with`, we can write the previous `main` function as:

```
fun main()
  with read(1)
  ask().println  // prints 1
```

The `with` syntax can also be applied to `handler` expressions. For example, we can rewrite the definition of the `read` function as follows.

```
fun read(x, action)
  with handler
    ask(){ resume(x) }
  action()
```

We will use the `with` syntax throughout the rest of the paper.

2.2 Named Handlers

The `read` effect discussed above allows us to maintain a single, read-only state. The ability is however not sufficient when we wish to use multiple instances of such state. Consider the following program:

```
fun main()
  with read(2)
  with read(1)
  (ask() + ask()).println  // prints 2
```

The program creates two `read` handlers and makes two calls to the `ask` operation. Since operations are always handled by the inner-most enclosing handler, both occurrences of `ask()` are interpreted as `1`, and hence the program prints `2`.

But what if we would like the first `ask` to be handled by the inner handler, and the second `ask` by the outer handler? One possible solution is to use the *masking* technique [Biernacki et al. 2017; Convent et al. 2020; Leijen 2014; Wu et al. 2014]. Intuitively, masking allows one to skip the innermost handler surrounding an operation. For instance, the following program interprets the first `ask()` operation using the outer reader handler, and thus evaluates to `3`:

```
fun main()
  with read(2)
  with read(1)
  (mask⟨read⟩{ ask() } + ask()).println  // prints 3
```

The main purpose of `mask` is however to encapsulate effects, and hence it is not an ideal tool for selecting specific handlers. In general, if we use `mask` to choose among nested handlers, we must know the exact number and order of handlers surrounding an expression.

```
named effect file
  read-line() : string

fun file(fname, action)
  var ls := read-text-file(fname).lines
  with f <- named handler
    read-line()
      match ls
        Nil        → resume("")
        Cons(x,xx) → { ls := xx; resume(x) }
  action(f)
```

```
fun catlines( t )
  t.fst.read-line() ++ ","
   ++ t.snd.read-line()

fun main()
  with f_1 <- file("foo.txt".path)
  with f_2 <- file("bar.txt".path)
  catlines( (f_1,f_2) ).println
```

Fig. 1. File Handling

A better solution to the above problem is to use *named handlers*. Named handlers have a unique name, which allows the programmer to specify the intended association between an operation and a handler. In Koka, we can implement the `read` effect using named handlers as:

```
named effect read
  ask() : int

fun read(x, action)
  with h <- named handler
              ask(){ resume(x) }
  action(h)
```

```
fun main()
  with h_2 <- read(2)
  with h_1 <- read(1)
  (h_2.ask() + h_1.ask()).println
  // prints 3
```

In the definition of the `read` function, the action is a function taking in a handler name, rather than a mere thunk. In the main program, the inner and outer handlers are given names h_1 and h_2, respectively, and the two calls to the `ask` operation use these names to specify the intended interpretation. This allows the first `ask` to skip the inner handler h_1 and be handled by the outer handler h_2. Thus, the program prints `3` as desired.

Of particular interest here is that the handler names h_1 and h_2 are *first-class* values, introduced simply by *lambda abstractions*. Indeed, if we expand the syntactic sugar of `with`, we get:

```
fun main()
  read(2, fn(h_2){ read(1), fn(h_1) {(h_2.ask() + h_1.ask()).println} }) // prints 3
```

The first-class status of handler names and their introduction by lambda abstractions are different from existing named handler calculi (see Section 8). Furthermore, as we show in the next section, these allow us to build interesting programs using named handlers.

2.3 Examples of Named Handlers

Having seen a simple example of named handlers, we look at two programs that use named handlers in a more interesting way enabled by our design. The two examples presented in this section both rely on the first-class status of handler names (and thus cannot be written in existing named handler calculi [Biernacki et al. 2019; Zhang and Myers 2019]). Moreover, the second example also uses *dynamic instantiation* of handlers.

2.3.1 File Handling. In this section, we model file handling as an effect, and use named handlers to handle multiple opened files independently at the same time.

Figure 1 presents a program that prints the first lines of two files. We first declare `file` as a named effect, using the keywords `named effect`, with a single operation `read-line`. We next define the `file` function that creates a handler for the given file. Here, we use Koka's library functions (`read-text-file` and `lines`) and *local mutable state* (`var`) to open the file and store its content as a list of lines. When the content is non-empty, we use the first line as the interpretation of the `read-line` operation. Then, in the main program, we call the `file` function to create two file handlers: f_1 for

```
effect heap
  newref(init : int) : ref

named effect ref
  get() : int
  set(value : int) : ()

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

fun heap(action)
  with handler
    newref(init){ ref(init, resume) }
  action()

fun maketwo(x,y)
  (newref(x), newref(y))

fun main()
  with heap
  val (r1, r2) <- maketwo(20,22)
  println( r1.get() + r2.get() ) // 42
```

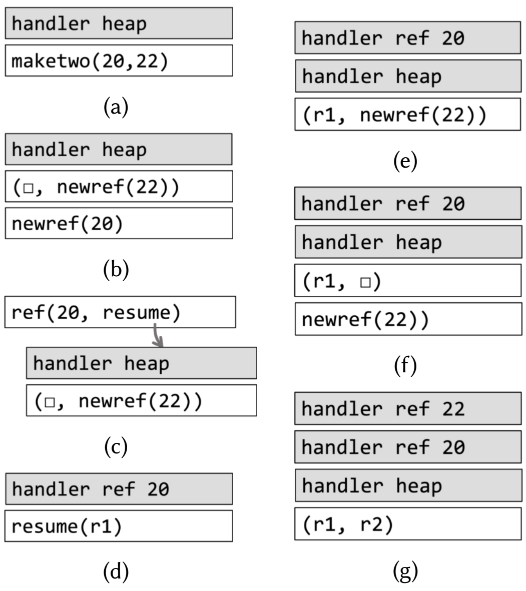


Fig. 2. First-class Heap

foo.txt, and f_2 for bar.txt. Since handler names are first-class values, we can package them in a tuple after creating handlers, and pass it to catlines, which uses the standard fst and snd functions to select out the handler names, reads a line from the two files, and returns the concatenated result.

Note that, without named handlers, the two calls of read-line would both evaluate to the first line of "bar.txt", just like what would happen when calling ask twice under two unnamed handlers. With named handlers, we can deal with each file independently, and easily read the content of a particular file in the presence of multiple opened files.

Note also that, if handler names were not first class, they could not be put into a tuple and selected out using standard functions. When being first class, handler names can be freely passed around, which is crucial for realistic applications (see Section 7).

2.3.2 A First-Class Heap. As a more advance application of named handlers, we demonstrate an implementation of first-class heap with dynamic references. The implementation shows that, in a language with named handlers, we do not need built-in state; we can express it by simply using named handlers.

In the read and file examples, we were able to distinguish between different states and files using handler names. However, in both examples, we had to instantiate handlers upfront over the scope that uses them. This is an unfortunate limitation if we wish to implement ML-style references, which can be created and returned dynamically in leaf functions and be used in a parent scope. Is it possible to model a first-class heap using algebraic effects?

It turns out that we can use a combination of a plain effect handler (for the heap) and named handlers (for the references) to model such dynamic resources. On the left of Figure 2, we show an implementation of such first-class heap with dynamic references. We begin by declaring two effects. The named effect ref represents a reference cell, consisting of the get and set operations for accessing and updating the reference value. For simplicity, we present only integer references, but in Koka, reference cells can be polymorphic (that is, a heap can have references of different types). The unnamed heap effect has an operation newref for creating a new reference cell. The function ref creates a new reference handler initialized to init, and the function heap creates a new heap handler, which interprets newref as handling the rest of the computation resume using the ref function.

In the main function, we first install the heap handler, and then allocate a tuple of references using maketwo. When called, the function adds the current values of the references and prints the result. An important observation in this example is that *the references* r_1 *and* r_2 *are returned dynamically – without being created directly under a handler*! To see what it means, consider how we would write this example by using solely named handlers:

```
fun main()
  with heap
  with r₁ <- ref(20)
  with r₂ <- ref(22)
  println( r₁.get() + r₂.get() ) // 42
```

In this implementation, we statically instantiate two reference cells upfront. But in Figure 2, we dynamically create and return two handlers. Such dynamic creation can further depend on the values that are only available at runtime. For example, we can define a maken function that creates n reference cells, with n provided only at runtime.

The trick used here is that we treat newref as a regular operation in the heap handler. When newref is called, it installs a fresh reference handler just *above the heap handler*, and resumes back with the result. More specifically, newref calls the ref function to insert a fresh reference handler, and passes its own resumption resume as the action argument. The ref function uses a named handler to implement the get and set operations with name r, and passes that name to its action argument, effectively resuming to the original call site with a fresh reference name.

To show how this trick works, we present the evaluation steps of our program on the right of Figure 2. Here, each evaluation frame is represented as a box, with handlers highlighted in grey. The evaluation frames grow top-down, with the expression currently being evaluated on the bottom.

Let us focus on the key part of the program, as given in 2a. First, the maketwo function expands to a tuple of two newrefs, and the program turns into 2b. The current control is newref(20), and its result will be plugged into the hole □ in the evaluation frame above it.

Next, the expression newref(20) is handled by the heap handler, and the program turns into 2c. The ref function creates a new reference cell, and applies its action argument resume (which points to the current continuation) to a fresh name r_1, as given in 2d.

Now, the continuation resume is called with r_1, and the program turns into 2e. By comparing 2e and 2b, we can see that the result of evaluating newref(20) is a new handler that is *dynamically* installed above the heap handler, with a new name r_1 referring to it.

With the first element of the tuple evaluated to a value, we proceed to the second element of the tuple, as given in 2f. By repeating the process from 2b to 2e, we dynamically create another reference above the heap handler (and above the reference handler created before), with a fresh name r_2 referring to the reference, as shown in 2g. What is left is to evaluate println(r_1.get() + r_2.get()), which can be done in a straightforward manner.

Observe that, in 2g, we have two reference cells available in the evaluation context, as well as the original heap handler that can be used to create more reference cells. This is quite a feat: while the semantics for (named) effect handlers has no concept of a heap or mutation, it can still express a dynamic heap with polymorphic references! Moreover, as ref handlers can be installed at the outer scope of the umbrella heap handler, the references can be created and used freely under that encompassing scope and returned from functions.

Let us close this section by comparing the file and heap examples. In the former, the handlers for the two files are created *statically* at the top level. In the latter, the handlers for the reference cells are created *dynamically* during evaluation right above the top-level heap handler. With the ability to dynamically create handlers, the programmer does not need to know how many handlers are required at implementation time. As such, dynamic handlers can be useful not only for implementing references, but also for other resources such as tasks and network connections.

2.4 Locally Isolated Handler State

In both file and heap examples, we used local mutable state in the form of the `var` declarations. Hence, the implementation may not seem fully functional to the reader. However, as we show in this section, we can express local mutable state in terms of plain effect handlers. This means, all examples we have shown so far are fully expressible with just (named) effect handlers!

There are many occasions where a handler needs a form of local state. In the algebraic effects literature, there is an elegant solution known as *parameterized handlers* [Bauer and Pretnar 2015; 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.

Locally isolated handler state is a new technique that requires no extensions to the core calculus. The idea is to use standard masking (Section 2.2) combined with a regular monadic state handler. The definition of monadic local state using an effect handler is standard [Kammar and Pretnar 2017; Plotkin and Pretnar 2009], and can be implemented as[2]:

```
effect state
  peek() : int            // read the state
  poke( x : int ) : ()    // write the state

fun mstate(action)
  with handler
    return(x){ fn(s){ x } }
    peek()   { fn(s){ resume(s)(s) } }
    poke(x)  { fn(s){ resume(())(x) } }
  action()

fun state(init : a, action : () → ⟨state|e⟩ b) : e b
  val f = mstate(action)
  f(init)
```

We can now use the `state` handler to implement programs involving local variables of the form `var s := init`. Specifically, we replace all assignments and dereferences of `s` with `poke` and `peek` respectively. For example, we can rewrite the `ref` handler as:

```
fun ref(init, action)
  with state(init)
  with r <- named handler
    get() { resume(peek()) }
    set(x){ poke(x); resume(()) }
  action(r)
```

This is almost right, except that it allows `action` to access our local state as well! Here is where `mask` comes into play: it can be used to hide the state effect in the action such that the state is truly local to the handler and not observable from anywhere else:

[2]For simplicity, the example shows the encoding of the *integer* state, though in Koka, effects can be polymorphic and we can define monadic polymorphic local state.

```
fun ref(init, action)
  with state(init)
  with r <- named handler
    get() { resume(peek()) }
    set(x){ poke(x); resume(()) }
  mask⟨state⟩{ action(r) }
```

This interpretation of `var` removes the need for parameterized handlers in the semantics and runtime system. In Koka, we further reduce the overhead caused by the monadic encoding by implementing the standard `state` handler using direct mutable state in the backend [Xie and Leijen 2020].

3 FORMALIZING NAMED HANDLERS

Having seen the practical applications of named handlers, let us turn our attention to the formalization of named handlers. As shown by previous work [Biernacki et al. 2019; Leijen 2018; Zhang and Myers 2019], named handlers pose a challenge to type soundness: a naive type system may accept a program that gets stuck at runtime. In the following subsections, we illustrate the known challenge (Section 3.1) and provide an overview of our approach (Section 3.2).

3.1 Named Handlers and Scopes

The main challenge with named handlers is that names may escape their scope in the course of evaluation. Let us consider the following program and its evaluation steps:

```
fun main()
  with h2 <- read(2)
  fun f()
    with h1 <- read(1)    ⟶   with h2 <- read(2)              ⟶   with h2 <- read(2)
    fun g(){ h1.ask() }       println((fun g(){h1. ask})())       println( h1. ask )
    g
  println( (f())() )
```

We see that the action being handled by the handler h_1 returns a function that performs the `ask` operation on h_1. Since the function `g` is a value, it is returned as the result of the handler h_1. The returned function is then applied, causing the `ask` operation to be performed. However, at this point, the h_1 handler is no longer surrounding the `ask` operation. As a consequence, the program fails to evaluate to a value. Thus, to build a type-sound calculus with named handlers, we must find a way to statically detect the escaping of names.

3.2 Our Solution

We propose a simple and flexible solution to the name escaping problem. Specifically, we keep handler names as *first-class* values and stay within the standard λ-calculus with higher-rank types. Then, we prevent name escaping through *scoped effects*: a novel concept that is orthogonal to named handlers. As the name suggests, scoped effects have a scope associated with them, which is maintained using rank-2 polymorphism. When combined with named handlers, the rank-2 type prevents names from escaping. In what follows, we briefly review the power of rank-2 polymorphism (Section 3.2.1), introduce the notion of scoped effects (Section 3.2.2), and present two ways of combining named handlers and scoped effects (Sections 3.2.3 and 3.2.4).

3.2.1 A Quick Review of `runST`. The escaping of handler names shares similarities with the escaping of mutable references from a state thread in Haskell. Let us consider the Haskell program below, which is borrowed from Peyton Jones and Launchbury [1995]:

```
let v = runST (newVar True) in runST (readVar v)
```

```
abstract type ix⟨s⟩
  Ix(i : int)

scoped effect vec⟨s⟩
  push(x : string) : ix⟨s⟩
  find(i : ix⟨s⟩) : string
```

```
fun vector(action : forall⟨s⟩ () → ⟨vec⟨s⟩|e⟩ a) : e a
  var vec := []
  with handler
    push(x)
      vec := vec ++ [x]
      resume(Ix(vec.length - 1))

    find(Ix(i))
      resume(vec[i])

  action()
```

Fig. 3. Vectors as Scoped Effects

Here, `newVar` is a function that allocates a new reference with an initial value, and `runST` is a function that runs a state thread. When programming with state, we must make sure that a reference from one thread is never used in a different thread. This is necessary for making the result of a program independent of the order of evaluation. The above program, however, involves an invalid use of the reference `v`: it is created in the first thread and accessed in the second thread. In other words, the reference `v` *escapes* from the first thread. This means the program may evaluate to different values under different evaluation strategies.

To statically reject inappropriate use of references, Peyton Jones and Launchbury [1995] assign the following type to `runST`:

```
runST :: forall a. (forall s. ST s a) → a
```

Observe that the type uses *rank-2 polymorphism*: it has a universal quantifier in the domain of an arrow type. The quantified type variable `s` can be understood as the name of a state thread[3]. Making `s` parametric means that the argument of `runST` cannot make any assumption about the initial state. The order of the two quantifiers further means that the argument cannot return the reference or any computation that depends on the final state.

With this rank-2 type, we can statically reject the problematic example above. The reason is as follows. Assuming we have `v : MutVar s Bool` in the typing environment, we can derive `readVar v : ST s Bool`. In order to pass it to `runST`, we must universally quantify the type variable `s`. However, we cannot do this because `s` occurs free in the typing environment. Thus, the rank-2 type of `runST` combined with the `ST s` monad makes it impossible for a reference to escape from a state thread.

3.2.2 From `runST` to Scoped Effects. The trick behind `runST` is useful not just for monadic encapsulation; it also allows us to implement a safe resource interface as an (unnamed) effect handler. In Figure 3, we present an instance of such interface, which is inspired by an example from Derek Dreyer's Milner Award lecture [Dreyer 2018]. Here, the `vec` effect associates an abstract index `ix` with a list of strings, which is maintained by the `vector` function using a local state `vec`. The state is initially set to an empty list, and can be extended by the `push` operation or looked up by the `find` operation.

To safely program with the `vec` effect, we need to guarantee that a lookup of a state is always performed with the index of that particular state. For instance, the program below would cause an out-of-bound error, because the index `i` passed to `find` is obtained from the first handler, whose state was non-empty, while the lookup is handled by the second handler, whose state is still empty.

```
val i = vector( fn(){ push("hello") } )
vector( fn(){ find(i) } )
```

[3]The scope variable `s` is a *phantom type* [Hinze 2003; Leijen and Meijer 1999], in that it is not inhabited by any value.

```
named scoped effect read⟨s⟩
  ask() : int

fun read(x : int,
         action : forall⟨s⟩ read⟨s⟩ → ⟨read⟨s⟩|e⟩ a
        ) : e a
  with h <- named handler
              ask(){ resume(x) }
  action(h)
```

```
fun main()
  with h2 <- read(2)
  fun f()
    with h1 <- read(1)
    fun g(){ h1.ask() }
    g
  println( (f())() )
  // rejected
```

Fig. 4. Scoped `read` Effect

We enforce safe lookups by defining `vec` as a *scoped effect*, as given in Figure 3. A scoped effect is an effect whose label carries a scope variable. In our example, the effect label `vec` has a type variable `s`, representing the scope of a particular handler instance (and corresponding to the `s` parameter in the type of `runST`). The handler of a scoped effect is assigned a rank-2 type. In our case, the function `vector` takes in an action whose type is polymorphic over the scope `s`, meaning that the action must work regardless of the initial state of the handler[4]. Lastly, observe that the type `ix⟨s⟩` of indices is attached a scope variable `s`. This tells us that every index is associated with a specific handler scope.

Now, since the type `ix` is abstract (e.g., the constructor `Ix` is private), we can be sure that vector lookups never fail at runtime – the index is always within the bounds of the local list. The reason is that each index of type `ix⟨s⟩` is uniquely associated with a handler that handles the `vec⟨s⟩` effect. This prevents us from passing to `find` an index obtained by `push`ing to some other state.

Back to the problematic example above, we can see that it is statically rejected for the same reason as the `runST` example from Section 3.2.1. In particular, the index type of `i` includes the scope variable of the first action, which can no longer be universally quantified now as its scope variable occurs free in the typing environment.

Note that it is important that the effect itself (`vec⟨s⟩`) includes the scoped type variable. This ensures that we cannot hide the use of an index inside a lambda. For example,

```
val f = vector( fn(){ val i = push("hi"); (fn(){ find(i) }) } )
f()
```

is statically rejected, because the returned function has type `() → vec⟨s⟩ string`, meaning that `s` has escaped through the effect type. In Haskell, the escaping is prevented by the use of the `ST s` monad instead.

3.2.3 Named Handlers with Scoped Effects.

Having introduced scoped effects, we return to our initial question: how to statically guarantee well-scopedness of handler names. Our answer is to combine named handlers and scoped effects. In this section, we sketch one combination of these concepts, which allows us to safely implement the `read` (Section 2.2) and `file` effects (Section 2.3.1).

In Figure 4, we present a new implementation of the `read` effect. We first declare `read` as a *named* and *scoped* effect, and thus `read` has a rank-2 type, where the action type is polymorphic over the scope. Note that the first input type `read⟨s⟩` of `action` represents a handler name.

With these modifications, it is no longer possible for a handler name to escape its scope. In our specific example, the type of the function `g` mentions the scope variable associated to the inner handler `h1`. This variable cannot be universally quantified as it occurs free in the typing environment.

[4]The type `a → ⟨l|e⟩ b` represents a function that accepts an input of type `a`, performs an effect `l` and other effects in `e`, and produces an output of type `b`.

```
scoped effect heap⟨s⟩
  newref(init : int) : ref⟨s⟩

named effect ref⟨s⟩ in heap⟨s⟩
  get() : int
  set(value : int) : ()

fun ref(
  init: int,
  action:(ref⟨s⟩ → ⟨ref⟨s⟩|e⟩ a) → e a
         ) : e a
  var s := init
  with r <- named handler
    get() { resume(s) }
    set(x){ s := x; resume() }
  action(r)
```

```
fun heap(action : forall⟨s⟩ () → ⟨heap⟨s⟩|e⟩ a
       ) : e a
  with handler
    newref(init){ ref(init, resume) }
  action()

fun maketwo(x : int, y : int)
  : heap⟨s⟩ (ref⟨s⟩,ref⟨s⟩)
  (newref(x), newref(y))

fun main() : console ()
  with heap
  val (r₁,r₂) <- maketwo(20,22)
  println( r₁.get() + r₂.get() )
```

Fig. 5. First-class Heap with Umbrella Effect

Based on the same idea, we can also maintain a safety invariant in the file example. Specifically, we declare the `file` effect as a scoped effect, and thus prevent any file handle from escaping the scope of its associated handler.

3.2.4 Named Handlers under Scoped Effects. By combining named handlers and scoped effects in a different way, we can also implement the heap example from Section 2.3.2 in a safe manner. The idea is to scope the dynamic named handlers (e.g., the references) under a single scoped effect (e.g. the heap), which we call an *umbrella effect*.

In Figure 5, we present the implementation of a first-class scoped heap. Here, the `heap` effect serves as the umbrella effect scoping over the `ref` handlers under it. The interplay between the two effects is managed with care at the level of types. Observe that `ref` is a named effect defined *in* the scoped effect `heap`, and thus carries the same scope `s` as `heap`. Defining `ref` this way gives the `get` and `set` operations the `heap` effect – for example, `get : ref⟨s⟩ → heap⟨s⟩ int`. Moreover, since `heap` is a scoped effect, its handler is given a rank-2 type, where the action type has a universal quantification over the scope variable `s`.

The use of an umbrella effect in this example helps us prevent the escaping of references. In particular, all of the `newref`, `get`, and `set` operations must be under the scope of a `heap` handler, which in turn guarantees the existence of all dynamic `ref` handlers, since `ref` handlers are created right above the `heap` handler.

Besides dynamic creation of references under a `heap` scope, the umbrella effect allows us to treat references homogeneously. Without an umbrella effect, different references would have different scopes (e.g., $\mathtt{ref}\langle s_1\rangle$ and $\mathtt{ref}\langle s_2\rangle$). With an umbrella effect, all references are scoped under a single `heap⟨s⟩` handler, hence they all have the same scope variable `s` in their type `ref⟨s⟩`.

3.2.5 Design Trade-offs. In this section, we started with plain named handlers (Section 3.1), introduced scoped effects (Section 3.2.2), and outlined two ways of combining named handlers and scoped effects (Sections 3.2.3 and 3.2.4). These combinations demonstrate the trade-offs in the design space of named handlers.

The first design trade-off is between plain and scoped named handlers. Scoped named handlers are safer in that they avoid the name escaping problem by taking inspirations from monadic encapsulation. On the other hand, plain named handlers are easier to use since handler names do not carry a scope variable in their type. Therefore, language designers may choose to implement one of them or both, depending on what they prioritize. We implement both variations in Koka. Note that, in the unscoped setting, using a handler name outside of its scope results in an exception,

$$
\begin{array}{llcl}
\text{Expression} & e & ::= & v \mid e\ e \mid e\ [\sigma] \\
 & & \mid & \mathsf{handle}^{\epsilon}_{m}\ h^{\ell^{\eta}}\ e \\
\text{Value} & v, f & ::= & x \mid \lambda^{\epsilon}(x:\sigma).\ e \mid \Lambda\alpha^{\kappa}.\ v \\
 & & \mid & \mathsf{handler}^{\epsilon}\ h^{\ell} \\
 & & \mid & \mathsf{perform}^{\epsilon}\ op\ \overline{\sigma} \\
 & & \mid & \mathsf{perform}^{\epsilon}\ op\ \overline{\sigma}\ v \\
 & & \mid & (m,\ h^{\ell^{\eta}}) \\
\text{Handler} & h & ::= & \{\ \overline{op_i \mapsto f_i}\ \} \\
 & & & \\
\text{Type} & \sigma & ::= & \alpha^{\kappa} \mid c^{\kappa} \mid \sigma_1\ \sigma_2 \mid \sigma \rightarrow \epsilon\ \sigma \mid \forall\alpha^{\kappa}.\ \sigma \mid \mathsf{ev}\ \ell^{\eta} \\
\text{Effect row} & \epsilon & ::= & \langle\rangle \mid \langle \ell^{\eta} \mid \epsilon \rangle \\
\text{Kind} & \kappa & ::= & * \mid \kappa \rightarrow \kappa \mid \mathsf{lab} \mid \mathsf{eff} \mid \mathsf{S} \\
 & & & \\
\text{Term context} & \Gamma & ::= & \varnothing \mid \Gamma,\ x:\sigma \\
\text{Effect context} & \Sigma & ::= & \{\ \overline{\ell_i\ :\ sig^{\ell_i}}\ \} \\
\text{Effect signature} & sig^{\ell} & ::= & \{\ \overline{op_i\ :\ \forall\overline{\alpha_i^{\kappa_i}}.\ \sigma_i \rightarrow^{\ell^{\eta}} \sigma_i'}\ \}
\end{array}
$$

Fig. 6. Syntax of System $\mathsf{F}^{\epsilon+\mathsf{sn}}$

and the programmer is responsible for handling such exceptions. In addition to the implementation, we give a formalization of a calculus with plain named handlers in the appendix of the technical report [Xie et al. 2021].

The second design trade-off is between scoped effects and umbrella effects. We have seen that umbrella effects are more expressive as they allow dynamic allocation of handlers sharing the same scope variable. However, their expressive power also comes with the price of extra complexity in the formalization, in particular to ensure type safety. Again, language designers may make the decision of which to implement. In Koka, both variations are supported (Section 6).

4 SYSTEM $\mathsf{F}^{\epsilon+\mathsf{sn}}$: NAMED HANDLERS WITH SCOPED EFFECTS

In this section, we present the formalization of System $\mathsf{F}^{\epsilon+\mathsf{sn}}$, a calculus of named handlers and scoped effects outlined in Section 3.2.3.

4.1 Syntax

Figure 6 defines the syntax of System $\mathsf{F}^{\epsilon+\mathsf{sn}}$. The system is explicitly typed, and includes internal forms generated during evaluation (highlighted in gray). Below, we detail each syntactic category.

Expressions and Values. Expressions and values include variables, abstractions and applications for terms and types[5]. 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, the requirement validates a type-erasure based semantics [Xie et al. 2020]. Operation performing takes the form $\mathsf{perform}^{\epsilon}\ op\ \overline{\sigma}\ e_1\ e_2$, where op is an operation name, $\overline{\sigma}$ is a sequence of types (used to instantiate the type variables in the signature of op), a handler name e_1, and an operation argument e_2. Among the four arguments, e_1 and e_2 are passed via the application rule; the partially applied form $\mathsf{perform}^{\epsilon}\ op\ \overline{\sigma}\ v$ is considered as a value.

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 f (i.e., a thunked computation), handles the operations in effect ℓ that are performed by f. Here, we assume that h^{ℓ} has exactly one clause

[5]We do not include constants (e.g., integers) in the formal systems, but we assume their existence in examples.

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

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

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

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

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

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

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

(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. 7. Typing Rules of System $\mathsf{F}^{\epsilon+\mathsf{sn}}$.

for each operation of effect ℓ[6]. The expression form $\mathsf{handle}^{\epsilon}_{m}\ h^{\ell^{\eta}}\ e$ represents a specific instance of a handler, which is internally generated during evaluation (see Section 4.3). The role of this construct is to associate the label ℓ with a scope η, and to pair it with a marker m, which serves as an identifier of the handler. We regard a marker-handler pair $(m,\ h^{\ell^{\eta}})$ as a handler name, and call it an *evidence*. In principle, the marker m itself is sufficient as a handler name, but as we will show in Section 6, this representation of a marker-handler pair establishes a close connection between handler names and the concept of evidence [Xie et al. 2020; Xie and Leijen 2021] already supported in the implementation of Koka.

[6]For simplicity, we do not include a return clause in handlers, which is often found in an effect handler calculus [Bauer and Pretnar 2014 2015]. This does not cause loss of expressiveness, as one can always integrate the computation of the return clause into the action.

Types and Kinds. Types include type variables α^κ, type constructors c^κ of kind κ (e.g., int^*, $list^{* \to *}$), type applications $\sigma_1\ \sigma_2$, function types $\sigma \to \epsilon\ \sigma$, quantified types $\forall\alpha^\kappa.\ \sigma$, and the evidence type ev ℓ^η, which is inhabited by evidence of the form $(m,\ h^{\ell^\eta})$. Function types $\sigma_1 \to \epsilon\ \sigma_2$ has three components: an input type σ_1, an output type σ_2, and an effect type ϵ representing 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). As a convention, we use μ and η to denote effect and scope type variables, respectively.

To distinguish among value types (of kind $*$ or $\kappa \to \kappa$), effect labels (ℓ^η : lab), effect rows ($\mu,\ \epsilon$: eff), and scopes (η : S), we use a standard kind system (included in the appendix of [Xie et al. 2021]). For clarity of presentation we do not maintain an explicit kind environment for type variables; instead, as a well-formedness condition, we assume that all occurrences of a type variable α always have the same kind κ (subject to alpha-renaming).

4.2 Typing Rules

Figure 7a gives the typing rules of expressions, values, and handlers. The typing judgment for expressions takes the form $\Gamma\ \vdash\ e : \sigma \mid \epsilon$. This reads: expression e has type σ under context Γ, and may perform operations associated with effect labels in ϵ. The judgment for values has a different form $\Gamma \vdash_{\mathsf{val}}\ v : \sigma$. It does not have an effect component because values are effect-free. In the judgment $\Gamma \vdash_{\mathsf{ops}}\ h : \sigma \mid \ell \mid \epsilon$ for handlers, σ is the return type of handler h, ℓ is the effect label handled by h, and ϵ is the effects to be handled by outer handlers.

We 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 are consistent 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. It extends the original effect ϵ with an additional label ℓ^η, which comes from the signature of the operation being performed. The partially applied form perform$^\epsilon$ $op\ \overline{\sigma}\ v$ is type checked via the applicition rule APP.

Rule HANDLER serves as the elimination rule for effects. It removes the label ℓ^η from the effect of the action, as that effect is handled by the handler. In addition to eliminating effects, the rule plays a key role in ensuring the well-scopedness of handler names. Here, the handler name has an evidence type ev ℓ^η, where η is a scope variable introduced in the action type. Since $\eta \notin \mathsf{ftv}(\epsilon,\ \sigma)$, it is guaranteed that η cannot escape through the return type and effect, which in turn means the handler name cannot escape. In the conclusion, we have a rank-2 type, reflecting the similarity between scoped effects and `runST`.

Lastly, rule HANDLE takes care of an internal expression obtained by reducing handler. Compared to HANDLER, HANDLE is closer to the handler rule in other effect calculi: it simply eliminates the effect ℓ^η handled by h.

By observing the two rules for handlers, we can see that our approach to the name escaping problem relies on the existence and the careful design of the handler construct. If we only had handle, or if we represented actions as a computation, we would not be able to use a rank-2 type to ensure well-scopedness of handler names.

In addition to the typing rules, we define a set of rules for deciding whether two row types are equivalent or not (Figure 7b). Row equivalence is defined by reflexivity, transitivity, commutativity,

$$
\begin{array}{lllll}
\text{Evaluation context} & \mathsf{E} & ::= & \square \mid \mathsf{E}\ e \mid v\ \mathsf{E} \mid \mathsf{E}\ [\sigma] \mid \mathsf{handle}^{\epsilon}_{m}\ h^{\ell^{\eta}}\ \mathsf{E} &
\end{array}
$$

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

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

Fig. 8. Operational Semantics of System $\mathsf{F}^{\epsilon+\mathsf{sn}}$

and head equivalence. The commutativity rule differs from the corresponding rule found in row-based calculi with unnamed handlers [Hillerström and Lindley 2016; Leijen 2017; Xie et al. 2020]. In those calculi, commutativity only applies to distinct labels. For instance, the labels *exn* and *nondet* can be swapped, but *st int* and *st bool* cannot. This is because operations in those calculi are always handled by the innermost handler, which means the order of labels is relevant to typing. In our calculus, commutativity applies to any labels. This is because operations are handled by an arbitrary handler specified by the user, which makes the order of labels irrelevant.

4.3 Operational Semantics

System $\mathsf{F}^{\epsilon+\mathsf{sn}}$ is equipped with a call-by-value, typed semantics defined in Figure 8. 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 ($\longmapsto$) in terms of small-step reduction ($\longrightarrow$).

Among the small-step rules, rules APP and TAPP are standard. Rule HANDLER is unique to our system: it generates a fresh scope variable η and a unique marker m. The scope variable is computationally irrelevant: it is only used to make the semantics fully typed. 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 v 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 op, and apply f to the type instantiations $\overline{\sigma}$, the operation argument v, and the resumption k. Note that for a well-typed program, the type of the operation always matches during reduction (due to the PERFORM rule).

4.4 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 4.1. (*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?

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:

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

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. 2020]:

Definition 4.2. (*Handle-safe Expressions*)
(1) The set of handle-safe expressions includes any well-typed, closed expression if it contains no handle term; (2) Moreover, if e_1 is handle-safe, and $e_1 \longmapsto e_2$, then e_2 is handle-safe.

Notably, (2) implies that handle-safe expressions still allow occurrences of handle, but only ones that are reduced from handler.

It turns out that the progress theorem holds for handle-safe expressions. That is, if we start evaluation from a handle-safe expression (or equivalently, a user-written program), we will never get stuck.

Theorem 4.3. (*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. 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 4.4. (*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 $m_1\ \neq\ m_2$.

5 SYSTEM $\mathsf{F}^{\epsilon+u}$: 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 3.2.4. Here we focus on the formalization of effect-related constructs, which is different from System $\mathsf{F}^{\epsilon+\mathsf{sn}}$.

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

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

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

(a) Scoped Effects (l) with Unnamed Handlers

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

$$\frac{\Gamma\ \vdash_{\mathsf{ops}}\ h\ :\ \sigma \mid \ell \mid \epsilon}{\Gamma\ \vdash_{\mathsf{val}}\ \mathsf{handler}^{\epsilon}\ h^{\ell}\ :\ (\mathsf{ev}\ \ell^{\eta} \rightarrow \epsilon\ \sigma) \rightarrow \epsilon\ \sigma}\ [\text{N-HANDLER}]$$

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

(b) Named Handlers (ℓ) under Scoped Effects (l)

Fig. 9. Typing Rules of System $\mathsf{F}^{\epsilon+u}$

5.1 Syntax

The syntax of System $\mathsf{F}^{\epsilon+u}$ is basically the same as System $\mathsf{F}^{\epsilon+\mathsf{sn}}$. The only difference is that we classify effect labels into two categories: those handled by a named handler (ℓ) and those declared as scoped effects (l). For example, in the case of the heap example in Figure 5, the `ref` and `heap` labels are represented as *ref* and *heap*, respectively (remember that `heap` is scoped but `ref` is not). We make this distinction because the typing of effect-related constructs depends on which category an effect label belongs to, as we will see shortly.

5.2 Typing

In Figure 9a, we define the typing rules for scoped (umbrella) effects and their handlers. To highlight the idea of umbrella effects, here we restrict their handlers to be *unnamed*; extending them with named handlers can be done easily. Since handlers are unnamed, 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 $\mathsf{F}^{\epsilon+\mathsf{sn}}$, but they now take a unit value instead of an evidence due to the absence of names. A handle construct simply eliminates the effect l^{η} to be handled (rule U-HANDLE).

In Figure 9b, we give the typing rules for named handlers under scoped effects. 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. This corresponds to the design of the `ref` effect from the heap example: the `get` and `set` operations produce a `heap` effect, which serves as the umbrella of `ref`. Since there is no ℓ effect, handlers do not discharge the effect of their action

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

(a) Scoped Effects (l) with Unnamed Handlers

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

(b) Named Handlers (ℓ) under Scoped Effects (l)

Fig. 10. Operational Semantics of System $\mathsf{F}^{\epsilon+u}$

(rules N-HANDLER and N-HANDLE), but they keep the effect l^{η} scoped under η, which is the scope of its umbrella, e.g., the scope of `heap`.

5.3 Operational Semantics

As with the typing rules, we have two sets of rules defining the operational semantics (Figure 10). The first three rules take care of unnamed handlers for scoped effects. Rule (*u-handler*) 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 performed 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 deal with named handlers under scoped effects. 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, representing the name of the handler. Rules (*n-return*) and (*n-perform*) remain the same as the corresponding rules in System $\mathsf{F}^{\epsilon+\mathsf{sn}}$.

5.4 Type Soundness

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

Theorem 5.1. (*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. In fact, the heap example in Section 2.3.2 is type-safe.

To see why this is the case, recall that the reference effect `ref⟨s⟩` carries a scope variable `s`. This means the effect must be in the scope of the umbrella effect `heap⟨s⟩`. Assuming the reference handler `ref` is private, the heap handler `heap` is the only place where a new reference handler may be generated. When an operation in `ref⟨s⟩` is performed, it produces the `heap⟨s⟩` effect, hence in a well-typed program, the operation performing must be surrounded by a handler for `heap⟨s⟩`. Having

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

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

Fig. 11. Selected Rules for $\mathsf{F}^{\epsilon+u}$ with the Umbrella Witness and the Resume Effect

a heap⟨s⟩ handler further implies the existence of a ref⟨s⟩ handler, because the latter is pushed right above the former when the reference is created.

Unfortunately, general umbrella effects may result in accidental name escaping. To see how this could happen, suppose the reference handler ref in the heap example is accessible to the programmer. In that case, it is possible to write the following program:

```
fun main()
  with r <- ref(1, id)
  r.get()   // error
```

Here, the ref function is called without being surrounded by a heap handler (which we disallowed before by making ref private). As the function is passed the identity function id as the action, it returns the generated handler name. The name is then used to perform the get operation, which naturally fails as there is no matching ref handler. Nevertheless, the program would be judged well-typed by System $\mathsf{F}^{\epsilon+u}$, because named handlers are not scoped in this system, and performing an operation on a named handler does not produce any effects.

As a different example, let us assume that heap has another operation bad. Now, consider the following program:

```
fun heap-bad(action)
  with handler
    newref(init){ ref(init, resume) }
    bad(){ resume }
  action()
```

```
fun main()
  fun b()
    with heap-bad
    with r <- newref(1)
    bad()
    r.get()
    (fn(){ () })
  (b())()   // stuck
```

When the bad operation is performed during evaluation of b(), it is handled by the heap-bad handler. The handler returns the resumption resume, which is essentially `fun(){ with heap-bad; r.get() }`. This function is then returned through the ref handler for r, and invoked in (b())(), yielding `with heap-bad; r.get()`. As we can see, the get operation is performed under a heap handler, but no longer under the ref handler for r. This means the program is stuck at this point.

Given the above examples, the reader might wonder: is there any way to make System $\mathsf{F}^{\epsilon+u}$ type-sound? The answer is "yes": we can restore the progress property by equipping the calculus with two restrictions. The restrictions are:

- A named handler for effect ℓ can only be used inside a handler for its umbrella effect l.
- A handler of an umbrella effect cannot return the resumption.

It is not hard to see that the original heap example (without bad) satisfies both restrictions. Specifically, the named handler ref is only used under the heap handler, and the handler heap of the umbrella effect does not return resume. We can also easily see that the modified heap example (with bad) is rejected by the second restriction.

To implement the above restrictions, we can either syntactically constrain the use of handlers and resumptions, or augment the typing rules with additional requirements. Here, we take the latter approach. In Figure 11, we present two rules selected from the carefully crafted variant of System $\mathsf{F}^{\epsilon+u}$. The system relies on two concepts that are not presented in the original System $\mathsf{F}^{\epsilon+u}$: the *umbrella witness* and the *resume effect*. These are both internal constructs, and are highlighted in gray in the figure.

Among the new concepts, the umbrella witness is used to implement the first restriction. As can be seen from rule N-HANDLER, whenever we create a named handler for an ℓ^{η} effect, we need an umbrella witness of type umb $\eta\ \epsilon\ \sigma$. As stated in rule U-OPS, such a witness is (and can only be) issued by the operation clauses of an umbrella effect handler. These ensure only umbrella effect handlers can create new named handlers. With the additional components ϵ and σ carried by the umbrella witness, we can also be sure that $\eta \notin \mathsf{ftv}(\epsilon, \sigma)$.

The other concept, namely the resume effect, is used to implement the second restriction. From rule U-OPS, we can see that every operation clause of an umbrella effect handler must have a resumption effect r^{η} in its return effect. Also from rule U-OPS, we can deduce that this effect can only be produced by calling the resumption. These make it impossible to return the resumption from an umbrella effect handler.

By combing the umbrella witness and the resume effect, we can prove progress for handle-safe expressions with general umbrella effects. We invite the interested reader to visit the appendix of the technical report [Xie et al. 2021] for the complete specification of the restricted System $\mathsf{F}^{\epsilon+u}$ and its soundness proof.

Theorem 5.2. (*Progress of Handle-safe System* $\mathsf{F}^{\epsilon+u}$)
If $\varnothing\ \vdash\ e_1\ :\ \sigma \mid \langle\rangle$ 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.

As in System $\mathsf{F}^{\epsilon+\mathsf{sn}}$, we can further prove the uniqueness of names for handle-safe expressions.

Theorem 5.3. (*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 $m_1\ \neq\ m_2$.

6 IMPLEMENTATION

We have implemented named handlers and scoped effects in the Koka programming language, supporting the two combinations discussed in Sections 4 and 5. In this section, we give an overview of the Koka compiler, and provide details on how Koka compiles unnamed and named handlers.

Overview of the Koka Compiler. 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. 2021]. To support (plain) effect handlers and first-class resumptions in C, the compiler uses two transformations. The first one targets a calculus bsed on the *evidence passing semantics* [Xie et al. 2020; Xie and Leijen 2021]. Here, every function receives the current *evidence vector*, which is a sequence of marker-handler pairs. This eliminates runtime search for matching handlers, leading to better performance. The second transformation targets a polymorphic lambda calculus à la System F. Here, control transfer and context capture are realized by a standard multi-prompt delimited control monad [Dyvbig et al. 2007; Gunter et al. 1995; Xie and Leijen 2021]. This allows us to implement the non-local behavior of effect handlers without needing any special runtime support (and can thus compile to portable C99 code).

Finally, Koka supports impredicative higher-rank polymorphism and performs type inference using the HMF system [Leijen 2008]. As such, it can directly express rank-2 types used in scoped effects (Section 3.2.2). The system also reduces the programmers' burden to write down explicit scope variable annotations when type inference is possible.

Compiling Unnamed Handlers. Let us first look at how regular (unnamed) handlers and operations are evaluated under the multi-prompt evidence passing semantics. Below are the evaluation rules from [Xie and Leijen 2021, Figure 1]:

$$\begin{array}{lll}
(\mathit{app}) & (\lambda x.\ e)\ v & \longrightarrow e[x := v] \\
(\mathit{handler}) & \mathsf{handler}\ h\ v & \longrightarrow \mathsf{prompt}\ m\ h\ (v\ ()) \quad \text{with unique } m \\
(\mathit{return}) & \mathsf{prompt}\ m\ h\ v & \longrightarrow v \\
(\mathit{yield}) & \mathsf{prompt}\ m\ h\ [\mathsf{yield}\ m\ f] & \longrightarrow f\ (\lambda x.\ \mathsf{prompt}\ m\ h\ \mathsf{E}[x]) \\
(\mathit{perform}) & w\ \vdash\ \mathsf{perform}\ op^l\ v & \longrightarrow \mathsf{yield}\ m\ (\lambda k.\ f\ v\ k) \quad (m, h)\ =\ w.l\ \land (op \mapsto f)\ \in h
\end{array}$$

Rule (*handler*) generates a fresh marker m and installs a new prompt. Rule (*yield*) uses a marker to directly yield to a prompt, while capturing the evaluation context E in a resumption. All operations are evaluated under an implicit evidence vector w, containing the current marker-handler pairs (m, h) for each prompt in the context. Rule (*perform*) finds an evidence (m, h) in the evidence vector, extracts the operation implementation f for op, and continues with a yield to the prompt m.

Due to the explicit markers and evidence vectors, there is no implicit search to the innermost handler anymore as in the standard operational semantics for effect handlers. In particular, each handler has a unique marker and is found from the evidence vector and yielded to directly.

As a final remark, we elaborate on the benefit of representing evidence as a marker-handler pair. As rule (*perform*) shows, we can use such a pair to construct the continuation $(\lambda k.\ f\ v\ k)$ that waits for the resumption to be built up. But this is not the only thing we can do: we can further perform *tail-resumptive optimization* [Xie and Leijen 2021], a technique for avoiding expensive yielding by evaluating a tail-resumptive operation in place. A tail-resumptive operation is of form $\lambda x.\ \lambda k.\ k\ e$ where $k \notin \mathsf{fv}(e)$. As an example, the reader handler from Section 2.1 is tail-resumptive. For tail-resumptive operations, the rule (*perform*) does not need to generate any yield[7]:

$$w\ \vdash\ \mathsf{perform}\ op^l\ v \longrightarrow (\lambda x.\ e)\ v \qquad (m, h)\ =\ w.l\ \land (op \mapsto \lambda x.\ \lambda k.\ k\ e)\ \in h\ \land k \notin \mathsf{fv}(e)$$

It has been shown that tail resumptive optimization can improve performance. We refer interested readers to Xie and Leijen [2021] for details.

Compiling Named Handlers. Since all handlers are identified with their marker, it turns out to be quite easy to support named handlers in this framework. Recall from Section 4.1 that we represent handler names as a pair of a marker and a handler – this is exactly the evidence in the evidence vectors w. The named handler extension thus requires no new mechanisms; we can translate named handlers to the existing evidence calculus in a straightforward way. Specifically, we simply do *not* insert evidence for named handler in the evidence vector w, and pass it instead explicitly as a first-class value to perform. The transition rules for named handlers and operations are:

$$\begin{array}{lll}
(\mathit{nhandler}) & \mathsf{handler}\ h^\ell\ v & \longrightarrow \mathsf{prompt}\ m\ \varnothing\ (v\ (m, h^\ell)) \quad \text{with unique } m \\
(\mathit{nperform}) & \mathsf{perform}\ op\ (m, h^\ell)\ v & \longrightarrow \mathsf{yield}\ m\ (\lambda k.\ f\ v\ k) \qquad (op \mapsto f)\ \in h^\ell
\end{array}$$

There are two differences from the unnamed counterparts. First, in the (*nhandler*) case, we directly pass the evidence to the action v, and use an empty handler ($\varnothing$) to prevent it from being inserted in the evidence vector. Thus, the action of a named handler receives a name (m, h^ℓ), while an action of an unnamed handler receives unit. Second, in the (*nperform*) case, we obtain the evidence directly as an explicit argument, without needing the evidence vector. This change comes from the fact that an operation handled by a named handler explicitly receives a name, while an operation

[7]To ensure that the operations in e are handled by the correct handlers, the actual (*performt*) rule [Xie and Leijen 2021] is slightly more complex and evaluates to $(\lambda x.\ \mathsf{under}\ l\ e)\ v$, where under adjusts the evidence vector for operations performed in e. It has been showed that tail-resumptive optimization is semantics preserving.

handled by an unnamed handler does not. This is also the point where things could go wrong: if a named handler escaped its scope, yield m would find no matching prompt m in the evaluation context. Notably, representing handler names as evidence also allows us to enjoy tail-resumptive optimization for named handlers: in rule (*nperform*), if f is tail-resumptive, then we can evaluate f in place without explicit yielding.

As a result, there were very few changes that needed to be made to the Koka runtime system and compiler – all internal translations already use "names" as evidence. The implementation is also consistent with the formalization presented in this paper, except for the following differences:

- In addition to named handlers with scoping, the Koka implementation 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 discussed in Section 5.4. Therefore, any umbrella operations also induce an exception effect, which is raised if an umbrella handler escapes its scope. We feel adding the exception effect is a reasonable implementation trade-off, 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.

7 FURTHER EXAMPLES

In this section, we present larger examples of named handlers. We use the examples to demonstrate how first-class handler names are useful in realistic applications.

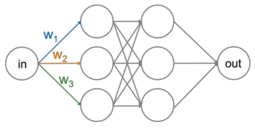
7.1 Neural Networks

As the first example, we show how to implement a neural network based on gradient descent using named handlers. Broadly speaking, a neural network is a statistical model that approximates functions based on training data (Figure 12a). It consists of multiple layers of artificial neurons, where the first layer carries inputs and the last layer carries outputs. To propagate values from one layer to the next, the network computes the *weighted* sum of values and adjusts the result by adding a *bias* value. After obtaining the output values, the network compares them with the expected values (i.e., the *ground truth*), and updates the weights and biases using gradient descent.

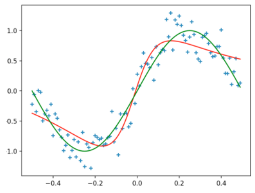
In Figure 13a, we present our implementation of neural layers. The key idea is to represent the matrix of weights and biases for each neural layer using named handlers, and then build a neural network as a list of handler names. The effect `layer` is similar to the `ref` effect from Section 2.3.2, in that it has operations for accessing, setting, and updating weights and biases. Each weight and bias is a `variable` data structure, consisting of a vector of values and a vector of the gradients of the error function at specific values.

In Figure 13b, we implement a program that approximates the sine function using the `layer` effect. First, we create a neural network `net` that has two hidden layers (layers other than the input and output layers). This is done by installing three named handlers of the `layer` effect and putting the names into a list. We then start iterating the learning process. At each iteration, we compute the gradient of the error function based on *backpropagation* (in the style of [Sigal 2021; Wang et al. 2019]) and updates the weights and biases. Here, the `backprop` function is an unnamed handler that serves as an interpreter of arithmetic operations designed specifically for automatic differentiation. As the result of learning, we obtain the red curve shown in Figure 12b.

Implementing a neural network this way has several advantages. First, the use of effect handlers allows us to seamlessly combine the `layer` effect with other useful effects. In the full implementation,



(a)



(b)

Fig. 12. Neural Network and Result of Learning (green line: actual sine function, blue dots: training data, red line: learned sine function)

```
named effect layer⟨a⟩
  fun get-weight(): a
  fun set-weight(w: a): ()
  fun get-bias(): a
  fun set-bias(b: a): ()
  fun sgd(lr: double): ()

fun layer(i, o, action)
  var w := init-w(i, o)
  var b := init-b(o)
  with h <- named handler
    fun get-weight()   { w }
    fun get-bias()     { b }
    fun set-weight(ws) { w := ws }
    fun set-bias(bs)   { b := bs }
    fun sgd(lr)
      val t₁ = w.data - lr * !w.grad
      val t₂ = b.data - lr * !b.grad
      w := variable(t₁)
      b := variable(t₂)
  action(h)
```

(a) Layers of Neural Networks

```
fun main()
  // num of neurons in input layer
  val i = 1
  // num of neurons in hidden layers
  val h = 3
  // num of neurons in output layer
  val o = 1
  val lr = 0.2 // learning rate
  val iters = 5000 // num of iterations

  with l₁ <- layer(i, h)
  with l₂ <- layer(h, h)
  with l₃ <- layer(h, o)

  val net = [l₁,l₂,l₃]
  var errs := []

  for(1,iters) fn(cnt:int)
    val loss = backprop { ... }
    errs := Cons(loss.data.at(0,0), errs)
    net.foreach( fn(l) l.sgd(lr) )
```

(b) Learning the Sine Function (excerpt)

Fig. 13. Example of Neural Networks

we use several built-in effects including exceptions and divergence, and we can easily extend the implementation with other effects such as tracing and backtracking. Second, effect handlers make it easy to change the behavior of learning. For instance, if we wish to add *momentum* to gradient descent, all we need is to define a new handler; there is no need to change the network itself. Third, named handlers provide a convenient and reliable way of distinguishing between the weights and biases of different layers. As a comparison, the implementation by Wang et al. [2019] uses object-oriented features to simulate names, leading to a discrepancy between the formalization and implementation. Lastly, the first-class status of handler names enables us to treat a neural network simply as a list.

There is also a non-trivial design decision behind the implementation. We chose to use plain named handlers to make the implementation as simple as possible. The absence of scopes results in loss of static safety guarantee, but it allows us to put handler names inside a homogeneous list (as in `[l₁,l₂,l₃]`), and we can still make a safety argument as handler names are obviously used within their scope.

```
type utype⟨s⟩                               // types with (scoped) unification variables
  UVar( v : variable⟨s⟩ )                   // unification variables
  UCon( tag : string )                       // type constructors
  UApp( t₁ : utype⟨s⟩, t₂ : utype⟨s⟩ ) // type applications

type ntype                           // complete types
  Con( tag : string )                // type constructors
  App( t₁ : ntype, t₂ : ntype ) // type applications

scoped effect subst⟨s⟩      // an umbrella effect subst
  fresh() : variable⟨s⟩     // generate a fresh unification variable

named effect variable⟨s⟩ in subst⟨s⟩  // a named variable effect under subst
  fun get() : maybe⟨utype⟨s⟩⟩          // get the variable's current type
  fun resolve( tp : utype⟨s⟩ ) : ()    // resolve the unification variable

fun subst(action:forall⟨s⟩ ()→ ⟨subst⟨s⟩,pure|e⟩ a) : ⟨pure|e⟩ a  // a handler for subst
  with fresh() with-var(resume)
  action()

fun with-var(action)  // dynamically install a named handler for variable
  var mtp := Nothing
  with v <- named handler
    fun get() mtp
    fun resolve(tp)
      match mtp
        Nothing → mtp := Just(tp)
        Just    → throw("already resolved")
  action(v)

// unify two types under a substitution
fun unify( tp₁ : utype⟨s⟩, tp₂ : utype⟨s⟩ ) : ⟨subst⟨s⟩,pure⟩ utype⟨s⟩
  match (tp₁,tp₂)
    (UCon(tag₁), UCon(tag₂)) | tag₁ == tag₂ → tp₁
    (UVar(v₁),_) → match v₁.get()
                     Nothing →
                       if occur(v₁,tp₂) then throw("occurs check")
                       v₁.resolve(tp₂)
                       tp₂
                     ...
    ...
    _ → throw("cannot unify types")

// resolve all unification variables to make a complete type
fun resolve-all( tp : utype⟨s⟩ ) : ⟨subst⟨s⟩,pure⟩ ntype
  match tp
    UCon(tag) → Con(tag)
    ...

pub fun example() : pure ntype
  with subst
  val a = fresh()
  val b = fresh()
  val tp₁ = mkfun(UVar(a),UVar(a))
  val tp₂ = mkfun(UVar(b),mklist(mkint()))
  unify(tp₁,tp₂).resolve-all
```

Fig. 14. A unification algorithm (excerpt)

7.2 Unification

As another example of named handlers, we look at defining a unification algorithm using named handlers. The key idea here is to use an umbrella effect to dynamically generate named handlers to represent fresh unification variables, where a unification variable can be resolved with a type. The scope variable associated with the umbrella effect delimits the scope of such unification variables, making sure that all unification variables are resolved after unification, and isolating the stateful substitution behind a pure functional interface.

We present the program in Figure 14. We distinguish between unifiable types `utype` and complete types `ntype`. In unifiable types, a type `variable` is a handler name which can be used under a umbrella `subst` handler, and is created using the `fresh` operation. Here we rely on the first-class nature of named handlers, as a handler name `variable` occurs in the `UVar` constructor. Each variable has two operations: we can `get` the current value as a `maybe⟨utype⟨s⟩⟩` which is `Nothing` if it is not yet resolved, and we can use `resolve` once to resolve a variable to its type.

The function `unify` unifies two types under a `subst` effect, and resolves a `variable` when necessary. Once unification is finished, the `resolve-all` function resolves all unification variables and returns a complete type `ntype`. We give a pure `example` function showing how to perform unification. Notice how the umbrella handler `subst` ensures statically that all unification variables must be resolved, since the scope `s` cannot escape. If the system contains polymorphic types, we can also define a generalization function that generalize all unsolved unification variables to universal quantifiers.

Let us turn to the advantages and design decisions of our implementation. Similar to the neural network example, our unification algorithm benefits from the composability of different effects: an umbrella effect `subst`, a named effect `variable`, and a built-in effect `pure` corresponding to Haskell's notion of purity (i.e., exceptions and divergence). The algorithm also shows the usefulness of first-class handler names: observe how names are passed as constructor arguments and extracted via pattern matching. Furthermore, the algorithm relies on *dynamic* generation of named handlers for the unifiable `variable`'s, which requires an umbrella effect.

8 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 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. [2020], 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, and thus maintain safety guarantees.

Named Handlers. The name escaping challenge described above has previously been addressed by two studies [Biernacki et al. 2019; Zhang and Myers 2019]. Among them, Biernacki et al. [2019] constrain the use of handler names in the following way. At the level of syntax, they introduce a new binder for names, as well as a corresponding application form (Figure 15). They then enforce second-class use of handler names by restricting the position where names can appear. At the level of typing, they use a typing judgment that carries an additional environment, containing names that are currently in scope. They also lift handler names to the type level and make them explicit in effect types. By designing the calculus in this way, they obtain the desired property: if a program is judged well-typed and effect-free, then the program does not get stuck at runtime.

The other related study, which is due to Zhang and Myers [2019], treats handler names as a capability to access a stack region. Their approach to well-scopedness is similar to that of Biernacki et al. [2019]. First, they impose a syntactic restriction on the use of handler names. Second, they

$$\begin{array}{rcl} v & ::= & x \mid \lambda x.\ e \mid \Lambda\alpha.\ e \mid \lambda\!\!\lambda a.\ e \\ e & ::= & v \mid e\ e \mid e\ a \mid \mathsf{perform}_a\ op\ v \mid \mathsf{handler}_a\ h\ e \end{array}$$

Fig. 15. Syntax of Biernacki et al.

use a simple form of region typing [Tofte and Talpin 1997] to keep track of handler names. The resulting calculus has proven sound with respect to contextual refinement via a logical relation.

While the existing approaches solve the name escaping problem, there are some limitations as well. First, the non-standard binding and typing mechanisms make it harder to incorporate named handlers into an existing type checker. In particular, one has to carefully consider the interaction with other type system features (such as higher-rank types). Second, the syntactic restrictions make it impossible to use handler names as first-class values. That is, one cannot return a handler name from a function (e.g., $\lambda h.\ h$), pass a name to a datatype constructor (e.g., Just h), or create a tuple of names (e.g., $(h_1,\ h_2)$). This limits the expressiveness of the calculus and they cannot encode the examples shown in Section 7 for example.

Effect Instances. In an old version of the Eff language [Bauer and Pretnar 2014 2015], there was the concept of *effect instances*, which essentially play the same role as named handlers. Effect instances in Eff are first-class, and can be created dynamically during evaluation. However, the dynamic creation feature is omitted in the formalization, because it lead to "significant complications, both in the effect system and in semantics". Also, as with [Biernacki et al. 2019] and [Zhang and Myers 2019], the type system is not fully stratified as instance names can appear in effect types.

Multi-prompt Control Operators. The notion of named handlers is also closely related to multi-prompt delimited control operators [Gunter et al. 1995; Kiselyov 2012; Shan 2007; Sitaram 1993]. When programming with these operators, one can specify the intended association between the control operator and the delimiter through *prompt tags*. An implication of this connection is that prompt tags suffer from the same problem with handler names: without special care, prompt tags may escape their scope during 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. As we discussed in Section 3.2.2, our approach to well-scopedness of handler names draws inspiration from Haskell's `runST`. Semmelroth and Sabry [1999] incorporate a direct-style counterpart of `runST` into a subset of ML, and establish a relationship between monadic encapsulation and effect masking. Timany et al. [2017] design a logical relations model of a higher-order functional programming language featuring a Haskell-style ST monad type with `runST`, and prove the encapsulation ability obtained by rank-2 types.

Dynamic References as Algebraic Effects. An algebraic-effect-based implementation of dynamically created reference cells has previously given by Kiselyov and Sivaramakrishnan [2017]. Their implementation is in OCaml, and uses a library for multi-prompt delimited control operators [Kiselyov 2012]. Like us, they treat creation of a new reference cell as an effect. Unlike us, they do not prevent the escaping of references, since OCaml does not have effect typing.

9 CONCLUSION

In this paper, we explored the design space of named effect handlers, where handler names are first-class and well-scoped. The first-class status is obtained by using regular lambdas to bind names, while the well-scoped guarantee is gained by assigning handlers a rank-2 type. We implemented named handlers in the Koka language, and demonstrated their usefulness through examples. We look forward to investigating new programming techniques enabled by named effect handlers and first-class handler names.

REFERENCES

Andrej Bauer, and Matija Pretnar. 2014. An Effect System for Algebraic Effects and Handlers. *Logical Methods in Computer Science* 10 (4). https://doi.org/10.1007/978-3-642-40206-7_1.

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

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. https://doi.org/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. https://doi.org/10.1145/3371116.

Eli Bingham, Jonathan P Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul Szerlip, Paul Horsfall, and Noah D Goodman. 2019. Pyro: Deep Universal Probabilistic Programming. *The Journal of Machine Learning Research* 20 (1). JMLR. org: 973–978. https://doi.org/10.48550/arXiv.1810.09538.

Oliver Bračevac, Nada Amin, Guido Salvaneschi, Sebastian Erdweg, Patrick Eugster, and Mira Mezini. 2018. Versatile Event Correlation with Algebraic Effects. *Proceedings of the ACM on Programming Languages* 2 (ICFP). ACM: 67. https://doi.org/10.1145/3236762.

Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. 2020. Effects as Capabilities: Effect Handlers and Lightweight Effect Polymorphism. *Proc. ACM Program. Lang.* 5. ACM. https://doi.org/10.1145/3428194.

Lukas Convent, Sam Lindley, Conor McBride, and Craig McLaughlin. Jan. 2020. Doo Bee Doo Bee Doo. *In the Journal of Functional Programming*, January. https://doi.org/10.1017/S0956796820000039.

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.

R Kent Dyvbig, Simon Peyton Jones, and Amr Sabry. 2007. A Monadic Framework for Delimited Continuations. *Journal of Functional Programming* 17 (6). Cambridge University Press: 687–730. https://doi.org/10.1017/S0956796807006259.

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. https://doi.org/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. https://doi.org/10.1145/224164.224173.

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. https://doi.org/10.1145/2976022.2976033.

Ralf Hinze. 2003. Fun with Phantom Types. In *The Fun of Programming, Cornerstones of Computing*, edited by Jeremy 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. https://doi.org/10.1145/237721.237731.

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. https://doi.org/10.1017/S0956796816000320.

Oleg Kiselyov. 2012. Delimited Control in OCaml, Abstractly and Concretely. *Theoretical Computer Science* 435. Elsevier: 56–76. https://doi.org/10.1007/978-3-642-12251-4_22.

Oleg Kiselyov, and KC Sivaramakrishnan. Dec. 2017. Eff Directly in OCaml. In *ML Workshop 2016*. https://doi.org/10.48550/arXiv.1812.11664. Extended version.

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

Daan Leijen. Sep. 2008. HMF: Simple Type Inference for First-Class Polymorphism. In *Proceedings of the 13th ACM Symposium of the International Conference on Functional Programming*. ICFP'08. Victoria, Canada. https://doi.org/10.1145/1411204.1411245.

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

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. https://doi.org/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. https://doi.org/10.1145/3240719.3241789.

Daan Leijen, and Erik Meijer. 1999. Domain Specific Embedded Compilers. In *In Proceedings of the 2nd Conference on Domain Specific Languages*, 109–122. Atlanta. https://doi.org/10.1145/331963.331977.

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. https://doi.org/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. https://doi.org/10.1007/3-540-13346-1_15.

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

Gordon D. Plotkin, and John Power. 2003. Algebraic Operations and Generic Effects. *Applied Categorical Structures* 11 (1): 69–94. https://doi.org/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. https://doi.org/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. https://doi.org/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. https://doi.org/10.1016/j.entcs.2015.12.003.

Alex Reinking, Ningning Xie, Leonardo de Moura, and Daan Leijen. 2021. Perceus: Garbage Free Reference Counting with Reuse. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 96–111. PLDI 2021. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3453483.3454032.

Taro Sekiyama, and Atsushi Igarashi. 2019. Handling Polymorphic Algebraic Effects. In *European Symposium on Programming*, 353–380. Springer. https://doi.org/10.1007/978-3-030-17184-1_13.

Miley Semmelroth, and Amr Sabry. 1999. Monadic Encapsulation in ML. In *Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming*, 8–17. ICFP '99. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/317636.317777.

Chung-chieh Shan. 2007. A Static Simulation of Dynamic Delimited Control. *Higher-Order and Symbolic Computation* 20 (4): 371–401. https://doi.org/10.1007/s10990-007-9010-4.

Jesse Sigal. 2021. Automatic Differentiation via Effects and Handlers: An Implementation in Frank. In *The ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM '21)*.

Dorai Sitaram. 1993. Handling Control. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, 147–155. https://doi.org/10.1145/173262.155104.

Asumu Takikawa, T Stephen Strickland, and Sam Tobin-Hochstadt. 2013. Constraining Delimited Control with Contracts. In *European Symposium on Programming*, 229–248. Springer. https://doi.org/10.1007/978-3-642-37036-6_14.

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). https://doi.org/10.1145/3158152.

Mads Tofte, and Jean-Pierre Talpin. 1997. Region-Based Memory Management. *Information and Computation* 132 (2): 109–176. https://doi.org/10.1006/inco.1996.2613.

Fei Wang, Daniel Zheng, James Decker, Xilun Wu, Grégory M Essertel, and Tiark Rompf. 2019. Demystifying Differentiable Programming: Shift/reset the Penultimate Backpropagator. *Proceedings of the ACM on Programming Languages* 3 (ICFP). ACM New York, NY, USA: 1–31. https://doi.org/10.1145/3341700.

Andrew K. Wright, and Matthias Felleisen. Nov. 1994. A Syntactic Approach to Type Soundness. *Inf. Comput.* 115 (1): 38–94. https://doi.org/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. https://doi.org/10.1145/2633357.2633358.

Ningning Xie, Jonathan Brachthäuser, Phillip Schuster, Daniel Hillerström, and Daan Leijen. Aug. 2020. Effect Handlers, Evidently. In *Proceedings of the 25th ACM SIGPLAN International Conference on Functional Programming (ICFP'2020).* ICFP '20. Jersey City, NJ. https://doi.org/10.1145/3408981.

Ningning Xie, Youyou Cong, Ikemori, and Daan Leijen. May 2021. *First-Class Names for Effect Handlers.* MSR-TR-2021-10. Microsoft. `https://www.microsoft.com/en-us/research/publication/first-class-named-effect-handlers`. Updated Oct 2022. Presented at the 8th ACM SIGPLAN Workshop on Higher-Order Programming with Effects (HOPE'21).

Ningning Xie, Youyou Cong, Kazuki Ikemori, and Daan Leijen. Sep. 2022. Artifact for "First-Class Names for Effect Handlers." https://doi.org/10.5281/zenodo.7062933. Also available at `https://hub.docker.com/r/daanx/oopsla22-namedh`.

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. https://doi.org/10.1145/3406088.3409022.

Ningning Xie, and Daan Leijen. Aug. 2021. Generalized Evidence Passing for Effect Handlers: Efficient Compilation of Effect Handlers to C. In *Proc. ACM Program. Lang.*, volume 5. ICFP. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3473576.

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