First-class Names for Effect Handlers

NINGNING XIE, The University of Hong Kong, China
YOUYOU CONG, Tokyo Institute of Technology, Japan
DAAN LEIJEN, Microsoft Research, USA

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

1 INTRODUCTION

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

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

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

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

Our specific contributions can be summarized as follows:

- *Named handlers*: We propose a novel design of named effect handlers, where handler names are first-class, lambda-bound values. In Section 2.3, we give an overview of our design, and illustrate the flexibility of first-class names.
- *Scoped effects*: We next introduce *scoped effects*, which are a useful concept for associating resources to handlers. In Section 2.4, we show that scopes can be handled using standard rank-2 polymorphism [Jones 1996; McCracken 1984]. We then demonstrate that scoped effects allow us

to implement a safe resource interface by modeling *locally isolated state* using scoped effects and *masking* [Biernacki et al. 2017; Convent et al. 2020; Leijen 2014; Wu et al. 2014].
- *Named handlers with scoped effects*: We then solve the name escaping problem of plain named handlers by incorporating scoped effects into the named handler calculus. In Section 2.5, we illustrate the design of the resulting system, focusing on how rank-2 types help us ensure well-scopedness of handler names.
- *Named handlers under scoped effects*: We further support *dynamic* creation of named handlers by incorporating the notion of *umbrella* effects [Leijen 2018]. In Section 2.6, we discuss implementation of reference cells as a motivation for dynamic named handlers, and show that umbrella effects allow us to implement isolated heaps with dynamically created references.
- We have implemented all the named handler variants, as well as locally isolated state, in the Koka programming language [Koka 2019].

2 OVERVIEW

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

2.1 Algebraic Effects

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

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

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

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

An effect signature only defines the type of operations; their interpretation is given separately by a *handler*. A handler takes the form handler $h\ e$, where e is the computation to be handled (which we call an *action*), and h is a list of operation implementations[2]. Each operation implementation is a function $\lambda x.\ \lambda k.\ e'$, where x stands for the argument of the operation, and k represents the delimited continuation within the handler, which can be used to resume the computation surrounding an operation. The following handler handles *ask* by applying k to 42, meaning that a call to *ask* is always interpreted as 42.

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

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

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

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

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

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

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

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

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

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

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

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

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

2.2 Named Handlers

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

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

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

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

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

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

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

As a different approach, Zhang and Myers [2019] adopt implicit naming for the user language and internally elaborate programs to use explicit names. The elaboration forces operations to be

handled by the closest, lexically enclosing handler, enabling modular reasoning in the presence of higher-order functions.

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

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

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

2.3 First-class Names

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

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

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

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

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

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

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

$$\begin{array}{l} reader\ :\ \forall\alpha\ \mu.\ int \to (\mathsf{ev}\ read \to \langle read \mid \mu\rangle\ \alpha) \to \mu\ \alpha \\ reader\ x\ f\ =\ \mathsf{handler}\ \{\ ask \mapsto \lambda().\ \lambda k.\ k\ x\ \}\ f \end{array}$$

We see that the name argument of the action f has type ev *read*, meaning that a read handler is available during evaluation of f. Other parts of the type signature are based on the standard, row-polymorphic effect system [Hillerström and Lindley 2016; Leijen 2005 2014; Xie et al. 2020]. Here we see that action f has effect $\langle read \mid \mu\rangle$, which means it can perform the *read* effect and possibly

more effects, denoted by the polymorphic effect variable μ. The handler construct discharges the *read* effect, hence the final effect is just μ. The empty effect is denoted by the empty row $\langle\rangle$ and is often omitted.

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

2.4 Scoped Effects

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

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

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

Previous studies by Biernacki et al. [2019] and Zhang and Myers [2019] solve the name escaping issue in the following way. First, they expose handler names in the effect information of expressions. For example, Biernacki et al. [2019] would assign $\mathsf{perform}_a\ \mathit{ask}\ v$ a type that mentions effect read_a, which implies the calculus must support a limited form of dependent typing. Second, they augment typing judgments with a separate environment for names to ensure well-scopedness of handler names. However, from a practical point of view, the demand for limited dependent types and a separate name environment makes it difficult to implement handler names in an existing framework. Moreover, the approach cannot be taken when we like to use names are first-class values.

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

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

Here we present a new approach that requires no extensions to the core calculus. The idea is to use standard masking (discussed in Section 2.2) and a regular state handler (given in Section 2.1).

```
type ix = Ix int

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

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

(a) Unsafe interface

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

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

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

(b) Safe interface with scoped effects

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

More specifically, we handle any state-related effects using two handlers: an outer one as the regular state handler, and an inner one for the user-defined effect. The inner handler uses the state provided by the outer h^{st}, while wrapping its action around $\text{mask}^{st\ \sigma}$ to make the state local. For convenience, we define a syntactic sugar (handler $e\ h\ f$), denoting a handler h handling action f with a local state initialized to e[3]:

```
handler e h f ≜ handler h^st (λ_.
                  x ← handler h (λ_. mask^st σ (f ())); λz. x) e
```

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

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

```
tick { tick : () →^tick int }

handler 0 { tick ↦λx. λk. i ← perform get ();
                          perform set (i + 1); k i }
```

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

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

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

[4] We use the keyword type to define a new type, and use $[\,]$, $+\!\!+$, and $v[i]$ to mean the empty list, list append, and list lookup.

not bound to specific handlers. The following program shows how *find* may cause an out-of-bound error at runtime:

```
withvec ( λ_. i ← perform push "hello";
              withvec (λ_. perform find i))  // out of bound
```

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

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

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

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

```
withvec (λ_. i ← perform push "hello";
              withvec (λ_. perform find i))  // statically rejected
```

Note that it is essential to attach a scoped type parameter η to not only the associated index ($ix\ \eta$) but also the effects (vec^η). If we kept *vec* unscoped, we could let resources escape the scope of the handler by returning a lambda that performs *find* on an index captured by that lambda (as now the type of the lambda would not reflect the use of η anymore). If we scope *vec* as vec^η, any use of a resource must be reflected in the type of functions and data types, and the rank-2 polymorphic type of *withvec* prevents any possible resource escaping.

2.5 Combining Scoped Effects and Named Handlers

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

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

```
reader : ∀α μ. int → (∀η. ev read^η → ⟨read^η | μ⟩ α) → μ α
reader = λx. λf. handler { ask ↦ λ(). λk. k x } f
```

[5] Note the difference between polymorphic effects, e.g., *st* α, where the effect *st* α is *parameterized* by α, and scoped effects, e.g., vec^η, where the effect *vec* (without η) is *scoped* by η.

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

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

(a) Mutable references

```
heap { // scoped effect
       newref : α →^{heap^η} (ev (ref α)^η) }

ref { // perform the heap effect
      getref : () →^{heap^η} int
    , setref : int →^{heap^η} () }

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

hp : ∀α. (∀η. () → ⟨heap^η | μ⟩ α) → μ α
hp f =
  handler { // scoped but unnamed handler
    newref ↦ λx. λk. makeref x k
  } f
```

(b) First-class heap using umbrella effects

Fig. 2. Mutable references and first-class heap

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

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

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

2.6 Scoping Named Handlers under Umbrella Effects

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

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

```
tworef : (∀η1 η2. ev ref^η1 → ev ref^η2 → ⟨ref^η1 | ref^η2 | μ⟩ α) → μ α
tworef f = makeref 1 (λr1. makeref 2 (λr2. f r1 r2))
```

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

While *makeref* correctly implements the operations *getref* and *setref*, it does not allow us to use reference cells as flexibly as in a calculus with native support for heap references. For instance, we must use references under their own *ref* handler, which means we need to *statically* instantiate as many handlers as the number of distinct references required. Moreover, when references are

implemented via scoped effects, different reference cells cannot be put in a homogeneous list as in $[r_1, r_2]$, since their types carry different scope variables (η_1 and η_2 respectively). Finally, there is no means to create fresh reference cells *dynamically* at runtime.

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

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

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

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

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

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

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

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

⟼* 3
```

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

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

2.7 Desirable Properties

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

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

Well-scopedness of handler names. We show that names can never escape the scope of their handler. This property is subsumed by the *type soundness* theorem, which can be proved by showing the *preservation* and *progress* properties [Wright and Felleisen 1994]. Preservation guarantees that the well-typedness of expressions is preserved by reduction. Progress ensures that a well-typed expression always takes a step, which, in our context, means an operation performing always finds the corresponding named handler. One thing to note here is that type soundness of umbrella effects needs extra care. In particular, while the interface in Figure 2b is type-safe, general umbrella effects may result in accidental escaping of names. In our technical report [Xie et al. 2021], we further discuss two type-theoretic restrictions on umbrella effects for ensuring type soundness, without losing the expressiveness required to encode first-class heaps.

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

REFERENCES

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Ningning Xie, Youyou Cong, and Daan Leijen. May 2021. *First-Class Named Effect Handlers*. MSR-TR-2021-10. Microsoft Research.

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

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