Algebraic Effect Handlers with Resources and Deep Finalization.

Microsoft Technical Report, MSR-TR-2018-10, v1.

DAAN LEIJEN, Microsoft Research

Algebraic effect handlers are a powerful abstraction mechanism that can express many complex control-flow mechanisms. In this article we first define a basic operational semantics and type system for algebraic effect handlers, and then build on that to formalize various optimizations and extensions. In particular, we show how to optimize tail-resumptive operations using skip frames, formalize a semantics and type rules for first-class resource handlers that can express polymorphic references, and formalize a design for finalizers and initializers to handle external resources with linearity constraints. We introduce the concept of *deep finalization* which ensures finalization is robust across operation handlers even if operations themselves do not resume.

updates:
v1, 2018-04-03: Initial version.

1. INTRODUCTION

Algebraic effects [36] and their extension with handlers [37, 38], are a novel way to describe many control-flow mechanisms in programming languages. In general any free monad can be expressed as an effect handler and they have been used to describe complex control structures such as iterators, async-await, concurrency, parsers, state, exceptions, etc. (without needing to extend the compiler or language) [10, 14, 16, 26, 48].

Recently, there are various implementations of algebraic effects, either embedded in other languages like Haskell [16, 48], Scala [5], or C [27], or built into a language, like Eff [2], Links [14], Frank [32], Koka [28], and Multi-core OCaml [10, 46]. Even though the theoretical foundations for algebraic effects and handlers are well-understood, for any practical implementation of effect handlers there are many open questions, like the potential efficiency of handlers, or their interaction with linear resources. In this paper we try to address some of these questions by giving an end-to-end description and formalization of effect handlers and various novel extensions.

In Section 4 we define the operational semantics of basic algebraic effects and handlers based on [28]. We build on these semantics as the foundation for various optimizations and extensions that are required when using effects in practice:

- Section 5 describes *skip frames* and formalizes an important optimization that avoids capturing and restoring the stack for tail-resumptive operations, and essentially turns them into dynamic method calls. Since almost all operations in practice are tail-resumptive, this is a very important optimization for any practical implementation. We prove that the extended formal semantics with the tail-resumptive optimization is sound with respect to the original semantics.
- Section 6 extends the semantics with effect *injection* allowing to skip innermost handlers. Biernacki et al. [3] show that this is an essential operation for writing composable higher-order functions that use effect handlers. Their formulation used a separate 0-*free* predicate, but we reformulate this using straightforward evaluation contexts, and we prove that these concepts are equivalent.
- Our main contribution is in Section 7 where we extend the semantics further with handlers that are can be addressed as first-class values, called *resources*. This adds a lot more expressiveness but surprisingly only needs minimal changes to the original semantics. As such, resources are a powerful abstraction but can still be understood and reasoned about as regular

effect handlers. We show through various examples how resources can be used to model files, polymorphic references, and event streams in CorrL [4].
- When using external resources, like files, we need a robust way to run finalization code. However, we cannot specialize this for just exception effects but need a general mechanism that works for any effect definition: any operation might not resume, or resume more than once. Both those cases cause problems when working with an external resource that has a linearity constraint. Section 8 extends the semantics further with general *finalizers* and *initializers* that work across any effect and can safely encapsulate linear resources. We also introduce the novel concept of *deep finalization* (Section 8.2) to write robust operation clauses.

Except for initializers[1], all of the above contributions have been fully implemented [25]. We first give a mini-overview of algebraic effects in Section 2, and formalize the syntax and type system in Section 3. After that we describe the basic semantics and discuss the various extensions. There is no separate related work section as we try to discuss related work in each section inline.

2. BASIC ALGEBRAIC EFFECTS

We are demonstrate all our examples in the context of the Koka programming language as this is one of the few implementations with a full type inference system that tracks effects of each function. The type of a function has the form $\tau \to\ \epsilon\ \tau'$ signifying a function that takes an argument of type τ, returns a result of type τ' and may have a side effect ϵ. We can leave out the effect and write $\tau \to\ \tau'$ as a shorthand for the total function without any side effect: $\tau \to\ \langle\rangle\ \tau'$. A key observation on Moggi's early work on monads [34] was that values and computations should be assigned a different type. Koka applies that principle where effect types only occur on function types; and any other type, like *int*, truly designates an evaluated value that cannot have any effect.

Koka has many features found in languages like ML and Haskell, such as type inference, algebraic data types and pattern matching, higher-order functions, impredicative polymorphism, open data types, etc. The pioneering feature of Koka is the use of row types with scoped labels to track effects in the type system. This gives Koka a very strong semantic foundation combining the purity of Haskell with the call-by-value semantics of ML.

There are various ways to understand algebraic effects and handlers. As described originally [36, 38], the signature of the effect operations forms a free algebra which gives rise to a free monad. Free monads provide a natural way to give semantics to effects, where handlers describe a *fold* over the algebra of operations [45]. Using a more operational perspective, we can also view algebraic effects as *resumable* exceptions. We therefore start our overview by modeling exceptional control flow.

2.1. Exceptions as Algebraic Effects

The exception effect `exc` can be defined in Koka as:

```
effect exc {
  raise( s : string ) : a
}
```

This defines a new effect type `exc` with a single primitive operation, `raise` with type `string → exc a` for any `a` (Koka uses single letters for polymorphic type variables). The raise operation can be used just like any other function:

[1]Initializers are already supported by the runtime but have no builtin syntax yet.

```
fun safediv( x, y ) {
  if (y==0) then raise("divide by zero") else x / y
}
```

Type inference will infer the type `(int,int) → exc int` propagating the exception effect. Up to this point we have introduced the new effect type and the operation interface, but we have not yet defined what these operations mean. The semantics of an operation is given through an algebraic effect *handler* which allows us to *discharge* the effect type. The standard way to discharge exceptions is by catching them, and we can write this using effect handlers as:

```
fun catch(action,h) {
  handle(action) {
    raise(s) → h(s)
  }
}
```

The `handle` construct for an effect takes an `action` to evaluate and a set of operation clauses. The inferred type of `catch` is:

```
catch : ( action: () → ⟨exc|e⟩ a, h : string → e a) → e a
```

The type is polymorphic in the result type `a` and its final effects `e`, where the `action` argument has an effect `⟨exc|e⟩`: this syntax denotes effect row extension, and states that `action` can have an `exc` effect and possibly more effects `e`. As we can see, the `handle` construct discharged the `exc` effect and the final result effect is just `e`. For example,

```
fun zerodiv(x,y) {
  catch( { safediv(x,y) }, fun(s){ 0 } )
}
```

has type `(int,int) → ⟨⟩ int` and is a total function. Note that the Koka *braced* syntax `{ safediv(x,y) }` denotes an anonymous function that takes no arguments, i.e. it is equivalent to `fun(){ safediv(x,y) }`.

Besides clauses for each operation, each handler can have a `return` clause too: this is applied to the final result of the handled action. In the previous example, we just passed the result unchanged, but in general we may want to apply some transformation. For example, transforming exceptional computations into `maybe` values:

```
fun to-maybe(action) {
  handle(action) {
    return x  → Just(x)
    raise(s)  → Nothing
}}
```

with the inferred type `(() → ⟨exc|e⟩ a) → e maybe⟨a⟩`. Note that Koka uses angled brackets for both effect row types, but also for type application as in the result type `maybe⟨a⟩`.

The `handle` construct is actually syntactic sugar over the more primitive `handler` construct:

```
handle(action) { ... }  ≡  (handler{ ... })(action)
```

A `handler` just takes a set of operation clauses for an effect, and returns a function that discharges the effect over a given action. This allows us to express `to-maybe` more concisely as a (function) value:

```
val to-maybe = handler {
  return x  → Just(x)
  raise(s)  → Nothing
}
```

with the same type as before.

Just like monadic programming, algebraic effects allows us to conveniently program with exceptions without having to explicitly plumb `maybe` values around. When using monads though we have to provide a `Monad` instance with a `bind` and `return`, and we need to create a separate discharge function. In contrast, with algebraic effects we only define the *operation* interface and the discharge is implicit in the handler definition.

2.2. State: Resuming Operations

The exception effect is somewhat special as it never resumes: any operations following the `raise` are never executed. Usually, operations will *resume* with a specific result instead of cutting the computation short. For example, we can have an `input` effect:

```
effect input {
  getstr() : string
}
```

where the operation `getstr` returns some input. We can use this as:

```
fun hello() {
  val name = getstr()
  println("Hello " + name)
}
```

An obvious implementation of `getstr` gets the input from the user, but we can just as well create a handler that takes a set of strings to provide as input, or always returns the same string:

```
val always-there = handler {
  return x   → x
  getstr()   → resume("there")
}
```

Every operation clause in a handler brings an identifier `resume` in scope which takes as an argument the result of the operation and resumes the program at the invocation of the operation – if the `resume` occurs at the tail position (as in our example) it is much like a regular function call. Executing `always-there(hello)` will output:

```
> always-there(hello)
Hello there
```

As another example, we can define a stateful effect:

```
effect state⟨s⟩ {
  get() : s
  put( x : s ) : ()
}
```

The `state` effect is polymorphic over the values `s` it stores. For example, in

```
fun counter() {
  val i = get()
  if (i ≤ 0) then () else {
    println("hi")
    put(i - 1);
    counter()
  }
}
```

Expressions	e	::=	$e(e)$	application
		|	val $x = e;\ e$	binding
		|	$\text{handle}^{\text{l}}_{h}(e)$	handler
		|	v	value
Values	v	::=	$x \mid C$	variables, constants
		|	$\lambda x.\ e$	lambda expressions
		|	op^{l}	operation of effect l
Clauses	h	::=	return $x \to e$	return clause
		|	$op(x) \to e;\ h$	operation clause
Static effect label	l	::=	l	(for now equal to effect constants)

Fig. 1. Syntax of expressions

the type becomes `() → ⟨state⟨int⟩,console,div|e⟩ ()` with the state instantiated to `int`. To define the `state` effect we could use the built-in state effect of Koka, but a cleaner way is to use *parameterized* handlers. Such handlers take a parameter that is updated at every `resume`. Here is a possible definition for handling state:

```
val state = handler(s) {
  return x  → (x,s)
  get()     → resume(s,s)
  put(s')   → resume((),s')
}
```

We see that the handler binds a parameter s (of the polymorphic type `s`), the current state. The `return` clause returns the final result tupled with the final state. The `resume` function in a *parameterized handler* takes now multiple arguments: the first argument is the result of the operation, while the last argument is the new handler parameter used *when handling the resumption*. The `get` operation leaves the current state unchanged, while the `put` operation resumes with the passed-in state argument as the new handler parameter. The function returned by the handler construct now takes the initial state as an extra argument:

```
state : (init: s, action: () → ⟨state⟨s⟩|e⟩ a) → e (a,s)
```

and we can use it as:

```
> state(2,counter)
hi
hi
```

This concludes the overview but we covered many essential aspects algebraic effects. We refer to other work for more in-depth explanations and examples [2, 14, 16, 29, 48].

3. SYNTAX AND TYPES

3.1. Type Rules

In this section we give a formal definition of our polymorphic row-based effect system for the core calculus of Koka. The material in this section (and large parts of the next section on semantics) has

Types	τ^k	::=	α^k	type variable
		|	$c^k\langle \tau_1^{k_1}, \ldots, \tau_n^{k_n}\rangle$	kind of c is $(k_1, \ldots, k_n) \to k$
Kinds	k	::=	$* \mid \mathsf{e}$	values, effects
		|	k	effect constants
		|	$(k_1, \ldots, k_n) \to k$	type constructor
Type scheme	σ	::=	$\forall \alpha^k.\, \sigma \mid \tau^*$	

Type constants	(), *bool*	::	$*$	unit, booleans
	$(\_ \to \_\ \_)$	::	$(*, \mathsf{e}, *) \to *$	functions
	$\langle\rangle$	::	e	empty effect
	$\langle \_ \mid \_ \rangle$	::	$(\mathsf{k}, \mathsf{e}) \to \mathsf{e}$	effect extension
	exn, …	::	k	various effect constants

Value types	τ	$\doteq$	τ^*	kinds are $*$ by default
Total functions	$\tau_1 \to \tau_2$	$\doteq$	$\tau_1 \to \langle\rangle\ \tau_2$	
Effect labels	l	$\doteq$	c^{k}	
Effect type	ι	$\doteq$	$l\langle \tau_1, \ldots, \tau_n\rangle$	
Effect rows	ϵ	$\doteq$	τ^{e}	
Effect row variables	μ	$\doteq$	α^{e}	
Closed effect rows	$\langle \iota_1, \ldots, \iota_n\rangle$	$\doteq$	$\langle \iota_1, \ldots, \iota_n \mid \langle\rangle\ \rangle$	
Effect row extension	$\langle \iota_1, \ldots, \iota_n \mid \epsilon\rangle$	$\doteq$	$\langle \iota_1 \mid \ldots \langle \iota_n \mid \epsilon\rangle \ldots \rangle$	

Fig. 2. Syntax of types and kinds

$$\epsilon \cong \epsilon \quad [\text{EQ-REFL}] \qquad \frac{\epsilon_1 \cong \epsilon_2 \quad \epsilon_2 \cong \epsilon_3}{\epsilon_1 \cong \epsilon_3}\ [\text{EQ-TRANS}]$$

$$\frac{\epsilon_1 \cong \epsilon_2}{\langle \iota \mid \epsilon_1\rangle \cong \langle \iota \mid \epsilon_2\rangle}\ [\text{EQ-HEAD}] \qquad \frac{\iota_1 \not\equiv \iota_2}{\langle \iota_1 \mid \langle \iota_2 \mid \epsilon\rangle\ \rangle \cong \langle \iota_2 \mid \langle \iota_1 \mid \epsilon\rangle\ \rangle}\ [\text{EQ-COMMUTE}]$$

$$\frac{l \neq l'}{l\langle \tau_1, \ldots, \tau_n\rangle \not\equiv l'\langle \tau_1', \ldots, \tau_n'\rangle}\ [\text{UNEQ-LABEL}]$$

Fig. 3. Row equivalence

been presented before in a similar form [28] but we include it again here to make this article self sufficient. The well informed reader may skip through these sections.

Figure 2 defines the syntax of types and expressions. The expression grammar is straightforward but we are careful to distinguish values v from expressions e that can have effects. Values consist of variables x, constants C, operations *op*, and lambda's. Expressions include handler expressions $\mathsf{handle}_h^{\mathsf{l}}(e)$ where h is a set of operation clauses and l identifies the effect that is handled in the expression ϵ. The handler construct of the previous section can be seen as syntactic sugar, where:

$\mathsf{handler}_h^{\mathsf{l}} \equiv \lambda f.\ \mathsf{handle}_h^{\mathsf{l}}(f())$

$$\frac{\Gamma(x) = \sigma}{\Gamma \vdash x : \sigma \mid \epsilon}\ [\textsc{Var}] \qquad \frac{\Gamma \vdash e_1 : \sigma \mid \epsilon \quad \Gamma, x : \sigma \vdash e_2 : \tau \mid \epsilon}{\Gamma \vdash \mathsf{val}\ x = e_1;\ e_2 : \tau \mid \epsilon}\ [\textsc{Let}]$$

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2 \mid \epsilon'}{\Gamma \vdash \lambda x.\ e : \tau_1 \to \epsilon'\ \tau_2 \mid \epsilon}\ [\textsc{Lam}] \qquad \frac{\Gamma \vdash e_1 : \tau_2 \to \epsilon\ \tau \mid \epsilon \quad \Gamma \vdash e_2 : \tau_2 \mid \epsilon}{\Gamma \vdash e_1(e_2) : \tau \mid \epsilon}\ [\textsc{App}]$$

$$\frac{\Gamma \vdash e : \tau \mid \langle\rangle \quad \overline{\alpha} \notin \mathsf{ftv}(\Gamma)}{\Gamma \vdash e : \forall \overline{\alpha}.\ \tau \mid \epsilon}\ [\textsc{Gen}] \qquad \frac{\Gamma \vdash e : \forall \overline{\alpha}.\ \tau \mid \epsilon}{\Gamma \vdash e : \tau[\overline{\alpha} \mapsto \overline{\tau}] \mid \epsilon}\ [\textsc{Inst}]$$

$$\frac{\begin{array}{ll}\Gamma \vdash e : \tau \mid \langle \iota \mid \epsilon \rangle & S(l) = \{op_1, \ldots, op_n\} \\ \Gamma, x : \tau \vdash e_r : \tau_r \mid \epsilon & \Gamma \vdash op_i : \tau_i \to \langle \iota \rangle\ \tau_i' \mid \langle\rangle \\ \Gamma, \mathit{resume} : \tau_i' \to \epsilon\ \tau_r,\ x_i : \tau_i \vdash e_i : \tau_r \mid \epsilon & \end{array}}{\Gamma \vdash \mathsf{handle}^l\{\ op_1(x_1) \to e_1;\ \ldots;\ op_n(x_n) \to e_n;\ \mathsf{return}\ x \to e_r\ \}(e) : \tau_r \mid \epsilon}\ [\textsc{Handle}]$$

Fig. 4. Type rules.

For simplicity we assume that all operations take just one argument. We also use membership notation $op(x) \to e \in h$ to denote that h contains a particular operation clause. Sometimes we shorten this to $op \in h$.

Well-formed types are guaranteed through kinds k which we denote using a superscript, as in τ^k. We have the usual kinds for value types $*$ and type constructors $\to$, but because we use a row based effect system, we also have kinds for effect rows ϵ, and effect constants l. When the kind of a type is immediately apparent or not relevant, we usually leave it out. For clarity, we use α for regular type variables, and μ for effect type variables.

Effect types are defined as a row of effect label types ι. Such row is either empty $\langle\rangle$, a polymorphic effect variable μ, or an extension of an effect ϵ with a label ι, written as $\langle \iota \mid \epsilon \rangle$. Effect labels must start with an effect constant l and are never polymorphic. By construction, effect types are either a *closed effect* of the form $\langle \iota_1, \ldots, \iota_n \rangle$, or an *open effect* of the form $\langle \iota_1, \ldots, \iota_n \mid \mu \rangle$.

We cannot use direct equality on types since we would like to regard effect rows equivalent up to the order of their effect constants. Figure 3 defines an equivalence relation ($\cong$) between effect rows. This relation is essentially the same as for the *scoped labels* record system [23] with the difference that we *ignore the type arguments when comparing labels*. This is apparent in the Uneq-Label rule. This is essential to guarantee a deterministic and terminating unification algorithm which is essential for type inference.

In contrast to other record calculi [13, 31, 39, 44], our approach does not require extra constraints, like *lacks* or *absence* constraints, on the types which simplifies the type system significantly. The system also allows *duplicate* labels, where an effect $\langle exc, exc \rangle$ is legal and different from $\langle exc \rangle$. This proves to be an essential feature to give a proper type to effect injection (Section 6), and also to emulate generative types where we need types of the form $\langle heap\langle s_1, \alpha \rangle, heap\langle s_2, \beta \rangle | \epsilon \rangle$ (Section 6.2).

3.2. Type Inference

The type rules for our calculus are given in Figure 4. A type environment Γ maps variables to types and can be extended using a comma: if Γ' equals $\Gamma, x : \sigma$, then $\Gamma'(x) = \sigma$ and $\Gamma'(y) = \Gamma(y)$ for any $x \neq y$. A type rule $\Gamma \vdash e : \tau \mid \epsilon$ states that under environment Γ, the expression e has type τ with possible effects ϵ.

The type rules are quite standard. The rule var derives the type of a variable x with an arbitrary effect ϵ. We may have expected to derive only the total effect $\langle\rangle$ since the evaluation of a variable has

no effect at all. However, there is no rule that lets one upgrade the final effect and instead we need to pick the final effect right away. Another way to look at this is that since the variable evaluation has no effect, we are free to assume any arbitrary effect. We use the VAR rule too for operations *op* and constants *c* where we assume the types of those are part of the initial environment.

The LAM rule is similar in that it assumes any effect ϵ for the result since the evaluation of a lambda is a value. At this rule, we also see how the effect derived for the body of a lambda ϵ' shifts to the derived function type $\tau_1 \ \to \ \epsilon' \ \tau_2$. Rule APP is standard and derives an effect ϵ requiring that its premises derive the same effect as the function effect.

Rules INST and GEN instantiate and generalize types. The generalization rule has an interesting twist as it requires the derived effect to be total. When combining our calculus with polymorphic mutable reference cells, this is required to ensure a sound semantics. This is the semantic equivalent to the syntactic value restriction in ML. In our core calculus we cannot define polymorphic reference cells directly so the restriction is not necessary persé but it seems good taste to leave it in as it is required for the full Koka language.

Finally, the HANDLE rule types effect handlers. We assume that effect declarations populate an initial environment Γ_0 with the types of declared operations, and also a *signature environment S* that maps declared effect labels to the set of operations that belong to it. We also assume that all operations have unique names, such that given the operation names, we can uniquely determine to which effect *l* they belong.

The rule HANDLE requires that all operations in the signature $S(l)$ are part of the handler, and we reject handlers that do not handle all operations that are part of the effect *l*. The return clause is typed with $x \ : \ \tau$ where τ is the result type of the handled action. All clauses must have the same result type τ_r and effect ϵ. For each operation clause $op_i(x_i) \ \to \ e_i$ we first look up the type of op_i in the environment as $\tau_i \ \to \ \langle l \rangle \ \tau_i'$, and bind x_i to the argument type of the operations, and bind *resume* to the function $\tau_i' \ \to \ \tau_r$ where the argument type of resume is the result type of the operation. The derived type of the handler is a function that discharges the effect type *l*.

As described in earlier work [24], type inference is straightforward based on Hindley-Milner [15, 33], and is sound and complete with respect to the declarative type rules.

3.3. Simplifying Types

The rule APP is a little surprising since it requires both the effects of the function and the arguments to match. This only works because we set things up to always be able to infer the effects of functions that are 'open' – i.e. have a polymorphic μ in their tail. For example, consider the identity function:

$$id \ = \ \lambda x.\ x$$

If we assign the valid type $\forall \alpha.\ \alpha \ \to \ \langle \rangle \ \alpha$ to the *id* function, we get into trouble quickly. For example, the application $id(raise(\text{"}hi\text{"}))$ would not type check since the effect of *id* is total while the effect of the argument contains *exc*. Of course, the type inference algorithm always infers a most general type for *id*, namely $\forall \alpha \ \mu.\ \alpha \ \to \ \mu \ \alpha$ which has no such problems.

In practice though we wish to simplify the types more and leave out 'obvious' polymorphism. In Koka we adopted two extra type rules to achieve this. The first rule opens closed effects of function types:

$$\frac{\Gamma \ \vdash \ e \ : \ \tau_1 \to \langle l_1, \ldots, l_n \rangle \ \tau_2 \mid \epsilon}{\Gamma \ \vdash \ e \ : \ \tau_1 \to \langle l_1, \ldots, l_n \mid \epsilon' \rangle \ \tau_2 \mid \epsilon} \ [\text{OPEN}]$$

Evaluation Contexts:

$$
\begin{array}{lll}
\mathsf{E} & ::= \Box \mid \mathsf{E}(e) \mid v(\mathsf{E}) \mid \mathsf{val}\ x\ =\ \mathsf{E};\ e \mid \mathsf{handle}^{l}_{h}(\mathsf{E}) & \\
\mathsf{H}^{l} & ::= \Box \mid \mathsf{H}^{l}(e) \mid v(\mathsf{H}^{l}) \mid \mathsf{val}\ x\ =\ \mathsf{H}^{l};\ e & \\
 & \quad \mid\ \mathsf{handle}^{l'}_{h}(\mathsf{H}^{l}) & \text{if } l \neq l'
\end{array}
$$

Reduction Rules:

$$
\begin{array}{llll}
(\delta) & C(v) & \longrightarrow\ \delta(C, v) & \text{if } \delta(c, v) \text{ is defined} \\
(\beta) & (\lambda x.\ e)(v) & \longrightarrow\ e[x \mapsto v] & \\
(let) & \mathsf{val}\ x\ =\ v;\ e & \longrightarrow\ e[x \mapsto v] & \\
 & & & \\
(return) & \mathsf{handle}^{l}_{h}(v) & \longrightarrow\ e[x \mapsto v] & \\
 & & \text{with} & \\
 & & \quad (\mathsf{return}\ x \rightarrow e)\ \in h & \\
 & & & \\
(handle) & \mathsf{handle}^{l}_{h}(\mathsf{H}^{l}[op^{l}(v)]) & \longrightarrow\ e[x \mapsto v,\ resume \mapsto r)] & \\
 & & \text{with} & \\
 & & \quad (op(x) \rightarrow e)\ \in h & \\
 & & \quad r\ =\ \lambda y.\ \mathsf{handle}^{l}_{h}(\mathsf{H}^{l}[y]) &
\end{array}
$$

Fig. 5. Reduction rules and evaluation contexts

With this rule, we can type the application *id*(*raise*("*hi*")) even with the simpler type assigned to *id* as we can open the effect type of *id* using the OPEN rule to match the effect of *raise*("*hi*"). We combine this with a closing rule (which is just an instance of INST/GEN):

$$
\frac{\begin{array}{l}\Gamma \vdash e : \forall \mu \overline{\alpha}.\ \tau_1 \rightarrow \langle l_1, \ldots, l_n \mid \mu \rangle\ \tau_2 \mid \epsilon \\ \mu \notin \mathsf{ftv}(\tau_1, \tau_2, l_1, \ldots, l_n)\end{array}}{\Gamma \vdash e : \forall \overline{\alpha}.\ \tau_1 \rightarrow \langle l_1, \ldots, l_n \rangle\ \tau_2 \mid \epsilon}\ [\textsc{close}]
$$

During inference, the rule CLOSE is applied (when possible) before assigning a type to a let-bound variable. In general, such technique would lead to incompleteness where some programs that were well-typed before, may now be rejected since CLOSE assigns a less general type. However, due to OPEN this is not the case: at every occurrence of such let-bound variable the rule OPEN can always be applied (possibly surrounded by INST/GEN) to lead to the original most general type – i.e. even though the types are simplified, the set of typeable programs is unchanged.

4. SEMANTICS

In this section we define a precise operational semantics. Even though algebraic effects are originally conceived with a semantics in category theory, we will give a more regular operational semantics that corresponds closely to various implementations in practice.

The operational semantics is given in Figure 5 and consists of just five evaluation rules. We use two evaluation contexts: the E context is the usual one for a call-by-value lambda calculus. The H^{l} context is used for handlers. In particular, it evaluates down through any handlers that do *not*

handle the effect l. This is used to express concisely that the *innermost handler* handles a particular operation.

The first three reduction rules, (δ), (β), and (*let*) are the standard rules of call-by-value evaluation. The final two rules evaluate handlers. Rule (*return*) applies the return clause of a handler when the argument is fully evaluated. Note that this evaluation rule subsumes both lambda- and let-bindings and we can define both as a reduction to a handler without any operations:

$$(\lambda x.\ e_1)(e_2)\ \equiv\ \mathsf{handle}^{\langle\rangle}\{\mathsf{return}\ x \to e_1\}(e_2)$$

and

$$\mathsf{val}\ x\ =\ e_1;\ e_2\ \equiv\ \mathsf{handle}^{\langle\rangle}\{\mathsf{return}\ x \to e_2\}(e_1)$$

This equivalence is used in the Frank language [32] to define everything, including functions and applications, as handlers.

The next rule, (*handle*), is where all the action is. Here we see how algebraic effect handlers are closely related to delimited continuations as the evaluation rules captures a delimited 'stack' $\mathsf{H}^l[op^l(v)]$ under the handler h. Using a H^l context ensures by construction that only the innermost handler for an effect l will handle the operation $op^l(v)$. Evaluation continues with the expression ϵ but besides binding the parameter x to v, also the *resume* variable is bound to the continuation: $\lambda y.\ \mathsf{handle}^l_h(\mathsf{H}^l[y])$. Applying *resume* results in continuing evaluation at H^l with the supplied argument as the result. Moreover, the continued evaluation occurs again under the handler h.

Resuming under the same handler is important as it ensures that our semantics correspond to the original categorical interpretation of algebraic effect handlers as a *fold* over the effect algebra [38]. If the continuation is not resumed under the same handler, it behaves more like a *case* statement doing only one level of the *fold*. Such handlers are sometimes called *shallow handlers* [16, 32]. In that case *resume* would only restore the stack without the handler: $resume\ \mapsto\ \lambda y.\ \mathsf{H}^l[y]$.

For simplicity, we ignore parameterized handlers for now but we come back to it in Section 8 when formalizing initializers. The reduction rule is straightforward though – a handler with a single parameter p is reduced as:

$$\begin{array}{l}\mathsf{handle}^l_h(p\ =\ v_p)(\mathsf{H}^l[op^l(v)]) \\ \quad\longrightarrow \{\ op(v) \to e\ \in h\ \} \\ e[x \mapsto v,\ p \mapsto v_p,\ resume \mapsto \lambda y\ q.\ \mathsf{handle}^l_h(p\ =\ q)(\mathsf{H}^l[y])]\end{array}$$

where it is apparent how *resume* now resumes under a handler with an updated parameter p.

Using the reduction rules of Figure 5 we can define the overall evaluation function ($\longmapsto$), where $\mathsf{E}[e]\ \longmapsto\ \mathsf{E}[e']$ iff $e\ \longrightarrow\ e'$. We also define the function $\longmapsto^*$ as the reflexive and transitive closure of $\longmapsto$.

4.1. Dot Notation

We can view the evaluation contexts E and H^l, as evaluation stacks, where handle^l_h are special frames on the stack. To make this notion of an evaluation context as a stack more apparent, we often use *dot notation* where we use the following shorthands:

$$\begin{array}{lcl}\mathsf{H}^l \cdot e & \doteq & \mathsf{H}^l[e] \\ \mathsf{E} \cdot e & \doteq & \mathsf{E}[e] \\ e_1 \cdot e_2 & \doteq & e_1(e_2)\end{array}$$

This makes most proofs and reduction rule easier to read and follow.

4.2. Return as an Operation

Instead of having a separate rule for "return x" clauses, we can treat them as syntactic sugar for an implicit $\mathit{return}(x)$ operation, where we consider:

$$\mathsf{handle}^{l}_{\{h;\ \mathsf{return}\ x \to e_r\}}(e)$$

as a shorthand for:

$$\mathsf{handle}^{l}_{\{h;\ \mathit{return}(x) \to e_r\}}(\mathit{return}(e))$$

where $\mathit{resume} \notin \mathsf{fv}(e_r)$ (and $\mathit{return} \notin \mathsf{op}(h)$). By treating return clauses as operations, we capture all substitution under a single rule, and don't require extra rules for parameterized handlers for example (where a return clause can access the local parameter as well).

We can imagine adding the simpler (*value*) rule to the reductions:

$$(\mathit{value}) \quad \mathsf{handle}^{l}_{h}(v) \quad \longrightarrow \quad v$$

but that it is not strictly necessary. Since we assume that every handler has a "return x" clause (which is by default "return $x \ \to \ x$"), all handlers always eventually end with the $\mathit{return}(v)$ operation and the plain value case never occurs.

We can show that the new syntactic transformation leads to equivalent reductions as with the original rule.

Proof. Starting from the premise of the original (*return*) rule, we have:

$$\begin{array}{l}
\mathsf{handle}^{l}_{\{h;\ \mathsf{return}\ x \to e_r\}}(v) \\
\quad \rightsquigarrow \ \{\ \text{syntax}\ \} \\
\mathsf{handle}^{l}_{\{h;\ \mathit{return}(x) \to e_r\}}(\mathit{return}(v)) \\
\quad \longrightarrow \\
e_r[x \mapsto v,\ \mathit{resume} \mapsto r] \\
\quad = \ \{\ \mathit{resume} \notin \mathsf{fv}(e_r)\ \} \\
e_r[x \mapsto v]
\end{array}$$

which is equivalent to the original rule. □

4.3. Comparison with Delimited Continuations

Shan [41] has shown that various variants of delimited continuations can be defined in terms of each other. Following Kammar et al. [16], we can define a variant of Danvy and Filinski's [8] shift and reset operators, called shift_0 and reset_0, as

$$\mathit{reset}_0(\mathsf{X}[\mathit{shift}_0(\lambda k.\ e)]) \longrightarrow e[k \mapsto \lambda x.\ \mathit{reset}_0(\mathsf{X}[x])]$$

where we write X for a context that does not contain a reset_0. Therefore, the shift_0 captures the continuation up to the nearest enclosing reset_0. Just like handlers, the captured continuation is itself also wrapped in a reset_0. Unlike handlers though, the handling is done by the shift_0 directly instead of being done by the delimiter reset_0. From the reduction rule, we can easily see that we can implement delimited continuations using algebraic effect handlers, where shift_0 is an operation in the *delim* effect and $\mathsf{X} \equiv \mathsf{H}^{\mathit{delim}}$:

$$\mathit{reset}_0(e) \ \doteq \ \mathsf{handle}^{\mathit{delim}}\{\ \mathit{shift}_0(f) \to f(\mathit{resume})\ \}(e)$$

Using this definition, we can show it is equivalent to the original reduction rule for delimited continuations, where we write h for the handler $shift_0(f) \;\to\; f(resume)$:

$$
\begin{aligned}
&reset_0(\mathsf{X}[shift_0(\lambda k.\ e)]) \\
&\quad \doteq \\
&\mathsf{handle}_h^{delim} \cdot \mathsf{H}^{delim} \cdot shift_0^{delim}(\lambda k.\ e) \\
&\quad \longrightarrow \\
&(f(resume))[f \mapsto \lambda k.\ e,\ resume \mapsto \lambda x.\ \mathsf{handle}_h^{delim} \cdot \mathsf{H}^{delim} \cdot x] \\
&\quad \longrightarrow \\
&(\lambda k.\ e)(\lambda x.\ \mathsf{handle}_h^{delim} \cdot \mathsf{H}^{delim} \cdot x) \\
&\quad \longrightarrow \\
&e[k \mapsto \lambda x.\ \mathsf{handle}_h^{delim} \cdot \mathsf{H}^{delim} \cdot x] \\
&\quad \doteq \\
&e[k \mapsto \lambda x.\ reset_0(\mathsf{X}[x])]
\end{aligned}
$$

Even though we can define this equivalence in our untyped calculus, we cannot give a general type to the $shift_0$ operation in our system. To generally type shift and reset operations a more expressive type system with answer types is required [1, 7]. Kammar et al. [16] also show that it is possible to go the other direction and implement handlers using delimited continuations.

4.4. Properties

It is shown in [28] using techniques from [47] that well-typed programs cannot go *wrong*:

Theorem 1. (*Semantic soundness*)
If $\cdot \vdash \ e \ : \ \tau \ \mid \ \epsilon$ then either $e \Uparrow$ or $e \mapsto\rangle \ v$ where $\cdot \vdash \ v \ : \ \tau \ \mid \ \epsilon$.

where we use the notation $e \Uparrow$ for a never-ending reduction. Another nice property that holds is:

Lemma 1. (*Effects are meaningful*)
If $\Gamma \ \vdash \ \mathsf{H}^l[op^l(v)] \ : \ \tau \mid \ \epsilon$, then $l \ \in \epsilon$.

This is a powerful lemma as it states that effect types cannot be discarded (except through handlers). This lemma also implies effect types are *meaningful*, e.g. if a function does not have an *exc* effect, it will never throw an exception.

5. OPTIMIZING TAIL RESUMPTIONS

From the reduction rules, we can already see some possible optimizations that can be used to compiler handlers efficiently. For example, if a handler never resumes, we can treat it similarly to how exceptions are handled and do not need to capture the execution context (there are some caveats here and we come back to this is Section 8 on finalization).

An important other optimization applies to *tail resumptions*, i.e. a *resume* that occurs in the tail position of an operation clause. In that case one implementation strategy could be to look up the innermost handler but not unwind the stack, and instead directly execute the operation branch in place. As its last action will be to resume, we now do not need to restore the stack but instead directly return the result in place.

Since almost all operations are tail-resumptive in practice, this is a huge opportunity for an efficient implementation, effectively turning most operations into dynamically scoped method calls. The main cost is then looking up the operations in the dynamic handler stack. Currently, one of the most efficient implementations in C can do 150 million of such operations per second on a Core i7 @ 2.6GHz [27]. One can imagine removing even the search cost by having a static ‘virtual method

Extended Syntax:

$$\begin{array}{rrcll} \text{Expressions} & e & ::= & \ldots & \\ & & \mid & \mathsf{skip}^{l}(e) & \text{skip frame} \end{array}$$

Extended Evaluation Contexts:

$$\begin{array}{rcl} \mathsf{E} & ::= & \ldots \\ & \mid & \mathsf{skip}^{l}(e) \\ \mathsf{H}^{l} & ::= & \ldots \\ & \mid & \mathsf{handle}_{h}^{l'}(\mathsf{H}^{l'}[\mathsf{skip}^{l'}(\mathsf{H}^{l})]) \end{array}$$

Extended Reduction Rules:

$$\begin{array}{lll} (\mathit{handle}_t) & \mathsf{handle}_{h}^{l} \cdot \mathsf{H}^{l} \cdot \mathit{op}^{l}(v) \longrightarrow \mathsf{handle}_{h}^{l} \cdot \mathsf{H}^{l} \cdot \mathsf{skip}^{l} \cdot \mathit{resume}(e)[x \mapsto v] & \\ & \qquad \text{with} \quad (\mathit{op}(x) \to \mathit{resume}(e)) \in h & \\ & \qquad\qquad \mathit{resume} \notin \mathsf{fv}(e) & \\ (\mathit{resume}_t) & \mathsf{handle}_{h}^{l} \cdot \mathsf{H}^{l} \cdot \mathsf{skip}^{l} \cdot \mathit{resume}(v) \longrightarrow \mathsf{handle}_{h}^{l} \cdot \mathsf{H}^{l} \cdot v & \end{array}$$

Fig. 6. Skip frames.

table' per effect that is updated by its handlers at runtime such that tail-resumptive operations truly become comparable to virtual method calls.

5.1. Skip Frames

However, implementing tail-resumptive operations in this way requires special skip frames to ensure that any operations called *inside* an operation clause are handled by the correct handler. In particular, since we leave the stack in place, we need to remember to ignore the part of the handler stack up to, and including, the handler that handles the operation. This is formalized in Figure 6 where we use the *dot notation* introduced in Section 4.1.

We extend the syntax with $\mathsf{skip}^{l}(e)$ frames that inactivates any handler up to, and including, the innermost handle_{h}^{l} frame. This is captured in the extra clause for the H^{l} contexts with the grammar rule $\mathsf{handle}_{h}^{l'} \cdot \mathsf{H}^{l'} \cdot \mathsf{skip}^{l'} \cdot \mathsf{H}^{l}$. Since there is no other way to handle through a skip frame, this ensures that no operations can be handled by any frames in the skip range.

There are two new reduction rules that capture the optimization. The (handle_t) rule can now apply to tail-resumptive operations where the branch is of the form $\mathit{op}(x) \to \mathit{resume}(e)$ where $\mathit{resume} \notin \mathsf{fv}(e)$. When this is the case, the reduction rule leaves the stack *as is*, and only pushes a skip^{l} frame and directly evaluates the branch in place. Once the branch is evaluated, the (resume_t) rule takes care of popping the skip^{l} frame again when it encounters the (unbound) resume call.

5.2. Soundness of Skip Frames

We would like to show that the optimization is *sound*: i.e. if we evaluate with the optimized (handle_t) rule we get the same results as if we would have used the plain (handle) rule.

To show this formally we use a subscript $_t$ to distinguish our new optimized reduction rules $\longrightarrow_t$ form the original reduction rules $\longrightarrow$, and similarly for the expression and evaluation contexts. We also define an *ignore* function denoted by a small overline, on expressions, $\overline{e}$, and contexts $\overline{\mathsf{E}}_t$ and $\overline{\mathsf{H}}_t^l$. This function removes any $\mathsf{handle}_h^l \cdot \mathsf{H}_t^l \cdot \mathsf{skip}^l$ sub expressions, effectively turning any of our extended expressions into an original one, and taking E_t to E, and H_t^l to H^l. Using this function, we can define soundness as:

Theorem 2. (*Soundness of Skip Frames*)
If $\mathsf{E}_t \cdot e_t \longrightarrow_t \mathsf{E}_t \cdot e_t'$ then $\overline{\mathsf{E}}_t \cdot \overline{e} \longrightarrow \overline{\mathsf{E}}_t \cdot \overline{e'}$.

Useful properties of the *ignore* function are $\overline{v} = v$, and $\overline{\mathsf{E}}_t \cdot \overline{\mathsf{handle}_h^l \cdot \mathsf{H}_t^l \cdot \mathsf{skip}^l}$ equals $\overline{\mathsf{E}}_t$.

Proof. (*Of Theorem 2*) We show this by case analysis on reduction rules. The first five rules are equivalent to the original rules. For ($handle_t$) we have:

$$
\begin{aligned}
&\overline{\mathsf{E}}_t \cdot \overline{\mathsf{handle}_h^l \cdot \mathsf{H}_t^l \cdot op(v)} \\
&\quad = \\
&\overline{\mathsf{E}_t \cdot \mathsf{handle}_h^l \cdot \mathsf{H}_t^l} \cdot op(v) \\
&\quad = \\
&\overline{\mathsf{E}}_t \cdot \mathsf{handle}_{\overline{h}}^{\overline{l}} \cdot \overline{\mathsf{H}}_t^l \cdot op(v) \\
&\quad \longrightarrow \\
&\overline{\mathsf{E}}_t \cdot \overline{e[x \mapsto v,\ resume \mapsto \lambda y.\ \mathsf{handle}_{\overline{h}}^{\overline{l}} \cdot \overline{\mathsf{H}}_t^l \cdot op(y)]} \\
&\quad = \{\ resume \notin \mathsf{fv}(e)\ \} \\
&\overline{\mathsf{E}}_t \cdot \overline{e[x \mapsto v]} \\
&\quad = \{\ op \in h\ \} \\
&\overline{\mathsf{E}}_t \cdot \overline{\mathsf{handle}_h^l \cdot \mathsf{H}_t^l \cdot \mathsf{skip}^l} \cdot \overline{e[x \mapsto v]} \\
&\quad = \\
&\overline{\mathsf{E}}_t \cdot \overline{\mathsf{handle}_h^l \cdot \mathsf{H}_t^l \cdot \mathsf{skip}^l \cdot e[x \mapsto v]}
\end{aligned}
$$

In the ($resume_t$) rule, we know by construction that in the original reduction rules, *resume* would have been bound as a regular $resume \mapsto \lambda y.\overline{\mathsf{handle}_h^l \cdot \mathsf{H}_t^l \cdot y}$:

$$
\begin{aligned}
&\overline{\mathsf{E}}_t \cdot \overline{\mathsf{handle}_h^l \cdot \mathsf{H}_t^l \cdot \mathsf{skip}^l \cdot resume(v)} \\
&\quad = \\
&\overline{\mathsf{E}}_t \cdot \overline{\mathsf{handle}_h^l \cdot \mathsf{H}_t^l \cdot \mathsf{skip}^l} \cdot resume(v) \\
&\quad = \\
&\overline{\mathsf{E}}_t \cdot resume(v) \\
&\quad = \\
&\overline{\mathsf{E}}_t \cdot (\lambda y.\overline{\mathsf{handle}_h^l \cdot \mathsf{H}_t^l \cdot y})(v) \\
&\quad \longrightarrow \\
&\overline{\mathsf{E}}_t \cdot \overline{\mathsf{handle}_h^l \cdot \mathsf{H}_t^l \cdot v}
\end{aligned}
$$

□

Extended Syntax:

$$\begin{array}{rrcll} \text{Expressions} & e & ::= & \ldots & \\ & & | & \mathsf{inject}^{\mathsf{l}}(e) & \text{effect injection} \end{array}$$

Extended Type Rules:

$$\frac{S(l) = \{op, \ldots\} \quad \Gamma \vdash op : \tau_1 \to \langle \iota \rangle\, \tau_2 \quad \Gamma \vdash e : \tau \,|\, \epsilon}{\Gamma \vdash \mathsf{inject}^{l}(e) : \tau \,|\, \langle \iota | \epsilon \rangle} \,[\textsc{inject}]$$

Extended Evaluation Contexts:

$$\mathsf{E} ::= \ldots \mid \mathsf{inject}^{\mathsf{l}}(\mathsf{E})$$

$$\begin{array}{rcll} \mathsf{H}^{l} & ::= & \ldots & \\ & | & \mathsf{inject}^{l'}(\mathsf{H}^{l}) & \text{if } l \neq l' \\ & | & \mathsf{handle}^{l}_{h}(\mathsf{H}^{l}[\mathsf{inject}^{l}(\mathsf{H}^{l})]) & \end{array}$$

Extended Reduction Rules:

$$(\mathit{inject})\ \mathsf{inject}^{l}(v) \to v$$

Fig. 7. Extension with injection

6. INJECTION

As observed by Biernacki et al. [3], to make algebraic effect handlers truly composable we need an inject function to *inject* effect types into the current effect row. For example, function `f` takes an `action` as an argument:

```
fun f( action : () → ⟨exn|e⟩ a) : e maybe⟨a⟩ {  // inferrred
  try {
    setup()
    Just(action())
  }
  fun(exn) {
    Nothing
  }
}
```

Suppose `f` only wants to handle exceptions raised by its own code (e.g. `setup`) but not any exceptions raised by `action`. As it is defined, `f` now handles any exception raised by `action` too. The solution is to use `inject` the exception effect into the effect of `action`:

```
fun f( action : () → e a ) : e maybe⟨a⟩ { // inferred
  try {
    setup()
```

```
      Just( inject⟨exn⟩{action()} )
    }
    fun(exn) {
      Nothing
    }
  }
```

Any exceptions raised by `action` will now propagate outside of `f` – i.e. inject has a runtime effect too as it needs to skip the exception handler in `f` when an exception is raised in `action`.

Figure 7 shows the extended syntax, evaluation contexts, and runtime reduction rules for inject. An inject expression $\mathsf{inject}^{\mathsf{l}}(e)$ injects the effect l into the effect of the expression e as shown in the type rules. Here we introduced an extra relation $\Gamma \vdash l : \iota$ that derives a full type from just an effect constant l as effects can generally have other parameters.

The reduction rule for injection does nothing: $\mathsf{inject}^{l}(v) \longrightarrow v$; Indeed, the essence of inject is in the extended handler context H. There are two new rules. The first one $\mathsf{inject}^{l'}(\mathsf{H}^{l})$ with $l \neq l'$ states that inject frames can be ignored if the effect labels differ. The second rule, $\mathsf{handle}^{l} \cdot \mathsf{H}^{l} \cdot \mathsf{inject}^{l} \cdot \mathsf{H}^{l}$, now allows a handler for l to pass over another handler for l as long as that handler is matched to an inject^{l}.

The use of handler H^{l} contexts allows us to concisely capture this notion without having to introduce explicit nesting levels. Biernacki et al. [3] formalize handler contexts using nesting levels where a handler can handle an operation if the context is "0-free". These approaches are equivalent and we can show that any handler context is 0-free by construction:

Theorem 3. (*Handler contexts are 0-free*)
For any H^{l}, we have 0-free(l, H^{l}).

We use the derivation rules of n-freeness in shown by Biernacki et al. [3] in Figure 2. To prove our theorem, we need the following lemma:

Lemma 2. (*Free extension*)
If n-free(l, E_1) and m-free(l, E_2), then $(n + m)$-free$(l, \mathsf{E}_1[\mathsf{E}_2])$.

This can be proved by straightforward induction over the length of the n-free(l, E_1) derivation and the structure of the evaluation contexts.

Proof. (*Of Theorem 3*) The proof proceeds by induction over the structure of H^{l}. Most cases are immediate – the interesting cases are the handle and inject rules:

- $\mathsf{handle}^{l} \cdot \mathsf{H}_1^{l} \cdot \mathsf{inject}^{l} \cdot \mathsf{H}_2^{l}$: by hypothesis, H_1^{l} and H_2^{l} are 0-free (**1**). $\mathsf{inject}^{l} \cdot \mathsf{H}_2^{l}$ is 1-free, and therefore by (1) and Lemma 2, $\mathsf{H}_1^{l} \cdot \mathsf{inject}^{l} \cdot \mathsf{H}_2^{l}$ is also 1-free. It follows by the handle rule that the full expression is 0-free.
- $\mathsf{inject}^{l'} \cdot \mathsf{H}^{l}$, and $\mathsf{handle}_h^{l'} \cdot \mathsf{H}^{l}$ with $l \neq l'$ (**1**): by hypothesis, H^{l} is 0-free, and with (1) it follows by the injection rule that the full expression is 0-free. □

As a consequence, the results of Biernacki et al. [3], and in particular the step-indexed relational interpretation of the calculus, carry over to our formalization without further changes.

6.1. Injection and Return

Biernacki et al. [3] remark that with injection we do not a special rule for return and can translate:

$$\mathsf{handle}^{l}_{\{h;\ \mathsf{return}\ x \to e_r\}}(e)$$

as a shorthand for:

$$\mathsf{handle}^{l}_{\{h\}}(\mathsf{val}\ x\ =\ e;\ \mathsf{inject}^{l}(e_r))$$

(although this would require the (*value*) rule in this case). However, this technique does not work for parameterized handlers where the return expression e_r can reference the handler parameter. For this reason, we prefer our original interpretation in Section 4.2 of return clauses as regular operations.

6.2. Generative Effect Types

Using *inject* also allows us to program with *first-class* effect handlers. As a running example, we are going to model a heap with polymorphic references:

```
effect heap⟨s,a⟩ {
  fun get() : a
  fun set( x : a ) : ()
}
```

The `s` parameter is a *phantom* type [30] that is used to emulate generative types [40]. The handler that creates fresh references uses a parameterized handler for the local state, but also provides an explicit type signature for the `action`:

```
fun new-ref( init : a, action : forall⟨s⟩ () → ⟨heap⟨s,a⟩|e⟩ b ) : e b {
  handle(action) (st = init) {
    get()  → resume(st,st)
    set(x) → resume((),x)
  }
}
```

The `action` gets a polymorphic *rank-2* type which ensures that we can only pass actions that are fully polymorphic in the `s` type, i.e. that are parametric with respect to the heap. Note that the use of a higher-rank type here is different for the use in the Haskell `ST` monad [22]: there the rank-2 parameter is used to ensure a reference cannot escape the heap scope but that is already done here through the effect system itself. In our case, we use it purely to work with multiple references at the same time.

We can now use *inject* to use a particular `heap` handler. For example, the $\mathtt{get}_2$ function selects the values of the innermost two heap handlers:

```
fun get₂() : ⟨heap⟨s₁,a⟩, heap⟨s₂,b⟩ | e⟩ (a,b) {
  ( get(), inject⟨heap⟩{get()} )
}
```

Note that due to the Uneq-Label rule in Figure 3 each `heap` effect is ordered in the final effect row. The `inject` construct now allows us to select a particular handler that is reflected in the effect type. For example:

```
fun test() : ⟨⟩ int {
  new-ref( 42, {
    new-ref( False, {
      val (x,y) = get₂()
      (if (x) then 1 else 0) + y
    })
  })
}
```

6.2.1. Use and Using keywords. Writing such nested higher-order functions can be a bit cumbersome and Koka provides the `using` and `use` keywords to make this more convenient. These constructs are just syntactic sugar, where:

`using` $f(e_1,\ldots,e_n)$; $body$ $\quad\rightsquigarrow\quad$ $f(e_1,\ldots,e_n,$ `fun(){` $body$ `})`
`use` x `=` $f(e_1,\ldots,e_n)$; $body$ $\quad\rightsquigarrow\quad$ $f(e_1,\ldots,e_n,$ `fun(` x `){` $body$ `})`

With this new syntactic sugar we can write our example more concisely as:

```
fun test() : ⟨⟩ int {
  using new-ref( 42 )
  using new-ref( False )
  val (x,y) = get₂()
  (if (x) then 1 else 0) + y
}
```

In this example, the type of `x` is inferred to be `bool` because of the way the handlers in `new-ref` are ordered. If we would swap their order, the example would no longer type check! Since the type inference cannot swap the order of the `heap` effect types automatically (due to Uneq-Label), we need to do this manually:

```
fun bypass( action : () → ⟨heap⟨s₂,b⟩, heap⟨s₁,a⟩, heap⟨s₂,b⟩ | e⟩ c) : ⟨heap⟨s₁,a⟩, heap⟨s₂,b⟩ | e⟩
  handle(action) {
    get()  → resume( inject⟨heap⟩{get()} )
    set(x) → resume( inject⟨heap⟩{set(x)} )
  }
}

fun test-swapped() : ⟨⟩ int {
  using new-ref( False )
  using new-ref( 42 )
  using bypass
  val (x,y) = get₂()
  (if (x) then 1 else 0) + y
}
```

Here the `bypass` function explicitly handles first `heap⟨s₂,b⟩` effect by forwarding to the handler further down using `inject` to skip over the `heap⟨s₁,a⟩` handler. The `bypass` handler is inserted after the swapped `new-ref` handlers to make the rest of the program well-typed.

Using `inject`, we are now able to explicitly refer to a particular handler and can even express first-class polymorphic references. However, there are clearly various drawbacks because each reference is expressed as a fresh `heap` effect in the effect row.

(1) Since every reference is expressed in the effect type, this may lead to very large types that are difficult to understand. (This also precludes a dynamic number of references; however this is not a real limitation as a dynamic number of references are of a particular type and we can model that using a handler that uses a list of references of that type as its local state.)
(2) To use a particular reference, we need to insert just the right amount of `inject` expressions to select it.
(3) To use abstractions we need to manually commute heap effects through the effect row.

In the following section, we are going to address these issues by using explicit values to refer to particular handlers instead of expressing it in the type only.

7. RESOURCES

In this section we are addressing the shortcomings of first-class handlers using inject expressions alone. The main idea is to enable referencing a particular handler through a regular value, called a *resource*. Building on our previous heap example, we like to give the following type to the heap operations:

```
get : (r : ref⟨a⟩ ) → ⟨heap,exn⟩ a
set : (r : ref⟨a⟩, x : a ) → ⟨heap,exn⟩ ()
```

All operations are now under a single 'umbrella' effect heap while the particular handler is identified through a *resource* of type ref⟨a⟩. The main drawback of this approach is that we lose some safety guarantees: because the particular handler is no longer reflected in the effect type, it is not statically guaranteed that the resource handler is in scope and an operation might fail dynamically at runtime with an exception. This is the reason for the additional exn effect. We feel though this is a reasonable tradeoff, similar to allowing partial functions like head. Note that even though we use the same name, our semantics of a resource is quite different from the resources that were originally present in the Eff language [2] – we discuss this further in Section 7.5.

In Koka, we can define resource effects as:

```
effect heap { /* empty */ }

effect resource ref⟨a⟩ in heap {
  fun get()       : a
  fun set(x : a) : ()
}
```

This declares an empty effect heap for the umbrella effect; in general this effect can also declare its own operations. For example, for file resources, the umbrella filesystem effect might provide the primitive operations necessary for handling files. The effect resource declaration declares a new resource ref⟨a⟩ under the heap effect with the given operations. The handler for the heap effect is trivial:

```
val heap = handler⟨heap⟩{ }
```

We can create a fresh resource by creating a handler for it:

```
fun new-ref( init : a, action : (ref⟨a⟩ → ⟨heap|e⟩ b) ) : ⟨heap|e⟩ b {
  handle resource (action) (st = init) {
    get() → resume(st,st)
    set(x) → resume((),x)
  }
}
```

We use again a parameterized handler to model the state of the reference. Because Koka infers that we handle a resource effect, the action parameter has an inferred type that takes a fresh resource ref⟨a⟩ as its first argument.

Using resources, we no longer need to use intricate inject expressions but can simply use the resources as an extra argument to the operations (where we use the use keyword as described in Section 6.2.1):

```
fun test() : exn int {
  using heap
  use x = new-ref(42)
  use y = new-ref(False)
```

```
  (if (y.get()) then 1 else 0) + x.get()
}
```

This is a big improvement compared to the approach based on `inject`. As remarked before though, the price is the appearance of the `exn` effect that signifies that we can no longer statically guarantee that a handler for a specific resource is in scope. Here is an example of a program that would fail at runtime with an exception:

```
fun wrong() : ⟨heap,exn⟩ ref⟨int⟩ {
  use x = new-ref(42)
  x  // escapes the scope of ‘new-ref‘
}

fun ouch() : exn int {
  using heap
  val y = wrong()
  y.get()
}
```

7.1. Overriding Resource Handlers

The `new-ref` function now creates a handler for a fresh resource, but it is also possible to override an existing handler for a resource. For example, we could track the contents of a particular reference:

```
fun track( r : ref⟨int⟩, action : () → ⟨heap,io|e⟩ a) : ⟨heap,io|e⟩ a {
  println("initial content: " + r.get().show )
  handle resource r (action) {
    get()  → resume( r.get() )  // pass-through
    set(x) → {
      println("new content: " + x.show )
      resume( r.set() )  // pass-through
    }
  }
}
```

By passing an extra argument `r` to the `handle resource` expression, we are handling that specific resource instead of creating a fresh one. Any operation on `r` in the action will now be handled by this handler. In the definition of each branch, we can ourselves use operations on `r` that are handled by the original handler. This gives us a very powerful extension mechanism to override behavior locally in a structured and typed way.

7.2. Injecting Resources

Now that we can override handlers for resources, it is natural to ask if it would be useful to also ‘inject’ a resource effect and being able to skip over the innermost resource handler. This functionality seems a bit far fetched but turns out to be very useful in practice.

Bračevac et al. [4] describe a reactive framework based on algebraic effect handlers. This work originally represented each event stream as a separate effect type but proved to be too cumbersome in practice. The extension of Koka with first-class resources was originally done to overcome these limitations. Now, each event stream can be represented using a resource instead. Essentially, the CorrL framework defines:

```
// empty umbrella effect
effect corrl { }
```

```
effect resource stream⟨a⟩ in corrl {
  fun push( x : a ) : ()
  ...
}
```

An important function in the CorrL framework is *alignment*. The $\texttt{align}_2$ takes two streams and aligns them: it waits until each stream performs a `push` and then actually performs both pushes and continues evaluation of each stream in an interleaved way. One way to implement this is to use local state and bypass each stream handler to wait until both have pushed:

```
fun align(s1 : stream⟨a⟩, s2 : stream⟨b⟩, action : () → ⟨corrl,pure|e⟩ () ) : ⟨corrl,pure|e⟩ ()
{
  var st1 := Nothing
  var st2 := Nothing

  val h1 = handler resource s1 {
    push(x) → match(st2) {
      Nothing → st1 := Just((x,resume))  // and do not resume
      Just((y,resumey)) → {
        s2.push(y)
        s1.push(x)
        interleaved { resume(()) } { resumey(()) }
      }
    }
  }

  val h2 = handler resource s2 {
    push(y) → match(st1) {
      Nothing → st2 := Just((y,resume))
      Just((x,resumex)) → {
        s2.push(y)
        // subtle: skip the outer ‘h1‘ handler!
        s1.inject-resource{ s1.push(x) }
        interleaved { resumex(()) } { resume(()) }
      }
    }
  }

  h1{ h2( inject-st(action) ) }
}
```

Here we use two local references to store whether either resource stream has been pushed to. Once both have been pushed to, we propagate the `push` to the original resource handler, and continue both streams in an interleaved way.

But there is twist: in the inner $\texttt{h}_2$ handler, when we want to push the value for $\texttt{s}_1$ we need to skip over the $\texttt{h}_1$ handler as we want to push to the original handler for $\texttt{s}_1$! Therefore, we use `inject-resource` to ‘inject’ a resource effect. This is automatically provided by Koka for each resource effect – here it has type:

Extended Syntax:

Expressions	e	::=	…	
		|	$\mathsf{new\ handle}^c_h(v)$	resource handler
Runtime effects	l	::=	l	effect constants
		|	x	variables

Extended Reduction Rules:

$$(new)\quad \mathsf{new\ handle}^c_h(v) \longrightarrow \mathsf{handle}^l_h(v(l)) \qquad \text{with fresh effect constant } l \text{ of type } c\langle\tau_1,\ldots,\tau_n\rangle$$

Extended Type Rules:

$$\frac{\begin{array}{l}\Gamma \vdash x : c\langle\overline{\tau}\rangle \mid \langle\rangle \qquad \Gamma \vdash e : \tau \mid \epsilon \qquad S(c) = \{op_1, \ldots, op_n\} \\ \Gamma, x_r : \tau \vdash e_r : \tau_r \mid \langle\iota\mid\epsilon\rangle \quad \Gamma \vdash op_i : (c\langle\overline{\tau}\rangle, \tau_i) \to \langle\iota, exn\rangle\ \tau_i' \mid \langle\rangle \\ \Gamma, \mathit{resume} : \tau_i' \to \epsilon\ \tau_r,\ x_i : \tau_i \vdash e_i : \tau_r \mid \langle\iota\mid\epsilon\rangle\end{array}}{\Gamma \vdash \mathsf{handle}^x\{\ op_1(x_1) \to e_1;\ \ldots;\ op_n(x_n) \to e_n;\ \mathsf{return}\ x_r \to e_r\ \}(e) : \tau_r \mid \langle\iota\mid\epsilon\rangle}\ [\textsc{Handle-Var}]$$

$$\frac{\begin{array}{l}\Gamma \vdash x : c\langle\overline{\tau}\rangle \mid \langle\rangle \qquad S(c) = \{op, \ldots\} \\ \Gamma \vdash op : (c\langle\overline{\tau}\rangle, \tau_1) \to \langle\iota, exn\rangle\ \tau_2\ \Gamma \vdash e : \tau \mid \langle\iota\mid\epsilon\rangle\end{array}}{\Gamma \vdash \mathsf{inject}^x(e) : \tau \mid \langle\iota\mid\epsilon\rangle}\ [\textsc{Inject-Var}]$$

$$\frac{\Gamma, x : c\langle\overline{\tau}\rangle \vdash \mathsf{handle}^x(e(x)) : \tau \mid \epsilon\ \text{fresh}\ x,\ \overline{\tau}}{\Gamma \vdash \mathsf{new\ handle}^c(e) : \tau \mid \epsilon}\ [\textsc{Handle-New}]$$

Fig. 8. Extension with first-class resources

```
inject-resource : stream⟨a⟩ → (() → ⟨corrl|e⟩ a) : ⟨corrl|e⟩ a
```

Since resource effects are under an umbrella effect, the inject of the resource is not reflected in the effect type; however, it has a runtime effect in that it will skip the inner-most handler for the resource just like injection for a regular effect. Writing the `align` function without resource injection is possible but more involved as one has to define an outer handler that joins both pushes and performs a push to the original handlers outside of the bypass handlers.

7.3. Formalizing Resources

It turns out that enabling first-class resources requires just minimal extensions to our original semantics as shown in Figure 8. We first add a new syntactic expression new $\mathsf{handle}^c(v)$ is the equivalent of `handle resource` in Koka and creates a fresh resource of type $c\langle\overline{\tau}\rangle$ and handles it. For simplicity we only allow value expressions v as the argument (usually a lambda expression). The second extension is to allow variables x as runtime labels l. This extension now allows us to handle specific resources using $\mathsf{handle}^x_h(e)$ (versus $\mathsf{handle}^l_h(e)$) and to inject resources using $\mathsf{inject}^x(e)$.

The operational semantics get one new transition rule (*new*) for new $\mathsf{handle}^c_h(v)$. This creates a fresh effect label l to identify the handler/resource uniquely at runtime and simply transitions

to a $\mathsf{handle}^{l}_{h}(v(l))$ expression where the argument v gets passed the new resource l as a runtime argument (of type $c\langle\tau_1, \ldots, \tau_n\rangle$).

All the other reduction rules are unchanged! The rules just depend on runtime labels l and keep working even if the labels are sometimes dynamically generated through the (*new*) rule. However, not all proofs work as before. In particular, the proof of Lemma 1.b is no longer valid: the reduction can get stuck for dynamic labels if they escape the scope of their handler. This is the reason for adding the `exn` effect to the type of their operations. In the operational semantics we may add a further rule that reduces unhandled resource operations to raising an exception.

Since the basic reduction rules are unchanged, this means we can understand and reason about resources just like regular effect handlers – there is no intrinsic extra complexity.

7.4. Type Rules

There are quite a few new type rules in Figure 8 for the resources since we need to add new rules for each syntactic construct that can now take a variable instead of just a constant effect label. The Handle-Var rule describes a resource handler and is quite similar to the Handle rule for effect constants. The main difference is that the operations now take an extra first argument that is the resource of type $c\langle\overline{\tau}\rangle$ and the effect of the operations include the *exn* effect. Moreover, in contrast to Hanlde, a resource handler does not discharge any effect and just passes the umbrella effect through.

The Handle-New rule just binds a new resource variable and refers to the Handle-Var rule to do the actual type check. Finally, the Inject-Var rule derives the umbrella effect ι by looking at the types of the operations of the resource effect. In contrast to Inject no actual effect type is injected though.

7.5. Resources in Eff

The Eff language originally [2] also had resources inspired by a co-algebraic interpretation of effects, although they are currently removed from the language (but may come back). However, the semantics of such resources was not specified in terms of ordinary handlers: instead they were handled by *default handlers* in the outer scope of the program and operation clauses could not issue operations themselves.

Recent work by Kiselyov and Sivaramakrishnan [19] embedded Eff directly in the OCaml language and they describe a more generalized notion of effect resources that is more similar to our semantics. They use a universal `New` effect as an umbrella effect for all resources to create fresh instances. They do not describe overriding resource handlers or resource injection but we believe it is possible to extend their work to allow for this. Of course, in that work there are no effect types as such, and the embedding is specified in terms of delimited continuations so it is hard to compare directly against our type system and semantics. In both approaches though we can understand resources in terms of regular effect handlers which is a nice property.

7.6. Linear Resources

As another example, we look at modeling external resources with linearity constraints, like files. The umbrella effect is the `filesystem` with various low-level functions to handle files:

```
abstract effect filesystem {
  fun fopen( path : string ) : fhandle
  fun fread( h : fhandle ) : string
  fun fclose( h : fhandle ) : ()
}
```

```
val filesystem : (action:() → ⟨filesystem|e⟩ a) → ⟨io|e⟩ =
  handler {
    fopen(path) → resume( std/os/fopen(path) )
    fread(h)    → resume( std/os/fread(h) )
    fclose(h)   → resume( std/os/fclose(h) )
  }
```

The umbrella effect is declared `abstract` to make the internal operations private to the module. A nice property of the `filesystem` handler is that it makes the interface explicit: the `action` parameter has the `filesystem` effect while the handler gets the broad `io` effect. This makes reasoning about potential side effects much more precise.

The resources in the `filesystem` are `file`s with just a single `read` operation:

```
effect resource file in filesystem {
  fun read() : string
}

fun file( path : string, action : file → ⟨filesystem|e⟩ a) : ⟨filesystem|e⟩ a {
  val h = fopen(path)
  val x = handle(action) {
            read() → resume( fread(h) )
          }
  fclose(h)
  x
}
```

The `file` handler opens a specific file, and provides a `read` operations that reads from the opened file handle. After the action is done, the file handle is closed the final result `x` is returned. For example:

```
fun file-length( path : string ) : io int {
  using filesystem
  use f = file(path)
  f.read().count
}
```

Clearly, the `file` handler is not ideal is it does not close open file handles if an exception occurs inside the `action`. It would not suffice though to install an exception handler – *any* operation called in `action` might be handled by a handler that simply does not call *resume*! Moreover, what should happen if an operation is resumed more than once and the as a result, the file handle might be closed more than once?

Dealing with linear resources is a gnarly problem for general effect handlers and we tackle it in the next section.

8. FINALIZATION

As we saw in the previous section, we need a general mechanism to finalize (and initialize) linear resources that is not specific to exceptions since any handler might decide not to resume, or resume more than once. In particular, in Koka we can rewrite the `file` handler of the previous section as:

```
fun file( path : string, action : file → ⟨filesystem|e⟩ a) : ⟨filesystem|e⟩ a {
  handle(action) (h) {
```

```
    initially → fopen(path)
    finally   → fclose(h)
    read()    → resume( fread(h) )
  }
}
```

Here we have two new special branch constructs: `initially` and `finally` that can be part of any handler. The `initially` branch is run before the `action` is called and initializes the parameterized state (`h` in the example). The `finally` branch is always executed when 'the scope is exited' and is used to close any external resources.

To be more precise, `finally` branches should be executed when either the `action` returns normally, or if any operation in `action` does not resume. The second situation is a bit tricky as it cannot be detected statically if a handler will resume or not. For example, when modeling asynchrony or concurrency [9–11, 26] the handler usually stores the `resume` function in a scheduler queue and calls those later according to some particular schedule. Another example is in Section 7.2 in the `align` function that stores the resumptions before invoking them later. At that point, a compiler cannot statically determine if such resumption will ever be called or not.

We therefore propose that handlers that really do not resume, call instead a new `finalize` function that ensures that all `finally` branches under it are executed before returning. This approach also has the advantage that the original semantics of algebraic effect handlers stays the same. As an example, the proper implementation of a handler that turns exception effects into a `maybe` type is now:

```
val to-maybe = handler {
  return x    → Just(x)
  raise(msg) → finalize(Nothing)
}
```

The argument to the `finalize` function is returned as its final result after running all `finally` branches and is there to mimic the `resume` function (and just like `resume`, `finalize` is an implicitly bound first-class function). Moreover, just like `resume`, a tail-resumptive finalizer can be implemented more efficiently. Note that we could still choose to not use `finalize` and immediately return with `Nothing` – but in that case no finalizers are run (as per the original semantics of algebraic effect handlers).

8.1. Formalizing Finally

Figure 9 extends the original semantics with finalizers. For now, we just extend with `finally` and discuss `initially` clauses later on. The syntax is extended to allow for finally $\rightarrow$ e clauses besides the return clauses. If unspecified, finally $\rightarrow$ () is used by default.

The new ($handle_f$) rule now binds an extra *finalize* identifier that, instead of resuming, runs all finalizers in the context. It does that by actually calling *resume* but with a special $\text{cancel}^l(v)$ expression (which can be regarded as a special builtin exception). The evaluation rule (*cancel*) for the "cancel" expression simply propagates the cancel value up through evaluation contexts F – these are just like regular evaluation contexts but don't include handle frames.

There is a formalization wrinkle here since *finalize* calls the resume function r with a $\text{cancel}^l(e)$ expression where we want to substitute the $\text{cancel}^l(e)$ expression *as is* without yet evaluating it.

Extended Syntax:

$$
\begin{array}{llll}
\text{Expressions} & e & ::= & \ldots \\
 & & \mid & \mathsf{cancel}^l(e) \qquad\qquad \text{cancel “exception”} \\
\\
\text{Clauses} & h & ::= & \mathsf{finally} \to e;\ \mathsf{return}\ x \to e \qquad \text{finally and return clauses} \\
 & & \mid & op(x) \to e \qquad\qquad \text{operation clause}
\end{array}
$$

Finalization Contexts:

$$\mathsf{F} ::= \Box \mid \mathsf{F}(e) \mid v(\mathsf{F}) \mid \mathsf{val}\ x\ =\ \mathsf{F};\ e \mid \mathsf{inject}^l(\mathsf{F})$$

Extended Reduction Rules:

$$
\begin{array}{lll}
(\mathit{handle}_f) & \mathsf{handle}^l_h \cdot \mathsf{H}^l \cdot op^l(v) & \longrightarrow\ e[x \mapsto v,\ \mathit{resume} \mapsto r,\ \mathit{finalize} \mapsto f] \\
 & & \quad \text{where} \\
 & & \quad (op(x) \to e)\ \in h \\
 & & \quad r\ =\ \boldsymbol{\lambda} y.\ \mathsf{handle}^l_h \cdot \mathsf{H}^l \cdot y \\
 & & \quad f\ =\ \boldsymbol{\lambda} y.\ r[\mathsf{cancel}^l(y)] \\
\\
(\mathit{cancel}) & \mathsf{F} \cdot \mathsf{cancel}^l(e) & \longrightarrow\ \mathsf{cancel}^l(e) \\
\\
(\mathit{unwind}) & \mathsf{handle}^{l'}_h \cdot \mathsf{cancel}^l(e) & \longrightarrow\ e_f;\ \mathsf{if}\ l = l'\ \mathsf{then}\ e\ \mathsf{else}\ \mathsf{cancel}^l(e) \\
 & & \quad \text{where} \\
 & & \quad (\mathsf{finally} \to e_f)\ \in h \\
\\
(\mathit{cancel}_i) & \mathsf{inject}^{l'} \cdot \mathsf{cancel}^l(e) & \longrightarrow\ \mathsf{if}\ l = l'\ \mathsf{then}\ \mathsf{cancel}^l(\mathsf{cancel}^l(e))\ \mathsf{else}\ \mathsf{cancel}^l(e)
\end{array}
$$

Fig. 9. Extension with (shallow) finalization. (This is refined in the next section)

We therefore define the resume function with an internal lambda function ($\boldsymbol{\lambda}$) and apply it using square brackets as $r[\mathsf{cancel}^l(e)]$ outside our regular reduction rules[2].

The (*unwind*) rule handles the situation where a $\mathsf{cancel}^l(v)$ value meets a handler frame by unwinding through all the handlers and executing their finalizers in sequence until the original handler l that issued the *finalize* is found.

For "return x" clauses we need to extend our previous syntactic sugar as defined in Section 4.2, and redefine:

$$\mathsf{handle}^l_{\{h;\ \mathsf{return}\ x \to e_r\}}(e)$$

as syntactic sugar for an implicit *return* operation that finalizes:

$$\mathsf{handle}^l_{\{h;\ \mathit{return}(x) \to \mathit{finalize}(e_r)\}}(\mathit{return}(e))$$

This ensures that after evaluating the return expression e_r, we will also run our own “finally” clause. In the next section we discuss further optimizations of this rule.

[2]For convenience we also use a $\boldsymbol{\lambda}$ as a regular λ if they are applied only to values but it would be more correct to bind the *resume* to $\lambda x.\ \ r[x]$ and *finalize* to $\lambda x.\ \ f[x]$ instead of just r and f respectively.

Extended Syntax:

$$\begin{array}{rrcll} \text{Expressions} & e & ::= & \ldots & \\ & & \mid & \mathsf{protect}_v(e) & \text{protect frame} \end{array}$$

Extended Contexts:

$$\begin{array}{r} \mathsf{E} ::= \ldots \mid \mathsf{protect}_f(\mathsf{E}) \\ \mathsf{H}^l ::= \ldots \mid \mathsf{protect}_f(\mathsf{H}^l) \end{array}$$

Extended Reduction Rules:

$$\begin{array}{lllll} (\mathit{handle}_d) & \mathsf{handle}_h^l \cdot \mathsf{H}^l \cdot \mathit{op}^l(v) & \longrightarrow & \mathsf{protect}_f \cdot e[x \mapsto v,\ \mathit{resume} \mapsto r,\ \mathit{finalize} \mapsto f] & \\ & & & \text{where} & \\ & & & \quad (\mathit{op}(x) \to e) \in h & \\ & & & \quad r = \lambda y.\ \mathsf{handle}_h^l \cdot \mathsf{H}^l \cdot y & \\ & & & \quad f = \lambda y.\ \text{if } \lvert r \rvert = 0 \text{ then } r[\mathsf{cancel}^l(y)] \text{ else } y & \\ (\mathit{protect}) & \mathsf{protect}_f \cdot \mathsf{cancel}^l(e) & \longrightarrow & f[\mathsf{cancel}^l(e)] & \\ (\mathit{unprotect}) & \mathsf{protect}_f \cdot v & \longrightarrow & v & \\ (\mathit{unwind}) & \mathsf{handle}_h^{l'} \cdot \mathsf{cancel}^l(e) & \longrightarrow & e_f;\ \text{if } l = l' \text{ then } e \text{ else } \mathsf{cancel}^l(e) & \\ & & & \text{where} & \\ & & & \quad (\mathsf{finally} \to e_f) \in h & \end{array}$$

Fig. 10. Update of Figure 9 with deep finalization. The (*unwind*) rule is repeated for completeness. The expression $|r|$ returns the number of calls to function r.

There are some subtle corner cases when running finalizers. In particular, the expression e_f in (*unwind*) should not be able call any operations in the handler that called *finalize*: that would cause an infinite loop as the same handler would be pushed over and over again! This is prevented in the (*unwind*) rule by not running the finally expression under the handler that defines it.

Another situation that might occur is that another operation is called in a finally clause during unwinding, that itself calls *finalize*. This situation is a bit like throwing an exception inside an exception handler itself. In that case, a new unwinding will take place but since it finalizes itself, all finally handlers will still be run in sequence (it might cut a finalization short though if the new handler is under the original one).

8.2. Deep Finalization

Unfortunately, the finalization as formalized in our previous section is not yet quite good enough. In essence, we want to ensure *finally* clauses get run reliably whenever an operation does not resume and calls *finalize* instead. This may still fail though if a finalizing operation is called from within an operation clause itself! Illustrating this is a bit complex, as we need at least 3 nested handlers to show this. For example, assume we write a handler for asynchrony where we store resumptions in a queue [10, 29], somewhat like:

```
fun async( action ) {
  handle(action) (queue = []) {
    await(f) → {
      ... // part A
      // queue this resumption
      val newqueue = queue.insert(resume,finalize)
      ... // part B
      newqueue.schedule()  // resume some other strand with an updated queue
    }
    finally → {
      // finalize any resumption still in the queue
      queue.foreach fun((resume,finalize)) {
        finalize()
      }
    }
  }
}
```

Now, suppose we use a `file` handler under the `async` handler, running under the `to-maybe` handler from the start of Section 8:

```
using to-maybe
using async
use f = file("path")
...
await(...)
```

If no exceptions happen, everything runs just fine and the finally clause in the `async` handler ensures that any outstanding strands are finalized if needed.

Unfortunately, if the B part in the `async` handler raises an exception, the handler for `to-maybe` will call `finalize` but since we are in an operation clause, the handle frame for `file` will not be on the stack and the `finally` clause is not executed! However, even if we would somehow execute the `finally` clause and finalize all outstanding strands, this still goes wrong if an exception happens in part A: in that case the `await` resumption is not stored in the `queue` and the `finally` clause will still not call `finalize` on it (which in turn means that the `finally` clause of the `file` handler is never invoked).

8.3. Protecting Operation Clauses

To ensure that finally clauses are reliably executed under finalization we need to *protect* operation clauses. When finalization happens inside the clause of an operation *and* the operation did not resume (or finalize) yet, then we should implicitly invoke *finalize* for that operation clause. We call this *deep finalization*.

In the previous example, that means that even if an exception is raised in part A, *finalize* is called automatically for that operation which ensures all *finally* clauses are executed, including the one for the current handler (which also causes all outstanding strands in the `queue` to be finalized). Moreover, we need to also ensure that a `finalize` call only resumes and finalizes once, and has no effect if called again (or after a `resume`). For example, if an exception occurs in part B, the operation clause automatically calls `finalize` but the `finalize` was also already inserted into the `queue`: when the `finally` clause runs this again causes a call to the `finalize` function for that strand (which has no effect this time around).

This is all formalized in Figure 10. We introduce a new $\text{protect}_f(e)$ stack frame that protects the expression e by running the finalizer f if needed. When we look at the new reduction rule for handlers, ($handle_d$), the new finalization function f is defined as:

$$f \;=\; \lambda\!\!\lambda y.\ \text{if } |r| = 0 \text{ then } r[\text{cancel}^l(y)] \text{ else } y$$

where the operation $|r|$ returns the number of times the resume function r has been called. This operation is very easy to implement in practice (using a mutable field that counts invocations) but cumbersome to fully formalize as it requires that the function r is shared (which we also signify with the double lambda $\lambda\!\!\lambda$). This can be done by threading a separate environment through the reduction rules that keeps track of shared beta reductions. This is similar to formalizations for the lazy evaluation [21] or optimal lambda reduction [20, 42]. For simplicity though we will treat the $|r|$ function here as an *oracle* without formalizing it fully. The finalization function f now only resumes with the $\text{cancel}^l(y)$ expression if the resume function was never invoked before, and otherwise it behaves like the identity function immediately returning its result (e.g. much like a *thunk* in lazy languages).

The evaluation of an operation clause is now under a $protect_f$ frame. The rules (*protect*) and (*unprotect*) define its behavior where the rule (*unprotect*) removes the protect frame when the operation clause returns with a result v (and thus protect frames behave like an identity function). However, if the evaluation of the operation clause leads to finalization, the rule (*protect*) ensures that the finalize function f for that operation is invoked. If the operation already resumed or finalized before, the *run once* check in the definition of the finalize function ensures that this happens at most once.

Note that the finalization function f is invoked with a $\text{cancel}^l(e)$ expression as the argument: if f resumes with a cancellation itself (through (*protect*)), the two will be nested as $\text{cancel}^l(\text{cancel}^{l'}(e))$, i.e. first finalize up to l and then continue finalizing up to l'. This is also important if handlers for the same effect are nested within each other and we need to finalize each one in order.

Since $protect_f$ frames only have an effect if the operation clause never resumes or finalizes, we can optimize an implementation by squeezing out protect frames once a resume or finalize happens. In particular, for any operation where *resume* and *finalize* refer to same r as protect_f, we have:

$$\begin{aligned}\text{protect}_f \cdot \mathsf{F} \cdot \mathit{resume}(e) &= \mathsf{F} \cdot \mathit{resume}(e)\\ \text{protect}_f \cdot \mathsf{F} \cdot \mathit{finalize}(e) &= \mathsf{F} \cdot \mathit{finalize}(e)\end{aligned}$$

This also means that if an operation is tail resumptive and has the form $\mathit{resume}(e)$ or $\mathit{finalize}(e)$, there is no need to push a protect_f frame in the first place.

8.4. Return as an Operation

As shown in Section 4.2 and 8.1, we define return clauses as syntactic sugar for a *return* operation:

$$\text{handle}^l_{\{h;\ \mathit{return}(x) \to \mathit{finalize}(e_r)\}}(\mathit{return}(e))$$

For an implementation this might seem to induce too much overhead by calling *finalize* on every return. In practice though we can optimize the implementation quite a bit. For example, the *finalize* on the *return* operation does not really need to resume but just run the finally clause (if it exists) since $\mathsf{H}^l \;=\; \Box$. Generally, we can use the following direct rule for returning with deep finalization:

$$(\mathit{return}_d)\quad \mathsf{handle}^{l}_{h}(v) \longrightarrow \begin{array}{l}\mathsf{val}\ y\ =\ \mathsf{protect}_{g}(e[x \mapsto v]);\ e_f;\ y\\ \text{where}\\ \quad (\mathsf{return}\ x \to e)\ \in h\\ \quad (\mathsf{finally} \to e_f)\ \in h\\ \quad g\ =\ \boldsymbol{\lambda} y.\ (e_f;\ y)\end{array}$$

Proof. We can prove this rule correct as follows; Starting from the translated return clause, we have:

$$\begin{array}{l}\mathsf{handle}^{l}_{h} \cdot \mathit{return}^{l}(v)\\ \quad \longrightarrow\\ \mathsf{protect}_{f} \cdot \mathit{finalize}(e_r)[x \mapsto v,\ \mathit{resume} \mapsto r,\ \mathit{finalize} \mapsto f]\\ \quad \longrightarrow\\ \mathsf{protect}_{f} \cdot f(e_r[x \mapsto v,\ \mathit{resume} \mapsto r,\ \mathit{finalize} \mapsto f])\\ \quad =\ \{\ \mathit{resume}, \mathit{finalize}\ \notin \mathsf{fv}(e_r)\ \}\\ \mathsf{protect}_{f} \cdot f(e_r[x \mapsto v])\end{array}$$

There are now three cases to consider: evaluation to a value, to a cancellation, or divergence. In case 1 the expression evaluates to a value w, $e_r[x \mapsto v] \longrightarrow^{*} w$:

$$\begin{array}{l}\mathsf{protect}_{f} \cdot f(e_r[x \mapsto v])\\ \quad \longrightarrow^{*} \{\ \mathit{hypothesis}\ \}\\ \mathsf{protect}_{f} \cdot f(w)\\ \quad \longrightarrow \{\ |r| = 0,\ \mathsf{H}^{l} = \square\}\\ \mathsf{protect}_{f} \cdot \mathsf{handle}^{l}_{h} \cdot \mathsf{cancel}^{l}(w)\\ \quad \longrightarrow\\ \mathsf{protect}_{f}(e_f;\ w)\\ \quad =\ \{\ \mathit{now}\ |r|\ > 0\ \}\\ e_f;\ w\\ \quad \longleftarrow\\ \mathsf{val}\ y\ =\ w;\ e_f;\ y\\ \quad \longleftarrow\\ \mathsf{val}\ y\ =\ \mathsf{protect}_{g}(w);\ e_f;\ y\\ \quad {}^{*}\!\longleftarrow \{\ \mathit{hypothesis}\ \}\\ \mathsf{val}\ y\ =\ \mathsf{protect}_{g}(e_r[x \mapsto v]);\ e_f;\ y\end{array}$$

In case 2 the expression evaluates to cancellation: $e_r[x \mapsto v] \longrightarrow^{*} \mathsf{cancel}^{l'}(e_c)$:

$$\begin{array}{l}\mathsf{protect}_{f} \cdot f(e_r[x \mapsto v])\\ \quad \longrightarrow^{*} \{\ \mathit{hypothesis}\ \}\\ \mathsf{protect}_{f} \cdot f(\mathsf{cancel}^{l'}(e_c))\\ \quad \longrightarrow\\ \mathsf{protect}_{f} \cdot \mathsf{cancel}^{l'}(e_c)\\ \quad \longrightarrow\\ f[\mathsf{cancel}^{l'}(e_c)]\\ \quad =\ \{\ |r|\ =\ 0\ \mathit{and}\ \mathsf{H}^{l}\ =\ \square\}\\ \mathsf{handle}^{l}_{h}(\mathsf{cancel}^{l}(\mathsf{cancel}^{l'}(e_c)))\\ \quad \longrightarrow\end{array}$$

$$
\begin{array}{l}
e_f;\ \mathsf{cancel}^{l'}(e_c) \\
\quad \longleftarrow (\mathit{cancel}) \\
\mathsf{val}\ y\ =\ (e_f;\ \mathsf{cancel}^{l'}(e_c));\ e_f;\ y \\
\quad = \\
\mathsf{val}\ y\ =\ (\boldsymbol{\lambda} y.\ e_f;\ y)[\mathsf{cancel}^{l'}(e_c)];\ e_f;\ y \\
\quad \longleftarrow \\
\mathsf{val}\ y\ =\ \mathsf{protect}_g(\mathsf{cancel}^{l'}(e_c));\ e_f;\ y \\
\quad \overset{*}{\longleftarrow} \{\ \mathit{hypothesis}\ \} \\
\mathsf{val}\ y\ =\ \mathsf{protect}_f(e_r[x \mapsto v]);\ e_f;\ y
\end{array}
$$

Case 3 for divergence is similar. □

8.5. Deep Finalization and Skip Frames

Deep finalization and skip frames also work well together. For any tail resumptive operation, we know the last action taken is the *resume* (or *finalize*) which entails that $|r|$ is always 0. From this we can derive that we can leave the (handle_t) rule (Figure 6) unchanged and only need to add an extra reduction rule for when a cancel frame meets a skip frame:

$$(\mathit{resume}_c) \quad \mathsf{skip}^{l'} \cdot \mathsf{cancel}^{l}(e) \quad \longrightarrow \quad \mathsf{cancel}^{l'}(\mathsf{cancel}^{l}(e))$$

Since we know we never resumed, a normal protect frame would cause finalization. With a skip frame we are already under the right execution context and we can simply propagate the existing cancellation by wrapping it which is very efficient.

8.6. Formalizing Initially

Initialization is the dual of finalization: whereas finalizers are concerned with operations that do not resume, the initializer is concerned with operations that resume more than once. Initially clauses can be present only on parametrized handlers where the give the initial value for the handler parameter, like the file handle in the our earlier example. The idea is now to arrange that a initially clause is re-executed whenever an operation under the handler is resumed more than once. Operations that resume more than once are rare but do occur in examples like probabilistic programming or backtracking [18].

Figure 11 formalizes the initially clauses. Since initially clauses can only be present on parameterized handlers, we also extend the syntax here with parameterized handlers together with two new evaluation rules for parameterized handlers that first evaluate the initial value of the handler parameter before evaluation the action.

There is new reduction context R for *rewinding* through the stack – it is equivalent to a regular evaluation context except it has no cases for parameterized handlers. We have a special identifier $\mathit{rewind}(e)$ that rewinds an evaluation context and re-executes any initially clauses in sequence. The rule (*rewind*) uses the R context to find the *outermost* parameterized handler and re-initializes the handler parameter with the initially expression. The (*unrewind*) rule terminates the unwinding continuing on from there.

The new reduction rule for handlers, (handle_i), now applies the *rewind* expression whenever it resumes more than once. As described in the previous section, we assume we have an operation $|r|$ that returns the number of times the resumption function r has been called.

8.7. Dynamic-wind

The Scheme language always supported delimited continuations and they have also struggled with initialization- and finalization for continuations that resumed more than once. The `unwind-protect`

Extended Syntax:

$$
\begin{array}{llll}
\text{Expressions} & e & ::= & \ldots \\
 & & & \mathsf{handle}^{\mathsf{l}}_{h}(p = e)(e) \qquad \text{Parameterized handler} \\
\text{Clauses} & h & ::= & \mathsf{initially} \to e;\ \mathsf{finally} \to e;\ \mathsf{return}\ x \to e \\
 & & \mid & op(x) \to e \qquad \text{operation clause}
\end{array}
$$

Extended Evaluation Contexts and Rewind Contexts:

$$
\begin{array}{rl}
\mathsf{E} & ::= \ldots \\
 & \mid\ \mathsf{handle}^{l}_{h}(p = \mathsf{E})(e) \mid \mathsf{handle}^{l}_{h}(p = v)(\mathsf{E}) \\
\mathsf{R} ::= & \square \mid \mathsf{R}(e) \mid v(\mathsf{R}) \mid \mathsf{val}\ x\ =\ \mathsf{R};\ e \mid \mathsf{inject}^{l}(\mathsf{R}) \mid \mathsf{handle}^{l}_{h}(\mathsf{R})
\end{array}
$$

Extended Reduction Rules:

$$
\begin{array}{lll}
(\mathit{handle}_i) & \mathsf{handle}^{l}_{h} \cdot \mathsf{H}^{l} \cdot op^{l}(v) \longrightarrow & \mathsf{protect}_{f} \cdot e[x \mapsto v,\ z \mapsto v_z,\ \mathit{resume} \mapsto r_w,\ \mathit{finalize} \mapsto f] \\
 & & \text{with} \\
 & & \quad (op(x) \to e)\ \in h \\
 & & \quad r\ =\ \boldsymbol{\lambda} y.\ \mathsf{handle}^{l}_{h} \cdot \mathsf{H}^{l} \cdot y \\
 & & \quad f\ =\ \boldsymbol{\lambda} y.\ \mathsf{if}\ |r| = 0\ \mathsf{then}\ r[\mathsf{cancel}^{l}(y)]\ \mathsf{else}\ y \\
 & & \quad r_w\ =\ \boldsymbol{\lambda} y.\ \mathsf{if}\ |r|\ =\ 0\ \mathsf{then}\ r[y]\ \mathsf{else}\ \mathsf{rewind} \cdot r[y] \\
(\mathit{rewind}) & \mathsf{rewind} \cdot \mathsf{R} \cdot \mathsf{handle}^{l}_{h}(p = v_p) \cdot e \longrightarrow & \mathsf{R} \cdot \mathsf{handle}^{l}_{h}(p = e_i)(\mathsf{rewind} \cdot e) \\
 & & \text{with} \\
 & & \quad (\mathsf{initially} \to e_i)\ \in h \\
(\mathit{unrewind}) & \mathsf{rewind} \cdot \mathsf{R} \cdot v \longrightarrow & \mathsf{R} \cdot v
\end{array}
$$

Fig. 11. Extension with initialization

in Scheme is like a `finally` clause, while `dynamic-wind` is like `initially/finally` with a pre- and postlude [6, 12, 17, 35]. Sitaram [43] describes how the standard `dynamic-wind` is not good enough in general: "*While this may seem like a natural extension of the first-order unwind-protect to a higher-order control scenario, it does not tackle the pragmatic need that unwind-protect addresses, namely, the need to ensure that a kind of 'clean-up' happens only for those jumps that* ***significantly exit*** *the block, and not for those that are a* ***minor excursion****. The crux is identifying which of these two categories a jump falls into.*". Interestingly, this is exactly what is addressed by algebraic effects where "significant exit"s are operations that do not resume, while "minor excursions" are regular operations that resume with a result.

8.8. Safety of Finalization

The semantics for deep finalization and initialization address many requirements for robust handling of external resources (even in the presence of multiple resumptions). Moreover, it is defined as a dynamic untyped semantics and does not depend on static properties like linearity constraints. Nevertheless there are some aspects that we would like to improve upon:

(1) As defined, there is no guarantee that all finally clauses are executed – this only happens if operation clauses that do not resume use *finalize* correctly. This is not a bad property persé as it is exactly what gives the expressive power to define abstractions like asynchronous execution. Nevertheless, in practice one would like to prevent accidentally forgetting about a *finalize* call. Actual implementations might opt for special syntax or static checks to invert this situation. For example by wrapping every operation clause by default with an implicit *finalize* to guarantee proper finalization if *resume* (or *finalize*) was not called explicitly; i.e. the programmer must declare explicitly the intention to *not* doing finalization in an operation clause instead of the other way around.
(2) The definition of *resume* and *finalize* behave differently on the first invocation than later on. This is unsatisfying from a semantic perspective (although still deterministic). As shown in previous sections, the property is important in practice but we would like to see if is possible to capture this behavior in a more declarative way.

We would like to show that wrapping every operation clause with a *finalize* leads to robust evaluation of every finally clause. First define results w as:

$$w ::= v \mid \mathsf{cancel}^l(e)$$

Then we can state the following theorem:

Theorem 4. (*Deep finalization is robust*)
For any evaluation that does not get stuck or diverges (i.e. it evaluates to a w), and where every operation clause is of the form *finalize*(e), then, if $\mathsf{E} \cdot \mathsf{handle}^l_h \cdot e \longmapsto^* w$ we have $\mathsf{E} \cdot \mathsf{handle}^l_h \cdot e \longmapsto^* \mathsf{E}' \cdot e_f \longmapsto^* w$, where $(\text{finally} \to e_f) \in h$.

A formal proof of this property is work in progress, but we have done an initial proof sketch using induction over the number of handlers in an expression.

9. CONCLUSION

We presented a formal semantics for various optimizations and extensions to algebraic effects that are all important for any practical implementations. We hope to experiment more with first-class algebraic resources and plan to investigate more optimized implementations based on virtual method tables for each effect.

Finally, an attractive property of algebraic effect handlers is that they have a solid semantic foundation in category theory – but the extended rules for deep finalization and initialization in this article are "just" operational reductions that we defined in a particular way. We hope that it is possible to find again the foundational categorical structures to capture the semantics of these reduction rules and that can lead to a more foundational framework for reasoning about linear resources.

Acknowledgements

We would like to thank Oliver Bračevac for explaining the intricacies of reactive programming with effect handlers leading to the design of first-class resources.

REFERENCES

[1] Kenichi Asai, and Yukiyoshi Kameyama. "Polymorphic Delimited Continuations." In *Proceedings of the 5th Asian Conference on Programming Languages and Systems*, 239–254. APLAS'07. Singapore. 2007. doi:10.1007/978-3-540-76637-7_16.

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

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

[4] Oliver Bračevac, Nada Amin, Guido Salvaneschi, Sebastian Erdweg, Patrick Eugster, and Mira Mezini. "Effectful Reactive Programming." 2018. `http://www.st.informatik.tu-darmstadt.de/artifacts/corrl/corrl_draft.pdf`. Draft article.

[5] Jonathan Immanuel Brachthäuser, and Philipp Schuster. "Effekt: Extensible Algebraic Effects in Scala." In *Scala'17*. Vancouver, CA. Oct. 2017.

[6] William D. Clinger. "Implementation of `unwind-Protect` in Portable Scheme." 2003. `http://www.ccs.neu.edu/home/will/UWESC/uwesc.sch`.

[7] Olivier Danvy, and Andrzej Filinski. *A Functional Abstraction of Typed Contexts*. DIKU, University of Copenhagen. 1989.

[8] Olivier Danvy, and Andrzej Filinski. "Abstracting Control." In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, 151–160. LFP '90. Nice, France. 1990. doi:10.1145/91556.91622.

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

[10] Stephen Dolan, Spiros Eliopoulos, Daniel Hillerström, Anil Madhavapeddy, KC Sivaramakrishnan, and Leo White. "Concurrent System Programming with Effect Handlers." In *Proceedings of the Symposium on Trends in Functional Programming*. TFP'17. May 2017.

[11] Stephen Dolan, Leo White, KC Sivaramakrishnan, Jeremy Yallop, and Anil Madhavapeddy. "Effective Concurrency through Algebraic Effects." In *OCaml Workshop*. Sep. 2015.

[12] Iulian Dragos, Antonio Cunei, and Jan Vitek. "Continuations in the Java Virtual Machine." *Second ECOOP Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems (ICOOOLPS'2007)*. Technische Universität Berlin, Berlin. 2007.

[13] Ben R. Gaster, and Mark P. Jones. *A Polymorphic Type System for Extensible Records and Variants*. NOTTCS-TR-96-3. University of Nottingham. 1996.

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

[15] J.R. Hindley. "The Principal Type Scheme of an Object in Combinatory Logic." *Trans. of the American Mathematical Society* 146 (December): 29–60. Dec. 1969. doi:10.2307/1995158.

[16] Ohad Kammar, Sam Lindley, and Nicolas Oury. "Handlers in Action." In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, 145–158. ICFP '13. ACM, New York, NY, USA. 2013. doi:10.1145/2500365.2500590.

[17] Richard Kelsey, William Clinger, and Jonathan Rees. "Revised^5 Report on the Algorithmic Language Scheme." 1998. chapter 6.4.

[18] Oleg Kiselyov, and Chung-chieh Shan. "Embedded Probabilistic Programming." In *Domain-Specific Languages*. 2009. doi:10.1007/978-3-642-03034-5_17.

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

[20] John Lamping. "An Algorithm for Optimal Lambda Calculus Reduction." In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 16–30. POPL '90. San Francisco, California, USA. 1990. doi:10.1145/96709.96711.

[21] John Launchbury. "A Natural Semantics for Lazy Evaluation." In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 144–154. POPL '93. Charleston, South Carolina, USA. 1993. doi:10.1145/158511.158618.

[22] John Launchbury, and Amr Sabry. "Monadic State: Axiomatization and Type Safety." In *In Proceedings of the 2nd ACM SIGPLAN International Conference on Functional Programming*, 227–238. ICFP'97. 1997. doi:10.1145/258948.258970.

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

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

[25] Daan Leijen. "The Koka Repository." 2016. `https://github.com/koka-lang/koka`, the extensions described in this paper are available in the `dev` branch with examples in the `test/resource` directory.

[26] Daan Leijen. "Structured Asynchrony with Algebraic Effects." In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Type-Driven Development*, 16–29. TyDe 2017. Oxford, UK. 2017. doi:10.1145/3122975.3122977.

[27] Daan Leijen. "Implementing Algebraic Effects in C." In *Programming Languages and Systems*, edited by Bor-Yuh Evan Chang, 339–363. Springer International Publishing. 2017.

[28] Daan Leijen. "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. Jan. 2017. doi:10.1145/3009837.3009872.

[29] Daan Leijen. *Structured Asynchrony with Algebraic Effects*. {MSR-TR-2017-21}. Microsoft Research. May 2017.

[30] Daan Leijen, and Erik Meijer. "Domain Specific Embedded Compilers." In *2nd USENIX Conference on Domain Specific Languages (DSL'99)*, 109–122. Austin, Texas. Oct. 1999. `http://www.cs.uu.nl/people/daan/pubs.html`. Also in ACM SIGPLAN Notices 35, 1, (Jan. 2000).

[31] Sam Lindley, and James Cheney. "Row-Based Effect Types for Database Integration." In *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation*, 91–102. TLDI'12. 2012. doi:10.1145/2103786.2103798.

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

[33] Robin Milner. "A Theory of Type Polymorphism in Programming." *Journal of Computer and System Sciences* 17: 248–375. 1978. doi:10.1016/0022-007890014-4.

[34] Eugenio Moggi. "Notions of Computation and Monads." *Information and Computation* 93 (1): 55–92. 1991. doi:10.1016/0890-5401(91)90052-4.

[35] Kent Pitman. "Unwind-Protect versus Continuations." 2003. `http://www.nhplace.com/kent/PFAQ/unwind-protect-vs-continuations-original.html`.

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

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

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

[39] Didier Rémy. "Type Inference for Records in Natural Extension of ML." In *Theoretical Aspects of Object-Oriented Programming*, 67–95. 1994. doi:10.1.1.48.5873.

[40] Andreas Rossberg, Claudio Russo, and Derek Dreyer. "F-Ing Modules." *Journal of Functional Programming* 24 (5): 529–607. Nov. 2014.

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

[42] Olin Shivers, and Mitchell Wand. "Bottom-up β-Reduction: Uplinks and λ-DAGs." In *Proceedings of the 14th European Conference on Programming Languages and Systems*, 217–232. ESOP'05. Edinburgh, UK. 2005. doi:10.1007/978-3-540-31987-0_16.

[43] Dorai Sitaram. "Unwind-Protect in Portable Scheme." In *Proceedings of the Scheme Workshop*. Nov. 2003.

[44] Martin Sulzmann. *Designing Record Systems*. YALEU/DCS/RR-1128. Yale University. Apr. 1997.

[45] Wouter Swierstra. "Data Types à La Carte." *Journal of Functional Programming* 18 (4): 423–436. Jul. 2008. doi:10.1017/S0956796808006758.

[46] Leo White. "Effect Types for OCaml." Sep. 2016. `https://github.com/lpw25/ocaml-typed-effects`.

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

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