

# Effect Handlers, Evidently

NINGNING XIE, Microsoft Research, USA

JONATHAN IMMANUEL BRACHTHÄUSER, University of Tübingen, Germany

DANIEL HILLERSTRÖM, The University of Edinburgh, United Kingdom

PHILIPP SCHUSTER, University of Tübingen, Germany

DAAN LEIJEN, Microsoft Research, USA

Algebraic effect handlers are a powerful way to incorporate effects in a programming language. Sometimes perhaps even *too* powerful. In this article we define a restriction of general effect handlers with *scoped resumptions*. We argue one can still express all important effects, while improving local reasoning about effect handlers. Using the newly gained guarantees, we define a sound and coherent evidence translation for effect handlers which directly passes the handlers as evidence to each operation. We prove full soundness and coherence of the translation into plain lambda calculus. The evidence in turn enables efficient implementations of effect operations; in particular, we show we can execute tail-resumptive operations *in place* (without needing to capture the evaluation context), and how we can replace the runtime search for a handler by indexing with a constant offset

## 1 INTRODUCTION

Algebraic effects [Plotkin and Power 2003] and the extension with handlers [Plotkin and Pretnar 2013], are a powerful way to incorporate effects in programming languages. Algebraic effect handlers can express any free monad in a concise and composable way, and can be used to express complex control-flow, like exceptions, asynchronous I/O, local state, backtracking, and much more.

Even though there are many language implementations of algebraic effects, like Koka [Leijen 2014], Eff [Pretnar 2015], Frank [Lindley et al. 2017], Links [Lindley and Cheney 2012], and Multicore OCaml [Dolan et al. 2015], the implementations may not be as efficient as one might hope. Generally, handling effect operations requires a linear search at runtime to the innermost handler. This is a consequence of the core operational rule for algebraic effect handlers:

$$\text{handle}_m h E[\text{perform } op \ v] \longrightarrow f \ v \ k$$

requiring that  $(op \rightarrow f)$  is in the handler  $h$  and that  $op$  is not in the bound operations in the evaluation context  $E$  (so the innermost handler gets to handle the operation). The operation clause  $f$  gets passed the operation argument  $v$  and the resumption  $k = \lambda x. \text{handle}_m h E[x]$ . Inspecting this rule, we can see that implementations need to search through the evaluation context to find the innermost handler, capture the context up to that point as the resumption, and can only then invoke the actual operation clause  $f$ . This search often is linear in the size of the stack, or in the number of intermediate handlers in the context  $E$ .

In prior work, it has been shown that the vast majority of operations can be implemented much more efficiently, often in time constant in the stack size. Doing so, however, requires an intricate runtime system [Dolan et al. 2015; Leijen 2017a] or explicitly passing handler implementations, instead of dynamically searching for them [Brachthäuser et al. 2018; Schuster et al. 2019; Zhang and Myers 2019]. While the latter appears to be an attractive alternative to implement effect handlers, a correspondence between handler passing and dynamic handler search has not been formally established in the literature.

In this article, we make this necessary connection and thereby open up the way to efficient compilation of effect handlers. We identify a simple restriction of general effect handlers, called *scoped resumptions*, and we show that under this restriction we can perform a sound and coherent *evidence translation* for effect handlers. In particular:

- The ability of effect handlers to capture the resumptions  $k$  as a first-class value is very powerful – perhaps *too* powerful as it can interfere with the ability to do local reasoning. We define the notion of *scoped resumptions* (Section 2.2) as a restriction of general effect handlers where resumptions can only be applied under the scope of their original handler context. We believe all important effect handlers can be written with scoped resumptions, while at the same time ruling out many “wild” applications that have non-intuitive semantics. In particular, it no longer allows handlers that change semantics of *other* operations than the ones it handles itself. This improves the ability to use local reasoning over effects, and the coherence of evidence translation turns out to only be preserved under scoped resumptions (more precisely: an evidence translated program does not get stuck if resumptions are scoped). In this paper, we focus on the evidence translation and use a dynamic check in our formalism. We show various designs on how to check this property statically, but leave full exploration of such a check to future work.
- To open up the way to more efficient implementations, we define a type directed *evidence translation* (Section 4) where the handlers are passed down as an implicit parameter to all operation invocations; similar to the dictionary translation in Haskell for type classes [Jones 1992], or capability passing in Effekt [Brachthäuser et al. 2020]. This turns out to be surprisingly tricky to get right, and we describe various pitfalls in Section 4.2. We prove that our translation is sound (Theorem 4 and 7) and coherent (Theorem 8), and that the evidence provided at runtime indeed always corresponds exactly to the dynamic innermost handler in the evaluation context (Theorem 5). In particular, on an evaluation step:

$$\text{handle}_m h E[\text{perform } op \text{ ev } v] \longrightarrow f v k \quad \text{with } op \notin \text{bop}(E) \wedge (op \rightarrow f) \in h$$

the provided evidence  $ev$  will be exactly the pair  $(m, h)$ , uniquely identifying the actual (dynamic) handler  $m$  and its implementation  $h$ . This is the essence to enabling further optimizations for efficient algebraic effect handlers.

Building on the coherent evidence translation, we describe various techniques for more efficient implementations (Section 6):

- In practice, the majority of effects is *tail resumptive*, that is, their operation clauses have the form  $op \rightarrow \lambda x. \lambda k. k e$  with  $k \notin e$ . That is, they always resume once in the end with the operation result. Note that  $e$  may use  $x$  or perform operations itself, as it has already captured (closed over) the specific evidence it needs when the handler was instantiated. We can execute such tail resumptive operation clauses *in place*, e.g.

$$\text{perform } op (m, h) v \longrightarrow f v (\lambda x. x) \quad (op_{\text{tail}} \rightarrow f) \in h$$

This is of course an important optimization that enables truly efficient effect operations at a cost similar to a virtual method call (since we can implement handlers  $h$  as a vector of function pointers where  $op$  is at a constant offset such that  $f = h.op$ ).

- Generally, evidence is passed as an *evidence vector*  $w$  where each element is the evidence for a specific effect. That means we still need to select the right evidence at run-time which is a linear time operation (much like the dynamic search for the innermost handler in the evaluation context). We show that by keeping the evidence vectors in canonical form, we can index the evidence in the vector at a *constant offset* for any context where the effect is non-polymorphic.
- Since the evidence provides the handler implementation directly, it is no longer needed in the context. We can follow Brachthäuser and Schuster [2017] and use an implementation based

on multi-prompt delimited continuations [Dyvig et al. 2007; Gunter et al. 1995] instead. Given evidence  $(m, h)$ , we directly yield to a specific prompt  $m$ :

```

handlem h E[perform op (m, h) v]
  ~
promptm E[yieldm (λk. (h.op) v k)]

```

We define a *monadic multi-prompt translation* (Section 5) from an evidence translated program (in  $F^{ev}$ ) into standard call-by-value polymorphic lambda calculus ( $F^v$ ) where the monad implements the multi-prompt semantics, and we prove that this monadic translation is sound (Theorem 10) and coherent (Theorem 11). Such translation is very important, as it provides the missing link between traditional implementations based on dynamic search for the handler [Dolan et al. 2015; Leijen 2014; Lindley et al. 2017] and implementations of lexical effect handlers using multi-prompt delimited control [Biernacki et al. 2019; Brachthäuser and Schuster 2017; Zhang and Myers 2019]. It also means we can use a standard compilation backend where all usual optimizations apply that would not hold under algebraic effect semantics directly (since all effects become explicit now). For example, as all handlers become regular data types, and evidence is a regular parameter, standard optimizations like inlining can often completely inline the operation clauses at the call site without any special optimization rules for effect handlers [Pretnar et al. 2017]. Moreover, no special runtime system for capturing the evaluation context is needed anymore, like split-stacks [Dolan et al. 2015] or stack copying [Leijen 2017a], and we can generate code directly for any host platform (including C or WebAssembly). In particular, recent advances in compilation guided reference counting [Ullrich and Moura 2019] can readily be used. Such reference counting transformations cannot be applied to traditional effect handler semantics since any effect operation may not resume (or resume more than once), making it impossible to track the reference counts directly.

We start by giving an overview of algebraic effects and handlers and their semantics in an untyped calculus  $\lambda^e$  (Section 2), followed by a typed polymorphic formalization  $F^e$  (Section 3) for which we prove various theorems like soundness, preservation, and the meaning of effect types. In Section 4 we define an extension of  $F^e$  with explicit evidence vector parameters, called  $F^{ev}$ , define a formal evidence passing translation, and prove this translation is coherent and preserves the original semantics. Using the evidence translated programs, we define a coherent monadic translation in Section 5 (based on standard multi-prompt semantics) that translates into standard call-by-value polymorphic lambda-calculus (called  $F^v$ ). Section 6 discusses various immediate optimization techniques enabled by evidence passing, in particular tail-resumption optimization, effect-selective monadic translation, and bind-inlining to avoid explicit allocation of continuations.

For space reasons, we put all evaluation context type rules and the full proofs of all stated lemmas and theorems in our Appendix, which also includes further discussion of possible extensions.

## 2 UNTYPED ALGEBRAIC EFFECT HANDLERS

We begin by formalizing a minimal calculus of untyped algebraic effect handlers, called  $\lambda^e$ . The formalization helps introduce the background, sets up the notations used throughout the paper, and enables us to discuss examples in a more formal way.

The formalization of  $\lambda^e$  is given in Figure 1. It is essentially standard call-by-value lambda calculus extended with a rule to perform operations and a rule to handle them. It corresponds closely to the untyped semantics of Forster et al. [2019], and the effect calculus presented by Leijen [2017c]. Sometimes, effect handler semantics are given in a form that does not use evaluation contexts, e.g.

Expressions		Values	
$e ::= v$	(value)	$v ::= x$	(variables)
$e e$	(application)	$\lambda x. e$	(functions $f$ )
$\text{handle } h e$	(handler instance)	$\text{handler } h$	(effect handler)
		$\text{perform } op$	(operation)
Handlers		$h ::= \{op_1 \rightarrow f_1, \dots, op_n \rightarrow f_n\}$	(operation clauses)
Evaluation Context		$F ::= \square \mid F e \mid v F$	(pure evaluation)
		$E ::= \square \mid E e \mid v E \mid \text{handle } h E$	(effectful computation)
(app)	$(\lambda x. e) v$	$\longrightarrow e[x:=v]$	
(handler)	$(\text{handler } h) v$	$\longrightarrow \text{handle } h \cdot v ()$	
(return)	$\text{handle } h \cdot v$	$\longrightarrow v$	
(perform)	$\text{handle } h \cdot E \cdot (\text{perform } op) v$	$\longrightarrow f v k$	iff $op \notin \text{bop}(E) \wedge (op \rightarrow f) \in h$ where $k = \lambda x. (\text{handle } h \cdot E \cdot x)$
$\frac{e \longrightarrow e'}{E \cdot e \longmapsto E \cdot e'} \text{ [STEP]}$		$\text{bop}(\square) = \emptyset$	
		$\text{bop}(E e) = \text{bop}(E)$	
		$\text{bop}(v E) = \text{bop}(E)$	
		$\text{bop}(\text{handle } h E) = \text{bop}(E) \cup \{op \mid (op \rightarrow f) \in h\}$	

Fig. 1.  $\lambda^\epsilon$ : Untyped Algebraic Effect Handlers

[Kammar and Pretnar 2017; Pretnar 2015], but in the end both formulations are equivalent (except that using evaluation contexts turns out to be convenient for our proofs).

There are two differences to earlier calculi: we leave out return clauses (for simplicity) and instead of one  $\text{handle } h$  expression we distinguish between  $\text{handle } h e$  (as an expression) and  $\text{handler } h$  (as a value). A  $(\text{handler } h) v$  evaluates to  $\text{handle } h (v ())$  and just invokes its given function  $v$  with a unit value under a  $\text{handle } h$  frame. As we will see later, handler is generative and instantiates  $\text{handle}$  frames with a unique marker. As such, we treat  $\text{handle}$  as a strictly internal frame that only occurs during evaluation.

The evaluation contexts consist of *pure* evaluation contexts  $F$  and effectful evaluation contexts  $E$  that include  $\text{handle } h E$  frames. We assume a set of operation names  $op$ . The  $(\text{perform } op) v$  construct calls an effect operation  $op$  by passing it a value  $v$ . Operations are handled by  $\text{handle } h e$  expressions, which can be seen in the *(perform)* rule. Here, the condition  $op \notin \text{bop}(E)$  ensures that the *innermost*  $\text{handle}$  frame handles an operation. To evaluate an operation call, evaluation continues with the body of the *operation clause*  $(op \rightarrow f)$ , passing the argument value  $v$  and the *resumption*  $k$  to  $f$ . Note that  $f v k$  is not evaluated under the handler  $h$ , while the resumption always resumes under the handler  $h$  again; this describes the semantics of *deep* handlers and correspond to a *fold* in a categorical sense (as opposed to *shallow* handlers that are more like a *case*) [Kammar et al. 2013].

For conciseness, we often use the *dot notation* to decompose and compose evaluation contexts, which also conveys more clearly that an evaluation context essentially corresponds to a runtime stack. For example, we would write  $v \cdot \text{handle } h \cdot E \cdot e$  as a shorthand for  $v (\text{handle } h (E[e]))$ . The dot notation can be defined as:

$$\begin{aligned} E \cdot e &\doteq E[e] & v \square \cdot E &\doteq v \cdot E &\doteq v E \\ \square e \cdot E &\doteq E e & \text{handle } h \square \cdot E &\doteq \text{handle } h \cdot E &\doteq \text{handle } h E \end{aligned}$$

## 2.1 Examples

Here are some examples of common effect handlers. Almost all practical uses of effect handlers are a variation of these.

**Exceptions:** Assuming we have data constructors just and nothing, we can define a handler for exceptions that converts any exceptional computation  $e$  to either just  $v$  on success, or nothing on an exception:

$$\text{handler } \{ \text{throw} \rightarrow \lambda x. \lambda k. \text{nothing} \} (\lambda_. \text{just } e)$$

For example using  $e = \text{perform throw}()$  evaluates to nothing while  $e = 1$  evaluates to just 1.

**Reader:** In the exception example we just ignored the argument and the resumption of the operation but the *reader* effect uses the resumption to resume with a result:

$$\text{handler } \{ \text{get} \rightarrow \lambda x. \lambda k. k \ 1 \} (\lambda_. \text{perform get}() + \text{perform get}())$$

Here we handle the *get* operation to always return 1 so the evaluation proceeds as:

$$\begin{aligned} & \text{handler } \{ \text{get} \rightarrow \lambda x. \lambda k. k \ 1 \} (\lambda_. \text{perform get}() + \text{perform get}()) \\ \mapsto^* & \text{handle } h \cdot \text{perform get}() + \text{perform get}() \\ \mapsto^* & (\lambda x. \text{handle } h \cdot (\square + \text{perform get}()) \cdot x) \ 1 \\ \mapsto & \text{handle } h \cdot (\square + \text{perform get}()) \cdot 1 \\ \mapsto^* & \text{handle } h \cdot (1 + \square) \cdot 1 \\ \mapsto^* & 2 \end{aligned}$$

**State:** The *state* effect is more involved with pure effect handlers as we need to return functions from the operation clauses (essentially as a state monad) (variant 1):

$$\begin{aligned} h = \{ & \text{get} \rightarrow \lambda x. \lambda k. (\lambda y. k \ y \ y), \text{set} \rightarrow \lambda x. \lambda k. (\lambda y. k \ () \ x) \} \\ & (\text{handler } h \ (\lambda_. (\text{perform set } 21; x \leftarrow \text{perform get}(); (\lambda y. x + x))) \ 0 \end{aligned}$$

where we assume  $x \leftarrow e_1; e_2$  as a shorthand for  $(\lambda x. e_2) \ e_1$ , and  $e_1; e_2$  for  $(\_ \leftarrow e_1; e_2)$ .

The evaluation of an operation clause now always returns directly with a function that takes the current state as its input; which is then used to resume with:

$$\begin{aligned} & (\text{handler } h \ (\lambda_. \text{perform set } 21; x \leftarrow \text{perform get}(); (\lambda y. x + x))) \ 0 \\ \mapsto^* & (\square \ 0) \cdot \text{handle } h \cdot (\square; x \leftarrow \text{perform get}(); (\lambda y. x + x)) \cdot \text{perform set } 21 \\ \mapsto^* & (\square \ 0) \cdot (\lambda y. k \ () \ 21) \quad \text{with } k = \lambda x. \text{handle } h \cdot (\square; x \leftarrow \text{perform get}(); (\lambda y. x + x)) \cdot x \\ = & (\lambda y. k \ () \ 21) \ 0 \\ \mapsto & k \ () \ 21 \\ \mapsto & (\text{handle } h \cdot (\square; x \leftarrow \text{perform get}(); (\lambda y. x + x)) \cdot ()) \ 21 \\ = & (\square \ 21) \cdot \text{handle } h \cdot ((); x \leftarrow \text{perform get}(); (\lambda y. x + x)) \\ \mapsto & 42 \end{aligned}$$

Clearly, defining local state as a function is quite cumbersome, so usually one allows for *parameterized handlers* [Leijen 2016; Plotkin and Pretnar 2013] that keep a local parameter  $p$  with their handle frame, where the evaluation rules become:

$$\begin{aligned} \text{phandler } h \ v' \ v & \longrightarrow \text{phandle } h \ v' \cdot v \ () \\ \text{phandle } h \ v' \cdot E \cdot \text{perform op } v & \longrightarrow f \ v' \ v \ k \quad \text{iff } \text{op} \notin \text{bop}(E) \wedge (\text{op} \rightarrow f) \in h \end{aligned}$$

where  $k = \lambda y \ x. (\text{phandle } h \ y \cdot E \cdot x)$ . Here the handler parameter  $v'$  is passed to the operation clause  $f$  and later restored in the resumption which now takes a fresh parameter  $y$  besides the result value  $x$ . With a parameterized handler the state effect can be concisely defined as (variant 2):

$$\begin{aligned} h = \{ & \text{get} \rightarrow \lambda y \ x \ k. k \ y \ y, \text{set} \rightarrow \lambda y \ x \ k. k \ x \ () \} \\ & \text{phandler } h \ 0 \ (\lambda_. \text{perform set } 21; x \leftarrow \text{perform get}(); x + x) \end{aligned}$$

Another important advantage in this implementation is that the state effect is now *tail resumptive* which is very beneficial for performance (as shown in the introduction).

There as yet another elegant way to implement local state by Biernacki et al. [2017], where the *get* and *set* operations are defined in separate handlers (variant 3):

```


$$h_1 = \{ \text{get} \rightarrow \lambda\_k. k \ 0 \}$$


$$h_2 = \{ \text{set} \rightarrow \lambda x \ k. \text{handler } \{ \text{get} \rightarrow \lambda\_k. k \ x \} (\lambda\_k. k \ ()) \}$$


$$\text{handler } h_1 (\lambda\_k. \text{handler } h_2 (\lambda\_k. \text{perform set } 42; x \leftarrow \text{perform get } (); x + x))$$


```

The trick here is that every *set* operation installs a fresh handler for the *get* operation and resumes under that (so the innermost *get* handler always contains the latest state). Even though elegant, there are some drawbacks to this encoding: a naive implementation may use  $n$  handler frames for  $n$  set operations, typing this example is tricky and usually requires *masking* [Biernacki et al. 2017; Hillerström and Lindley 2016], and, as we will see, it does not use *scoped resumptions* and thus cannot be used with evidence translation.

**Backtracking:** By resuming more than once, we can implement backtracking using algebraic effects. For example, the *amb* effect handler collects all possible results in a list by resuming the *flip* operation first with a true result, and later again with a false result:

```


$$\text{handler } \{ \text{flip} \rightarrow \lambda\_k. xs \leftarrow k \ \text{true}; ys \leftarrow k \ \text{false}; xs ++ ys \}$$


$$(\lambda\_x. x \leftarrow \text{perform flip } (); y \leftarrow \text{perform flip } (); [x \ \&\& \ y])$$


```

returning the list [true, false, false, false] in our example. This technique can also be used for probabilistic programming [Kiselyov and Shan 2009].

**Async:** We can use resumptions  $k$  as first class values and for example store them into a queue to implement cooperative threads [Dolan et al. 2017] or asynchronous I/O [Leijen 2017b]. Assuming we have a state handler  $h_{\text{queue}}$  that maintains a queue of pending resumptions, we can implement a mini-scheduler as:

```


$$h_{\text{async}} = \{ \text{fork} \rightarrow \lambda f \ k. \text{perform enqueue } k; \text{schedule } f; \text{next } ()$$


$$\text{yield} \rightarrow \lambda\_k. \text{perform enqueue } k; \text{next } () \}$$


$$\text{next} = \lambda\_k. k \leftarrow \text{perform dequeue } (); k \ ()$$


```

where *enqueue* enqueues a resumption  $k$ , and *dequeue* () returns one, or returns an identity function if the queue is empty. The *schedule* function runs a new action  $f$  under the *async* handler:

```


$$\text{schedule} = \lambda f. \text{handler } h_{\text{async}} (\lambda\_f. f \ ())$$


$$\text{async} = \lambda f. \text{handler } h_{\text{queue}} (\lambda\_k. \text{schedule } f)$$


```

The main wrapper *async* schedules an action under a fresh scheduler queue handler  $h_{\text{queue}}$ , which is shared by all forked actions under it.

## 2.2 Scoped Resumptions

The ability of effect handlers to capture the resumption as a first-class value is very powerful – and can be considered as perhaps *too* powerful. In particular, it can be (ab)used to define handlers that change the semantics of *other* handlers that were defined and instantiated orthogonally. Take for example an operation  $op_1$  that is expected to always return the same result, say 1. We can now define another operation  $op_{\text{evil}}$  that changes the return value of  $op_1$  after it is invoked! Consider the following program where we leave  $f$  and  $h_{\text{evil}}$  undefined for now:

```


$$h_1 = \{ op_1 \rightarrow \lambda x \ k. k \ 1 \}$$


$$e = \text{perform } op_1 \ (); \text{perform } op_{\text{evil}} \ (); \text{perform } op_1 \ ()$$


$$f (\text{handler } h_1 (\lambda\_k. \text{handler } h_{\text{evil}} (\lambda\_k. e)))$$


```



Even though  $h_1$  is defined as a pure reader effect and defined orthogonal to any other effect, the  $op_{evil}$  operation can still cause the second invocation of  $op_1$  to return 2 instead of 1! In particular, we can define  $f$  and  $h_{evil}$  as <sup>1</sup>:

$$\begin{aligned} h_2 &= \{ op_1 \rightarrow \lambda x k. k \ 2 \} \\ h_{evil} &= \{ op_{evil} \rightarrow \lambda x k. k \} \\ f &= \lambda k. \text{handler } h_2 (\lambda_. k ()) \end{aligned}$$

The trick is that the handler  $h_{evil}$  does not directly resume but instead returns the resumption  $k$  as is, after unwinding through  $h_1$  it is passed to  $f$  which now invokes the resumption  $k$  under a fresh handler  $h_2$  for  $op_1$  causing all subsequent  $op_1$  operations to be handled by  $h_2$  instead.

We consider this behavior undesirable in practice as it limits the ability to do local reasoning. In particular, a programmer may not expect that calling  $op_{evil}$  changes the semantics of  $op_1$ . Yet there is no way to forbid it. Moreover, it also affects static analysis and it turns out for example that efficient evidence translation (with its subsequent performance benefits) is not possible if we allow resumptions to be this dynamic.

The solution we propose in this paper is to limit resumptions to be *scoped* only: that is, *a resumption can only be applied under the same handler context as it was captured*. The handler context is the evaluation context where we just consider the handler frames, e.g. for any evaluation context  $E$  of the form  $F_0 \cdot \text{handle } h_1 \cdot F_1 \cdot \dots \cdot \text{handle } h_n \cdot F_n$ , the handler context,  $\text{hctx}(E)$ , is  $h_1 \cdot h_2 \cdot \dots \cdot h_n$ . In particular, the evil example is rejected as it does not use a scoped resumption:  $k$  is captured under  $h_1$  but applied under  $h_2$ .

Our definition of scoped resumption is *minimal* in the sense that it is the minimal requirement needed in the proofs to maintain coherence of evidence translation. In this paper, we guarantee scoped resumptions using a dynamic runtime check in evidence translated programs (called *guard*), but it is also possible to check it statically. It is beyond the scope of this paper to give a particular design, but some ways of doing this are:

- Lexical scoping: a straightforward approach is to syntactically restrict the use of the resumption to be always in the lexical scope of the handler: i.e. fully applied within the operation clause and no occurrences under a lambda (so it cannot escape or be applied in nested handler). This can perhaps already cover all reasonable effects in practice, especially in combination with parameterized handlers<sup>2</sup>.
- A more sophisticated solution could use generative types for handler names, together with a check that those types do not escape the lexical scope as described by Zhang and Myers [2019] and also used by Biernacki et al. [2019] and Brachthäuser et al. [2020]. Another option could be to use rank-2 types to prevent the resumption from escaping the lexical scope in which the handler is defined [Leijen 2014; Peyton Jones and Launchbury 1995].

It turns out that the seminal work on algebraic effect handlers by Plotkin and Pretnar [2013] also used a similar restriction as scoped resumptions, and as such, we believe that scoped resumptions are closer to the original categorical interpretation of effect handlers. Plotkin and Pretnar use the first technique to syntactically restrict the use of resumptions under the scope of the operation clause. Resumptions variables  $k$  are in a separate syntactic class, always fully applied, and checked under a context  $K$  separate from  $\Gamma$ . However, they still allow occurrences under a lambda, allowing a resumption to escape, although in that case the evaluation would no longer type check (i.e. there is no preservation of typings under evaluation). As such it is not quite the same as scoped resumptions:

<sup>1</sup>Note that this example is fine in  $\lambda^e$  but cannot be typed in  $F^e$  as is – we discuss a properly typed version in Section 4.5.

<sup>2</sup>The lexical approach could potentially be combined with an “unsafe” resumption that uses a runtime check as done this article to cover any remaining situations.

it is both more restrictive as it needs  $k$  to occur fully applied under an operation clause; but also more liberal as it allows the separate handler state example (since  $k$  can occur under a lambda).

### 2.3 Expressiveness

Scoped resumptions bring easier-to-reason control flow, and, as we will see, open up new design space for algebraic effects compilation. However, one might worry about the expressiveness of scoped resumptions. We believe that all important effect handlers in practice can be defined in terms of scoped resumptions. In particular, note that it is still allowed for a handler to grow its context with applicative forms, for example:

```
handler { tick  $\rightarrow$   $\lambda x$   $k$ . 1 +  $k$  () } ( $\lambda$ _. perform tick (); perform tick (); 1)
```

evaluates to 3 by keeping  $(1 + \square)$  frames above the resumption. In this example, even though the full context has grown,  $k$  is still a scoped resumption as it resumes under the same (empty) handler context. Similarly, the async scheduler example that stores resumptions in a stateful queue is also accepted since each resumption still resumes under the same handler context (with the state queue handler on top). Multiple resumptions as in the backtracking example are also fine.

There are three main exceptions we know of. First, the state variant 3 based on two separate handlers does not use scoped resumptions since the *set* resumption resumes always under a handler context extended with a *get* handler. However, we can always use, and due to the reasons we have mentioned we may actually prefer, the normal state effect or the parameterized state effect. Second, shallow handlers do not resume under their own handler and as a result generally resume under a different handler context than they captured. Fortunately, any program with shallow handler can be expressed with deep handlers as well [Hillerström and Lindley 2018; Kammar et al. 2013] and thus avoid the unscoped resumptions. Finally, Kiselyov et al. [2006] show an example of code migration that resumes locally captured continuations on another host, possibly under different handlers.

## 3 EXPLICITLY TYPED EFFECT HANDLERS IN SYSTEM $F^\epsilon$

To prepare for a type directed evidence translation, we first define a typed version of the untyped calculus  $\lambda^\epsilon$  called System  $F^\epsilon$  – a call-by-value effect handler calculus extended with (higher-rank impredicative) polymorphic types and higher kinds à la System  $F_\omega$ , and row based effect types. Figure 2 defines the extended syntax and evaluation rules with the syntax of types and kinds in Figure 3. System  $F^\epsilon$  serves as an explicitly typed calculus that can be the target language of compilers and, for this article, serves as the basis for type directed evidence translation.

Being explicitly typed, we now have type applications  $e[\sigma]$  and abstractions  $\Lambda\alpha^k. v$ . Also,  $\lambda^\epsilon x : \sigma. e$ ,  $\text{handle}^\epsilon h e$ ,  $\text{handler}^\epsilon h$ , and  $\text{perform}^\epsilon op \bar{\sigma}$  all carry an effect type  $\epsilon$ . Effect types are (extensible) rows of effect labels  $l$  (like *exn* or *state*). In the types, every function arrow  $\sigma_1 \rightarrow \epsilon \sigma_2$  takes three arguments: the input type  $\sigma_1$ , the output type  $\sigma_2$ , and its effects  $\epsilon$  when it is evaluated. When  $\epsilon$  is an empty row, we often omit the effect annotation.

Since we have effect rows, effect labels, and regular value types, we use a basic kind system to keep them apart and to ensure well-formedness ( $\vdash_{wrf}$ ) of types (as defined in the Appendix).

### 3.1 Effect Rows

An effect row is either empty  $\langle \rangle$  (the *total* effect), a type variable  $\mu$  (of kind *eff*), or an extension  $\langle l \mid \epsilon \rangle$  where  $\epsilon$  is extended with effect label  $l$ . We call effects that end in an empty effect *closed*, i.e.  $\langle l_1, \dots, l_n \rangle$ ; and effects that end in a polymorphic tail *open*, i.e.  $\langle l_1, \dots, l_n \mid \mu \rangle$ . Following Leijen [2014] and Biernacki et al. [2017], we use *simple* effect rows where labels can be duplicated, and where an effect  $\langle l, l \rangle$  is not equal to  $\langle l \rangle$ . We consider rows equivalent up to the order of the labels as defined



Expressions		Values	
$e ::= v$	(value)	$v ::= x$	(variables)
$e e$	(application)	$\lambda^\epsilon x : \sigma. e$	(abstraction)
$e[\sigma]$	(type application)	$\Lambda \alpha^k. v$	(type abstraction)
$\text{handle } h e$	(handler instance)	$\text{handler}^\epsilon h$	(effect handler)
		$\text{perform}^\epsilon \text{op } \bar{\sigma}$	(operation)
Handlers		$h ::= \{ \text{op}_1 \rightarrow f_1, \dots, \text{op}_n \rightarrow f_n \}$	
Evaluation Context		$F ::= \square \mid F e \mid v F \mid F[\sigma]$	
		$E ::= \square \mid E e \mid v E \mid E[\sigma] \mid \text{handle}^\epsilon h E$	
(app)	$(\lambda^\epsilon x : \sigma. e) v$	$\longrightarrow$	$e[x := v]$
(tapp)	$(\Lambda \alpha^k. v) [\sigma]$	$\longrightarrow$	$v[\alpha := \sigma]$
(handler)	$(\text{handler}^\epsilon h) v$	$\longrightarrow$	$\text{handle}^\epsilon h \cdot v ()$
(return)	$\text{handle}^\epsilon h \cdot v$	$\longrightarrow$	$v$
(perform)	$\text{handle}^\epsilon h \cdot E \cdot \text{perform op } \bar{\sigma} v$	$\longrightarrow$	$f[\bar{\sigma}] v k \quad \text{iff } \text{op} \notin \text{bop}(E) \wedge (\text{op} \rightarrow f) \in h$ where $\text{op} : \forall \bar{\alpha}. \sigma_1 \rightarrow \sigma_2 \in \Sigma(l)$ $k = \lambda^\epsilon x : \sigma_2[\bar{\alpha} := \bar{\sigma}]. \text{handle}^\epsilon h \cdot E \cdot x$

**Fig. 2.** System  $F^\epsilon$ : explicitly typed algebraic effect handlers. Figure 3 defines the types.

Types		Kinds	
$\sigma ::= \alpha^k$	(type variables of kind $k$ )	$k ::= *$	(value type)
$c^k \sigma \dots \sigma$	(type constructor of kind $k$ )	$k \rightarrow k$	(type constructors)
$\sigma \rightarrow \epsilon \sigma$	(function type)	$\text{eff}$	(effect type $(\mu, \epsilon)$ )
$\forall \alpha^k. \sigma$	(quantified type)	$\text{lab}$	(basic effect $(l)$ )
Effect signature	$\text{sig}$	$::= \{ \text{op}_1 : \forall \bar{\alpha}_1. \sigma_1 \rightarrow \sigma'_1, \dots, \text{op}_n : \forall \bar{\alpha}_n. \sigma_n \rightarrow \sigma'_n \}$	
Effect signatures	$\Sigma$	$::= \{ l_1 : \text{sig}_1, \dots, l_n : \text{sig}_n \}$	
Type Constructors	$\langle \rangle$	:	$\text{eff}$ empty effect row (total)
	$\langle \_ \mid \_ \rangle$	:	$\text{lab} \rightarrow \text{eff} \rightarrow \text{eff}$ effect row extension
Syntax	$\langle l_1, \dots, l_n \rangle$	$\doteq$	$\langle l_1 \mid \dots \mid \langle l_n \mid \langle \rangle \rangle \dots \rangle$
	$\langle l_1, \dots, l_n \mid \mu \rangle$	$\doteq$	$\langle l_1 \mid \dots \mid \langle l_n \mid \mu \rangle \dots \rangle$
	$\epsilon ::= \sigma^{\text{eff}}, \quad \mu ::= \alpha^{\text{eff}}, \quad l ::= c^{\text{lab}}$		

**Fig. 3.** System  $F^\epsilon$ : types

$\frac{}{\epsilon \equiv \epsilon} [\text{REFL}]$	$\frac{\epsilon_1 \equiv \epsilon_2 \quad \epsilon_2 \equiv \epsilon_3}{\epsilon_1 \equiv \epsilon_3} [\text{EQ-TRANS}]$
$\frac{l_1 \neq l_2 \quad \epsilon_1 \equiv \epsilon_2}{\langle l_1, l_2 \mid \epsilon_1 \rangle \equiv \langle l_2, l_1 \mid \epsilon_2 \rangle} [\text{EQ-SWAP}]$	$\frac{\epsilon_1 \equiv \epsilon_2}{\langle l \mid \epsilon_1 \rangle \equiv \langle l \mid \epsilon_2 \rangle} [\text{EQ-HEAD}]$

**Fig. 4.** Equivalence of row-types.

in Figure 4. Note rule  $\text{EQ-SWAP}$  only swaps distinct labels. Following Leijen [2014], we disallow polymorphism over labels. There exists a complete and sound unification algorithm for these row types [Leijen 2005] and thus these are also very suitable for Hindley-Milner style type inference.

We consider using simple row-types with duplicate labels a suitable choice for a core calculus since it extends System F typing seamlessly as we only extend the notion of equality between types. There are other approaches to typing effects but all existing approaches depart from standard System F typing in significant ways. Row typing without duplicate labels leads to the introduction of type constraints, as in T-REX for example [Gaster and Jones 1996], or kinds with presence variables (Rémy style rows) as in Links for example [Hillerström and Lindley 2016; Rémy 1994]. Another approach is using effect subtyping [Bauer and Pretnar 2014] but that requires a subtype relation between types instead of simple equality.

The reason we need equivalence between row types up to order of effect labels is due to polymorphism. Suppose we have two functions that each use different effects:

$$f_1 : \forall \mu. () \rightarrow \langle l_1 \mid \mu \rangle () \quad f_2 : \forall \mu. () \rightarrow \langle l_2 \mid \mu \rangle ()$$

We would still like to be able to express  $\text{choose } f_1 f_2$  where  $\text{choose} : \forall \alpha. \alpha \rightarrow \alpha \rightarrow \alpha$ . Using row types we can type this naturally as:

$$\Lambda \mu. \text{choose}[(\rightarrow \langle l_1, l_2 \mid \mu \rangle ()) (f_1[\langle l_2 \mid \mu \rangle]) (f_2[\langle l_1 \mid \mu \rangle])]$$

where the types of the arguments are now equivalent  $\langle l_1 \mid \langle l_2 \mid \mu \rangle \rangle \equiv \langle l_2 \mid \langle l_1 \mid \mu \rangle \rangle$  (without needing subtype constraints or polymorphic label flags).

Similarly, duplicate labels can easily arise due to type instantiation. For example, a *catch* handler for exceptions can have type:

$$\text{catch} : \forall \mu \alpha. ( () \rightarrow \langle \text{exn} \mid \mu \rangle \alpha ) \rightarrow ( \text{string} \rightarrow \mu \alpha ) \rightarrow \mu \alpha$$

where *catch* takes an action that can raise exceptions, and a handler function that is called when an exception is caught. Suppose though an exception handler itself raises an exception, and has type  $h : \forall \mu. \text{string} \rightarrow \langle \text{exn} \mid \mu \rangle \text{int}$ . The application  $\text{catch action } h$  is then explicitly typed as:

$$\Lambda \mu. \text{catch}[\langle \text{exn} \mid \mu \rangle, \text{int}] \text{action } h[\mu]$$

where the type application gives rise to the type:

$$\text{catch}[\langle \text{exn} \mid \mu \rangle, \text{int}] : ( () \rightarrow \langle \text{exn}, \text{exn} \mid \mu \rangle \text{int} ) \rightarrow ( \text{string} \rightarrow \langle \text{exn} \mid \mu \rangle \text{int} ) \rightarrow \langle \text{exn} \mid \mu \rangle \text{int}$$

naturally leading to duplicate labels in the type. As we will see, simple row types also correspond naturally to the shape of the runtime evidence vectors that we introduce in Section 4.1 (where duplicated labels correspond to nested handlers).

### 3.2 Operations

We assume that every effect  $l$  has a unique set of operations  $op_1$  to  $op_n$  with a signature  $\text{sig}$  that gives every operation its input and output types,  $op_i : \forall \bar{\alpha}_i. \sigma_i \rightarrow \sigma'_i$ . There is a global map  $\Sigma$  that maps each effect  $l$  to its signature. Since we assume that each  $op$  is uniquely named, we use the notation  $op : \forall \bar{\alpha}. \sigma_1 \rightarrow \sigma_2 \in \Sigma(l)$  to denote the type of  $op$  that belongs to effect  $l$ , and also  $op \in \Sigma(l)$  to signify that  $op$  is part of effect  $l$ .

Note that we allow operations to be polymorphic. Therefore  $\text{perform } op \bar{\sigma} v$  contains the instantiation types  $\bar{\sigma}$  which are passed to the operation clause  $f$  in the evaluation rule for (*perform*) (Figure 2). This means that operations can be used polymorphically, but the handling clause itself must be polymorphic in the operation types (and use them as abstract types).

### 3.3 Quantification and Equivalence to the Untyped Dynamic Semantics

We would like the property that if we do type erasure on the newly defined System  $F^\epsilon$  we have the same semantics as with the untyped dynamic semantics, i.e.

**Theorem 1.** (System  $F^\epsilon$  has untyped dynamic semantics)

If  $e_1 \longrightarrow e_2$  in System  $F^\epsilon$ , then either  $e_1^* \longrightarrow e_2^*$  or  $e_1^* = e_2^*$ .

where  $e^*$  stands for the term  $e$  with all types, type abstractions, and type applications removed. This seems an obvious property but there is a subtle interaction with quantification. Suppose we (wrongly) allow quantification over expressions instead of values, like  $\Lambda\alpha. e$ , then consider:

$$h = \{ \text{tick} \rightarrow \lambda x:() k:() \rightarrow \langle \rangle \text{int}. 1 + k() \}$$

$$\text{handle } h((\lambda x:\forall\alpha. \text{int}. x[\text{int}] + x[\text{bool}]) (\Lambda\alpha. \text{tick} (); 1))$$

In the typed semantics, this would evaluate the argument  $x$  at each instantiation (since the whole  $\Lambda\alpha. \text{tick} (); 1$  is passed as a value), resulting in 4. On the other hand, if we do type erasure, the untyped dynamic semantics evaluates to 3 instead (evaluating the argument before applying). Not only do we lose untyped dynamic semantics, but we also break parametricity (as we can observe instantiations). So, it is quite important to only allow quantification over values, much like the ML value restriction [Kammar and Pretnar 2017; Pitts 1998; Wright 1995]. In the proof of Theorem 1 we use in particular the following (seemingly obvious) Lemma:

**Lemma 1.** (Type erasure of values)

If  $v$  is a value in System  $F^\epsilon$  then  $v^*$  is a value in  $\lambda^\epsilon$ .

Not all systems in the literature adhere to this restriction; for example Biernacki et al. [2017] and Leijen [2016] allow quantification over expressions as  $\Lambda\alpha. e$ , where both ensure soundness of the effect type system by disallowing type abstraction over effectful expressions. However, we believe that this remains a risky affair since Lemma 1 does not hold; and thus a typed evaluation may take more reduction steps than the type-erased term, i.e. seemingly shared argument values may be computed more than once.

### 3.4 Type Rules for System $F^\epsilon$

Figure 5 defines the typing rules for System  $F^\epsilon$ . The rules are of the form  $\Gamma; \mathbf{w} \vdash e : \sigma \mid \epsilon \rightsquigarrow e'$  for expressions where the variable context  $\Gamma$  and the effect  $\epsilon$  are given ( $\Uparrow$ ), and  $\sigma$  is synthesized ( $\Downarrow$ ). The gray parts define the evidence translation which we describe in Section 4 and these can be ignored for now. Values are not effectful, and are typed as  $\Gamma \vdash_{\text{val}} v : \sigma \rightsquigarrow v'$ . Since the effects are inherited, the lambda needs an effect annotation that is passed to the body derivation (ABS). In the rule APP we use standard equality between types and require that all effects match. The VAL rule goes from a value to an expression (opposite of ABS) and allows any inherited effect. The HANDLER rule takes an action with effect  $\langle l \mid \epsilon \rangle$  and handles  $l$  leaving effect  $\epsilon$ . The HANDLE rule is similar, but is defined over an expression  $e$  and types  $e$  under an extended effect  $\langle l \mid \epsilon \rangle$  in the premise. In the rule OPS, we implicitly assume  $\{op_1, \dots, op_n\} = \Sigma(l)$ .

In the Appendix, there are type rules for evaluation contexts where  $\Gamma \vdash_{\text{ec}} E : \sigma_1 \rightarrow \sigma_2 \mid \epsilon$  signifies that a context  $E$  can be typed as a function from a term of type  $\sigma_1$  to  $\sigma_2$  where the resulting expression has effect  $\epsilon$ . These rules are not needed to check programs but are very useful in proofs and theorems. In particular,

**Lemma 2.** (Evaluation context typing)

If  $\emptyset \vdash_{\text{ec}} E : \sigma_1 \rightarrow \sigma \mid \epsilon$  and  $\emptyset \vdash e : \sigma_1 \mid \langle [E]^l \mid \epsilon \rangle$ , then  $\emptyset \vdash E[e] : \sigma \mid \epsilon$ .

where  $[E]^l$  extracts all labels  $l$  from a context in reverse order:

$$[F_0 \cdot \text{handle } h_1 \cdot F_1 \dots \text{handle } h_n \cdot F_n]^l = \langle l_n, \dots, l_1 \rangle \text{ iff } h_i \in \Sigma(l_i)$$

with  $l_i$  corresponding to each  $h_i$  (for any  $op \in h_i$ ,  $op \in \Sigma(l_i)$ ). The above lemma shows we can plug well-typed expressions in a suitable context. The next lemma uses this to show the correspondence between the dynamic evaluation context and the static effect type:

$$\begin{array}{c}
\boxed{
\begin{array}{c}
\Gamma; \mathbf{w} \vdash e : \sigma \mid \epsilon \quad \epsilon \rightsquigarrow e' \quad \text{F}^{ev} \\
\Gamma \vdash_{val} v : \sigma \quad \epsilon \rightsquigarrow v' \quad \text{F}^{ev} \\
\Gamma \vdash_{ops} h : \sigma \mid l \mid \epsilon \quad \epsilon \rightsquigarrow h' \quad \text{F}^{ev}
\end{array}
} \\
\\
\frac{x : \sigma \in \Gamma}{\Gamma \vdash_{val} x : \sigma \rightsquigarrow x} [\text{VAR}] \quad \frac{\Gamma, x : \sigma_1; z \vdash e : \sigma_2 \mid \epsilon \rightsquigarrow e' \quad \text{fresh } z}{\Gamma \vdash_{val} \lambda^e x : \sigma_1. e : \sigma_1 \rightarrow^e \sigma_2 \rightsquigarrow \lambda^e z : \text{evv } \epsilon, x : [\sigma_1]. e'} [\text{ABS}] \\
\\
\frac{\Gamma \vdash_{val} v : \sigma \rightsquigarrow v'}{\Gamma; \mathbf{w} \vdash v : \sigma \mid \epsilon \rightsquigarrow v'} [\text{VAL}] \quad \frac{\Gamma \vdash_{val} v : \sigma \rightsquigarrow v' \quad k \neq \text{lab}}{\Gamma \vdash_{val} \Lambda \alpha^k. v : \forall \alpha^k. \sigma \rightsquigarrow \Lambda \alpha^k. v'} [\text{TABS}] \\
\\
\frac{\Gamma; \mathbf{w} \vdash e_1 : \sigma_1 \rightarrow^e \sigma \mid \epsilon \rightsquigarrow e'_1 \quad \Gamma; \mathbf{w} \vdash e_2 : \sigma_1 \mid \epsilon \rightsquigarrow e'_2}{\Gamma; \mathbf{w} \vdash e_1 e_2 : \sigma \mid \epsilon \rightsquigarrow e'_1 w e'_2} [\text{APP}] \quad \frac{\Gamma; \mathbf{w} \vdash e : \forall \alpha^k. \sigma_1 \mid \epsilon \rightsquigarrow e' \quad \text{twf } \sigma : k}{\Gamma; \mathbf{w} \vdash e[\sigma] : \sigma_1[\alpha := \sigma] \mid \epsilon \rightsquigarrow e'[[\sigma]]} [\text{TAPP}] \\
\\
\frac{op : \forall \bar{\alpha}. \sigma_1 \rightarrow \sigma_2 \in \Sigma(l) \quad \bar{\alpha} \notin \text{ftv}(\Gamma)}{\Gamma \vdash_{val} \text{perform}^e op \bar{\sigma} : \sigma_1[\bar{\alpha} := \bar{\sigma}] \rightarrow \langle l \mid \epsilon \rangle \sigma_2[\bar{\alpha} := \bar{\sigma}] \rightsquigarrow \text{perform}^e op \bar{\sigma}} [\text{PERFORM}] \\
\\
\frac{op_i : \forall \bar{\alpha}. \sigma_1 \rightarrow \sigma_2 \in \Sigma(l) \quad \bar{\alpha} \not\in \text{ftv}(\epsilon \sigma) \quad \Gamma \vdash_{val} f_i : \forall \bar{\alpha}. \sigma_1 \rightarrow^e ((\sigma_2 \rightarrow^e \epsilon \sigma) \rightarrow^e \epsilon \sigma) \rightsquigarrow f'_i}{\Gamma \vdash_{ops} \{ op_1 \rightarrow f_1, \dots, op_n \rightarrow f_n \} : \sigma \mid l \mid \epsilon \rightsquigarrow \{ op_i \rightarrow f'_i \}} [\text{OPS}] \\
\\
\frac{\Gamma \vdash_{ops} h : \sigma \mid l \mid \epsilon \rightsquigarrow h'}{\Gamma \vdash_{val} \text{handler}^e h : () \rightarrow \langle l \mid \epsilon \rangle \sigma \rightarrow^e \epsilon \sigma \rightsquigarrow \text{handler}^e h'} [\text{HANDLER}] \\
\\
\frac{\Gamma \vdash_{ops} h : \sigma \mid l \mid \epsilon \rightsquigarrow h' \quad \Gamma; \langle l : (m, h') \mid \mathbf{w} \rangle \vdash e : \sigma \mid \langle l \mid \epsilon \rangle \rightsquigarrow e' \quad m \text{ fresh}}{\Gamma; \mathbf{w} \vdash \text{handle}^e h e : \sigma \mid \epsilon \rightsquigarrow \text{handle}_m^w h' e'} [\text{HANDLE}]
\end{array}$$

**Fig. 5.** Type Rules for System  $F^e$  combined with type directed evidence translation to  $F^{ev}$  (in gray.)

**Lemma 3.** (Effect corresponds to the evaluation context)

If  $\emptyset \vdash E[e] : \sigma \mid \epsilon$  then there exists  $\sigma_1$  such that  $\emptyset \vdash_{ec} E : \sigma_1 \rightarrow \sigma \mid \epsilon$ , and  $\emptyset \vdash e : \sigma_1 \mid \langle [E]^l \mid \epsilon \rangle$ .

Here we see that the rules guarantee that exactly the effects  $[E]^l$  in  $e$  are handled by the context  $E$ .

### 3.5 Progress and Preservation

We establish two essential lemmas about the meaning of effect types. First, in any well-typed total System  $F^e$  expression, all operations are handled (and thus, evaluation cannot get stuck):

**Lemma 4.** (Well typed operations are handled)

If  $\emptyset \vdash E[\text{perform } op \bar{\sigma} v] : \sigma \mid \langle \rangle$  then  $E$  has the form  $E_1 \cdot \text{handle}^e h \cdot E_2$  with  $op \notin \text{bop}(E_2)$  and  $op \rightarrow f \in h$ .

Moreover, effect types are meaningful in the sense that an effect type fully reflects all possible effects that may happen during evaluation:

**Lemma 5.** (Effects types are meaningful)

If  $\emptyset \vdash E[\text{perform } op \bar{\sigma} v] : \sigma \mid \epsilon$  with  $op \notin \text{bop}(E)$ , then  $op \in \Sigma(l)$  and  $l \in \epsilon$ , i.e. effect types cannot be discarded without a handler.

Expressions		Values	
$e ::= v$	(value)	$v ::= x$	(variables)
$e[\sigma]$	(type application)	$\lambda^\epsilon z : \text{evv } \epsilon, x : \sigma. e$	(evidence abstraction)
$e \ w \ e$	(evidence application)	$\Lambda \alpha^k. v$	(type abstraction)
$\text{handle}_m^w h \ e$	(handler instance)	$\text{handler}^\epsilon h$	(effect handler)
		$\text{perform}^\epsilon \text{ op } \bar{\sigma}$	(operation)
		$\text{guard}^w E \ \sigma$	(guarded abstraction)
Type Constructors			
marker		: $\text{eff} \rightarrow * \rightarrow *$	handler instance marker ( $m$ )
evv		: $\text{eff} \rightarrow *$	evidence vector ( $w, z$ )
ev		: $\text{lab} \rightarrow *$	evidence ( $ev$ )
Evidence Syntax			
$m$		: $\text{marker} \in \sigma$	handler instance marker
$(m, h)$		: $\text{ev } l$	evidence
$\langle \rangle$		: $\text{evv } \langle \rangle$	empty evidence vector
$\langle l_1 : \text{ev}_1, \dots, l_n : \text{ev}_n \rangle$		: $\text{evv } \langle l_1, \dots, l_n \rangle$	evidence vector, with every $l_i \leq l_{i+1}$
$(app)$	$(\lambda^\epsilon z : \text{evv } \epsilon, x : \sigma. e) \ w \ v$	$\longrightarrow$	$e[z := w, x := v]$
$(tapp)$	$(\Lambda \alpha^k. v) [\sigma]$	$\longrightarrow$	$v[\alpha := \sigma]$
$(handler)$	$(\text{handler}^\epsilon h) \ w \ v$	$\longrightarrow$	$\text{handle}_m^w h \ (v \ \langle l : (m, h) \mid w \rangle ())$ where $m$ is unique and $h \in \Sigma(l)$
$(return)$	$\text{handle}_m^w h \cdot v$	$\longrightarrow$	$v$
$(perform)$	$\text{handle}_m^w h \cdot E \cdot \text{perform}^\epsilon \text{ op } \bar{\sigma} \ w' \ v$	$\longrightarrow$	$f[\bar{\sigma}] \ w \ v \ w \ k$ iff $\text{op} \notin \text{bop}(E) \wedge (\text{op} \rightarrow f) \in h$ where $\text{op} : \forall \bar{\alpha}. \sigma_1 \rightarrow \sigma_2 \in \Sigma(l)$ $k = \text{guard}^w (\text{handle}_m^w h \cdot E) (\sigma_2[\bar{\alpha} := \bar{\sigma}])$
$(guard)$	$(\text{guard}^w E \ \sigma) \ w \ v$	$\longrightarrow$	$E[v]$

Fig. 6. System  $F^{ev}$  Typed operational semantics with evidence

Using these Lemmas, we can show that evaluation can always make progress and that the typings are preserved during evaluation.

**Theorem 2. (Progress)**

If  $\emptyset \vdash e_1 : \sigma \mid \langle \rangle$  then either  $e_1$  is a value, or  $e_1 \mapsto e_2$ .

**Theorem 3. (Preservation)**

If  $\emptyset \vdash e_1 : \sigma \mid \langle \rangle$  and  $e_1 \mapsto e_2$ , then  $\emptyset \vdash e_2 : \sigma \mid \langle \rangle$ .

#### 4 POLYMORPHIC EVIDENCE TRANSLATION TO SYSTEM $F^{ev}$

Having established a sound explicitly typed core calculus, we can now proceed to do evidence translation. The goal of evidence translation is to pass down evidence  $ev$  of the handlers in the evaluation context to place where operations are performed. This will in turn enable other optimizations (as described in Section 6) since we can now locally inspect the evidence instead of searching in the dynamic evaluation context.

Following Brachthäuser and Schuster [2017], we represent evidence  $ev$  for an effect  $l$  as a pair  $(m, h)$ , consisting of a unique *marker*  $m$  and its corresponding handler implementation  $h$ . The markers uniquely identify each handler frame in the context which is now marked as  $\text{handle}_m h$ . The

reason for introducing the separate handler  $h$  construct is now apparent: it *instantiates*  $\text{handle}_m h$  frames with a unique  $m$ . This representation of evidence allows for two important optimizations: (1) We can change the operational rule for `perform` to directly yield to a particular handler identified by  $m$  (instead of needing to search for the innermost one), shown in Section 5.1, and (2) It allows local inspection of the actual handler  $h$  so we can evaluate tail resumptive operations in place, shown in Section 6.

However, passing the evidence down to each operation turns out to be surprisingly tricky to get right and we took quite a few detours before arriving at the current solution. At first, we thought we could represent evidence for each label  $l$  in the effect of a function as separate argument  $ev\ l$ . For example,

$$f_1 : \forall \mu. \text{int} \rightarrow \langle l_1 \mid \mu \rangle \text{int} = \Lambda \mu. \lambda x. \text{perform } op_1\ x$$

would be translated as:

$$f_1 : \forall \mu. ev\ l_1 \rightarrow \text{int} \rightarrow \langle l_1 \mid \mu \rangle \text{int} = \Lambda \mu. \lambda ev. \lambda x. \text{perform } op_1\ ev\ x$$

This does not work though as type instantiation can now cause the runtime representation to change as well! For example, if we instantiate  $\mu$  to  $\langle l_2 \rangle$  as  $f_1[\langle l_2 \rangle]$  the type becomes  $\text{int} \rightarrow \langle l_1, l_2 \rangle \text{int}$  which now takes *two* evidence parameters. Even worse, such instantiation can be inside arbitrary types, like a list of such functions, where we cannot construct evidence transformers in general.

Another design that does not quite work is to regard evidence translation as an instance of qualified types [Jones 1992] and use a dictionary passing translation. In essence, in the theory of qualified types, the qualifiers are scoped over monomorphic types, which does not fit well with effect handlers. Suppose we have a function `foo` with a qualified evidence types as:

$$foo : Ev\ l_1 \Rightarrow (\text{int} \rightarrow \langle l_1 \rangle \text{int}) \rightarrow \langle l_1 \rangle \text{int}$$

Note that even though `foo` is itself qualified, the argument it takes is a plain function  $\text{int} \rightarrow \langle l_1 \rangle \text{int}$  and *has already resolved its own qualifiers*. That is too eager for our purposes. For example, if we apply `foo` ( $f_1[\langle \rangle]$ ), under dictionary translation we would get `foo`  $ev_1$  ( $f_1[\langle \rangle]$   $ev_1$ ). However, it may well be that `foo` itself applies  $f_1$  under a new handler for the  $l_1$  effect and thus needs to pass different evidence than  $ev_1$ ! Effectively, dictionary translation may partially apply functions with their dictionaries which is not legal for handler evidence. The qualified type we really require for `foo` uses higher-ranked qualifiers, something like  $Ev\ l_1 \Rightarrow (Ev\ l_1 \Rightarrow \text{int} \rightarrow \langle l_1 \rangle \text{int}) \rightarrow \langle l_1 \rangle \text{int}$ .

#### 4.1 Evidence Vectors

The design we present here instead passes all evidence as a single *evidence vector* to each (effectful) function: this keeps the runtime representations stable under type instantiation, and we can ensure syntactically that functions are never partially applied to evidence.

Figure 6 defines our target language  $F^{ev}$  as an explicitly typed calculus with evidence passing. All applications are now of the form  $e_1\ w\ e_2$  where we always pass an evidence vector  $w$  with the original argument  $e_2$ . Therefore, all lambdas are of the form  $\lambda^e z : evv\ \epsilon, x : \sigma. e$  and always take an evidence vector  $z$  besides their regular parameter  $x$ . Also,  $\text{handle}_m$  frames now carry the evidences and become  $\text{handle}_m^w$ . We also extend application forms in the evaluation context to take evidence parameters. The double arrow notation is used to denote the type of these “tupled” lambdas:

$$\sigma_1 \Rightarrow \epsilon\ \sigma_2 \doteq evv\ \epsilon \rightarrow \sigma_1 \rightarrow \epsilon\ \sigma_2$$

During evidence translation, every effect type  $\epsilon$  on an arrow is translated to an explicit runtime evidence vector of type  $evv\ \epsilon$ , and we translate type annotations as:

$$\begin{aligned} [\cdot] : k &\rightarrow k \\ [\forall \alpha. \sigma] &= \forall \alpha. [\sigma] & [\alpha] &= \alpha \\ [\tau_1 \rightarrow \epsilon\ \tau_2] &= [\tau_1] \Rightarrow \epsilon\ [\tau_2] & [c\ \tau_1 \dots \tau_n] &= c\ [\tau_1] \dots [\tau_n] \end{aligned}$$



Evidence vectors  $\langle l_1 : ev_1, \dots, l_n : ev_n \rangle$  are essentially a map from effect labels to evidence. Later we want to be able to select evidence from a vector with a constant offset instead of searching for the label, so we are going to keep them in a canonical form ordered by the effect types  $l$ . That is, we have every  $l_i \leq l_{i+1}$ . An empty evidence vector is denoted as  $\langle \rangle$ .

During evaluation we need to be able to select evidence from an evidence vector, and to insert new evidence when a handler is instantiated, and we define the following two operations:

$$\begin{aligned} \_ . l & : \forall \mu. \text{evv } \langle l \mid \mu \rangle \rightarrow \text{ev } l & (\text{select evidence from a vector}) \\ \langle l : \_ \mid \_ \rangle & : \forall \mu. \text{ev } l \rightarrow \text{evv } \mu \rightarrow \text{evv } \langle l \mid \mu \rangle & (\text{evidence insertion}) \end{aligned}$$

Where we assume the following two laws that relate selection and insertion:

$$\begin{aligned} \langle l : \text{ev} \mid w \rangle . l &= \text{ev} \\ \langle l' : \text{ev} \mid w \rangle . l &= w . l \quad \text{iff } l \neq l' \end{aligned}$$

Operation  $\langle l : \text{ev} \mid w \rangle$  inserts an evidence  $ev$  into the evidence vector  $w$  in an ordered way:

$$\begin{aligned} \langle l : \_ \mid \_ \rangle & : \forall \mu. \text{ev } l \rightarrow \text{evv } \mu \rightarrow \text{evv } \langle l \mid \mu \rangle \\ \langle l : \text{ev} \mid \langle \rangle \rangle &= \langle l : \text{ev} \rangle \\ \langle l : \text{ev} \mid \langle l' : \text{ev}', w \rangle \rangle &= \langle l' : \text{ev}', \langle l : \text{ev} \mid w \rangle \rangle \quad \text{iff } l > l' \\ \langle l : \text{ev} \mid \langle l' : \text{ev}', w \rangle \rangle &= \langle l : \text{ev}, l' : \text{ev}', w \rangle \quad \text{iff } l \leq l' \end{aligned}$$

Note how the dynamic representation as vectors of labeled evidence nicely corresponds to the static effect row-types, in particular with regard to duplicate labels, which correspond to nested handlers at runtime. Here we see why we cannot swap the position of equal effect labels as we need the evidence to correspond to their actual order in the evaluation context. Inserting all evidence in a vector  $w_1$  into another vector  $w_2$  is defined inductively as following. We reuse the same notation of inserting a single evidence.

$$\begin{aligned} \langle \langle \rangle \mid w_2 \rangle &= w_2 \\ \langle \langle l : \text{ev}, w_1 \rangle \mid w_2 \rangle &= \langle l : \text{ev} \mid \langle w_1 \mid w_2 \rangle \rangle \end{aligned}$$

and evidence selection can be defined as:

$$\begin{aligned} \_ . l & : \forall \mu. \text{evv } \langle l \mid \mu \rangle \rightarrow \text{ev } l \\ \langle l : \text{ev}, \_ \rangle . l &= \text{ev} \\ \langle l' : \text{ev}, w \rangle . l &= w . l \quad \text{iff } l \neq l' \\ \langle \rangle . l &= (\text{cannot happen}) \end{aligned}$$

## 4.2 Evidence Translation

The evidence translation is already defined in Figure 5, in the gray parts of the rules. The full rules for expressions are of the form  $\Gamma; w \vdash e : \sigma \mid \epsilon \rightsquigarrow e'$  where given a context  $\Gamma$ , the expression  $e$  has type  $\sigma$  with effect  $\epsilon$ . The rules take the current evidence vector  $w$  for the effect  $\epsilon$ , of type  $\text{evv } \epsilon$ , and translate to an expression  $e'$  of System  $F^{ev}$ .

The translation in itself is straightforward where we only need to ensure extra evidence is passed during applications and abstracted again on lambdas. The  $\text{ABS}$  rule abstracts fully over all evidence in a function as  $\lambda^{\epsilon} z : \text{evv } \epsilon, x : \sigma_1. e'$ , where the evidence vector is abstracted as  $z$  and passed to its premise. Note that since we are translating,  $z$  is not part of  $\Gamma$  here (which scopes over  $F^{\epsilon}$  terms). The type rules for  $F^{ev}$ , discussed below, do track such variables in the context. The dual of this is rule  $\text{APP}$  which passes the effect evidence  $w$  as an extra argument to every application as  $e'_1 w e'_2$ .

To prove preservation and coherence of the translation, we also include a translation rule for handle, even though we assume these are internal. Otherwise there are no surprises here and the main difficulty lies in the operational rules, which we discuss in detail in Section 4.4.

To prove additional properties about the translated programs, we define a more restricted set of typing rules directly over System  $F^{ev}$  in Figure 9 of the form  $\Gamma; w \Vdash e : \sigma \mid \epsilon$  (ignoring the gray parts), such that  $\Gamma \vdash w : \text{evv } \epsilon$ . Using this, we prove that the translation is sound:

**Theorem 4.** (*Evidence translation is Sound in  $F^{ev}$* )

If  $\emptyset; \langle \rangle \vdash e : \sigma \mid \langle \rangle \rightsquigarrow e' : [\sigma] \mid \langle \rangle$ .

**4.3 Correspondence**

The evidence translation maintains a strong correspondence between the effect types, the evidence vectors, and the evaluation contexts. To make this precise, we define the (reverse) extraction of all handlers in a context  $E$  as  $[E]$  where:

$$[F_1 \cdot \text{handle}_{m_1}^{w_1} h_1 \cdot \dots \cdot F_n \cdot \text{handle}_{m_n}^{w_n} h_n \cdot F] = \langle l_n : (m_n, h_n) \mid \dots \mid l_1 : (m_1, h_1) \rangle \text{ iff } h_i \in \Sigma(l_i)$$

$$[F_1 \cdot \text{handle}_{m_1}^{w_1} h_1 \cdot \dots \cdot F_n \cdot \text{handle}_{m_n}^{w_n} h_n \cdot F]^l = \langle l_n, \dots, l_1 \rangle$$

$$[F_1 \cdot \text{handle}_{m_1}^{w_1} h_1 \cdot \dots \cdot F_n \cdot \text{handle}_{m_n}^{w_n} h_n \cdot F]^m = \{m_n, \dots, m_1\}$$

With this we can characterize the correspondence between the evaluation context and the evidence used at perform:

**Lemma 6.** (*Evidence corresponds to the evaluation context*)

If  $\emptyset; w \Vdash E[e] : \sigma \mid \epsilon$  then for some  $\sigma_1$  we have  $\emptyset; \langle [E] \mid w \rangle \Vdash e : \sigma_1 \mid \langle [E]^l \mid \epsilon \rangle$ ,

and  $\emptyset; w \Vdash E : \sigma_1 \rightarrow \sigma \mid \epsilon$ .

**Lemma 7.** (*Well typed operations are handled*)

If  $\emptyset; \langle \rangle \Vdash E[\text{perform } op \ \bar{\sigma} \ v] : \sigma \mid \langle \rangle$  then  $E$  has the form  $E_1 \cdot \text{handle}_m^w h \cdot E_2$  with  $op \notin \text{bop}(E_2)$  and  $op \rightarrow f \in h$ .

These brings us to our main theorem which states that the evidence passed to an operation corresponds exactly to the innermost handler for that operation in the dynamic evaluation context:

**Theorem 5.** (*Evidence Correspondence*)

If  $\emptyset; \langle \rangle \Vdash E[\text{perform } op \ \bar{\sigma} \ w \ v] : \sigma \mid \langle \rangle$  then  $E$  has the form  $E_1 \cdot \text{handle}_m^{w'} h \cdot E_2$  with  $op \in \Sigma(l)$ ,  $op \notin \text{bop}(E_2)$ ,  $op \rightarrow f \in h$ , and the evidence corresponds exactly to dynamic execution context such that  $w.l = (m, h)$ .

**4.4 Operational Rules of System  $F^{ev}$** 

The operational rules for System  $F^{ev}$  are defined in Figure 6. Since every application now always takes an evidence vector argument  $w$  the new (*app*) and (*handler*) rules now only reduce when both arguments are present (and the syntax does not allow for partial evidence applications).

The (*handler*) rule differs from System  $F^e$  in two significant ways. First, it saves the current evidence in scope (passed as  $w$ ) in the handle frame itself as  $\text{handle}_m^w$ . Secondly, the evidence vector it passes on to its action is now extended with its own unique evidence, as  $\langle l : (m, h) \mid w \rangle$ .

In the (*perform*) rule, the operation clause ( $op \rightarrow f$ )  $\in h$  is now translated itself, and we need to pass evidence to  $f$ . Since it takes two arguments, the operation payload  $x$  and its resumption  $k$ , the application becomes  $(f[\bar{\sigma}] \ w \ x) \ w \ k$ . The evidence we pass to  $f$  is the evidence of the *original handler context* saved as  $\text{handle}_m^w$  in the (*handler*) rule. In particular, we should not pass the evidence  $w'$  of the operation, as that is the evidence vector of the context in which the operation itself evaluates (and an extension of  $w$ ). In contrast, we evaluate each clause under their original context and need the evidence vector corresponding to that. In fact, we can even ignore the evidence vector  $w'$  completely for now as we only need to use it later for implementing optimizations.

**4.5 Guarded Context Instantiation and Scoped Resumptions**

The definition of the resumption  $k$  in the (*perform*) rule differs quite a bit from the original definition in System  $F^e$  (Figure 2), which was:

$$k = \lambda^e x : \sigma_2[\bar{\alpha} := \bar{\sigma}]. \text{handle}^e h \cdot E \cdot x$$

while the  $F^{ev}$  definition now uses:

$$k = \text{guard}^w (\text{handle}_m^w h \cdot E) (\sigma_2[\bar{\alpha} := \bar{\sigma}])$$

where we use a the new  $F^{ev}$  value term  $\text{guard}^w E \sigma$ . Since  $k$  has a regular function type, it now needs to take an extra evidence vector, and we may have expected a more straightforward extension without needing a new guard rule, something like:

$$k = \lambda^\epsilon z : \text{evv } \epsilon, x : \sigma_2[\bar{\alpha} := \bar{\sigma}]. \text{handle}^\epsilon h \cdot E \cdot x$$

but then the question becomes what to do with that passed in evidence  $z$ ? This is the point where it becomes more clear that resumptions are special and not quite like a regular lambda since they restore a captured context. In particular, *the context  $E$  that is restored has already captured the evidence of the original context in which it was captured (as  $w$ ), and thus may not match the evidence of the context in which it is resumed (as  $z$ )!*

The new guarded application rule makes this explicit and only restores contexts that are resumed under the exact same evidence, in other words, only scoped resumptions are allowed:

$$(\text{guard}^w E \sigma) w v \longrightarrow E[v]$$

If the evidence does not match, the evaluation is stuck in  $F^{ev}$ .

As an example of how this can happen, we return to our *evil* example in Section 2.2 which uses non-scoped resumptions to change the meaning of  $op_1$ . Since we are now in a typed setting, we modify the example to return a data type of results to make everything well-typed:

$$\begin{aligned} \text{data } res &= \text{again} : () \rightarrow \langle one \rangle res \rightarrow res \\ &\quad | \text{done} : int \rightarrow res \\ \Sigma &= \{ one : \{ op_1 : () \rightarrow int \}, evil : \{ op_{evil} : () \rightarrow () \} \} \end{aligned}$$

with the following helper definitions:

$$\begin{aligned} h_1 &= \{ op_1 \rightarrow \lambda x k. k \ 1 \} & f(\text{again } k) &= \text{handler } h_2 (\lambda \_ . k \ ()); \ 0 \\ h_2 &= \{ op_1 \rightarrow \lambda x k. k \ 2 \} & f(\text{done } x) &= x \\ h_{evil} &= \{ op_{evil} \rightarrow \lambda x k. (\text{again } k) \} \end{aligned}$$

$$body = \text{perform } op_1 \ (); \text{perform } op_{evil} \ (); \text{perform } op_1 \ (); \text{done } 0$$

and where the main expression is evidence translated as:

$$\begin{aligned} &f(\text{handler } h_1 (\lambda \_ . \text{handler } h_{evil} (\lambda \_ . body))) \\ &\rightsquigarrow f \langle \rangle (\text{handler } h_1 \langle \rangle (\lambda z \_ . \text{handler } h_{evil} z \\ &\quad (\lambda z : \text{evv } \langle one, evil \rangle, \_ . \text{perform } op_1 \ z \ (); \text{perform } op_{evil} \ z \ (); \text{perform } op_1 \ z \ (); \text{done } 0))) \end{aligned}$$

Starting evaluation in the translated expression, we can now derive:

$$\begin{aligned} &\mapsto^* f \langle \rangle \cdot \text{handle}_{m_1}^{\langle \rangle} h_1 \cdot \text{handle}_{m_2}^{w_1} h_{evil} \cdot (\square; \text{perform } op_1 \ w_2 \ (); \text{done } 0) \cdot \text{perform } op_{evil} \ w_2 \ () \\ &\quad \text{with } w_1 = \langle one : (m_1, h_1) \rangle, w_2 = \langle evil : (m_2, h_{evil}), one : (m_1, h_1) \rangle \\ &\mapsto^* f \langle \rangle \cdot \text{handle}_{m_1}^{\langle \rangle} h_1 \cdot (\lambda z x z k. (\text{again } k)) w_1 \ () w_1 \ k \\ &\quad \text{with } k = \text{guard}^{w_1} (\text{handle}_{m_2}^{w_1} h_{evil} \cdot (\square; \text{perform } op_1 \ w_2 \ (); \text{done } 0)) \\ &\mapsto^* f \langle \rangle \cdot \text{handle}_{m_1}^{\langle \rangle} h_1 \cdot (\text{again } k) \\ &\mapsto^* f \langle \rangle (\text{again } k) \\ &\mapsto \text{handler } h_2 \langle \rangle (\lambda z \_ . k \ z \ ()) \\ &\mapsto^* \text{handle}_{m_3}^{\langle \rangle} h_2 \cdot k \ w_3 \ () \quad \text{with } w_3 = \langle one : (m_3, h_2) \rangle \\ &= \text{handle}_{m_3}^{\langle \rangle} h_2 \cdot \text{guard}^{w_1} (\text{handle}_{m_2}^{w_1} h_{evil} \cdot (\square; \text{perform } op_1 \ w_2 \ (); \text{done } 0)) w_3 \ () \end{aligned}$$

At this point, the guard rule gets stuck as we have captured the context originally under evidence  $w_1$ , but we try to resume with evidence  $w_3$ , and  $w_1 = \langle one : (m_1, h_1) \rangle \neq \langle one : (m_3, h_2) \rangle = w_3$ .

If we allow the guarded context instantiation anyways we get into trouble when we try to perform  $op_1$  again:

$$\begin{aligned} &\mapsto \text{handle}_{m_3}^{\langle \rangle} h_2 \cdot \text{handle}_{m_2}^{w_1} h_{evil} \cdot (\langle \rangle ; \text{perform } op_1 \ w_2 \ (); \text{ done } 0) \\ &\mapsto^* \text{handle}_{m_3}^{\langle \rangle} h_2 \cdot \text{handle}_{m_2}^{w_1} h_{evil} \cdot (\square; \text{done } 0) \cdot \text{perform } op_1 \ w_2 \ () \end{aligned}$$

in that case the innermost handler for  $op_1$  is now  $h_2$  while the evidence  $w_2.l$  is  $(m_1, h_1)$  and it no longer corresponds to the dynamic context! (and that would void our main correspondence Theorem 5 and in turn invalidate optimizations based on this).

#### 4.6 Uniqueness of Handlers

It turns out that to guarantee coherence of the translation to plain polymorphic lambda calculus, as discussed in Section 5, we need to ensure that all  $m$ 's in an evaluation context are always unique. This is a tricky property; for example, uniqueness of markers does *not* hold for arbitrary  $F^{ev}$  expressions: markers may be duplicated inside lambdas outside of the evaluation context, and we can also construct an expression manually with duplicated markers, e.g.  $\text{handle}_m^w \cdot \text{handle}_m^w \cdot e$ . However, we can prove that if we only consider initial  $F^{ev}$  expressions without  $\text{handle}_m^w$ , or any expressions reduced from that during evaluation, then it is guaranteed that all  $m$ 's are always unique in the evaluation context – even though the (*handler*) rule introduces  $\text{handle}_m^w$  during evaluation, and the (*app*) rule may duplicate markers.

##### Definition 1. (Handle-safe expressions)

A *handle-safe*  $F^{ev}$  expression is a well-typed closed expression that either (1) contains no  $\text{handle}_m^w$  term; or (2) is itself reduced from a handle-safe expression.

##### Theorem 6. (Uniqueness of handlers)

For any handle-safe  $F^{ev}$  expression  $e$ , if  $e = E_1 \cdot \text{handle}_{m_1}^{w_1} h \cdot E_2 \cdot \text{handle}_{m_2}^{w_2} h \cdot e_0$ , then  $m_1 \neq m_2$ .

#### 4.7 Preservation and Coherence

As exemplified above, the guard rule is also essential to prove the preservation of evidence typings under evaluation. In particular, we can show:

##### Theorem 7. (Preservation of evidence typing)

If  $\emptyset; \langle \rangle \Vdash e_1 : \sigma \mid \langle \rangle$  and  $e_1 \mapsto e_2$ , then  $\emptyset; \langle \rangle \Vdash e_2 : \sigma \mid \langle \rangle$ .

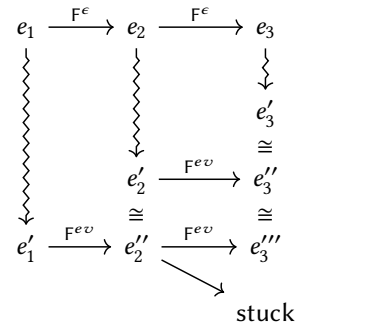


Fig. 7. Coherence

Even more important though is to show that our translation is *coherent*, that is, if we take an evaluation step in System  $F^e$ , the evidence translated expression will take a similar step such that the resulting expression is again a translation of the reduced  $F^e$  expression:

##### Theorem 8. (Evidence translation is coherent)

If  $\emptyset; \langle \rangle \vdash e_1 : \sigma \mid \langle \rangle \rightsquigarrow e'_1$  and  $e_1 \mapsto e_2$ , and (due to preservation)  $\emptyset; \langle \rangle \vdash e_2 : \sigma \mid \langle \rangle \rightsquigarrow e'_2$ , then there exists a  $e'_2''$ , such that  $e'_1 \mapsto e'_2''$  and  $e'_2'' \cong e'_2$ .

Interestingly, the theorem states that the translated  $e'_2$  is only coherent under an equivalence relation  $\cong$  to the reduced expression  $e'_2''$ , as illustrated in Figure 7. The reason that  $e'_2$

and  $e'_2''$  are not directly equal is due to guard expressions only being generated by reduction. In particular, if we have a  $F^e$  reduction of the form:

$$\text{handle}^e h \cdot E \cdot \text{perform } op \ \bar{\sigma} \ v \longrightarrow f \ \bar{\sigma} \ v \ k \quad \text{with } k = \lambda^e x : \sigma'. \text{handle}^e h \cdot E \cdot x$$

Expressions	$e ::= v \mid e \ e \mid e[\sigma]$	Context	$F ::= \square \mid F \ e \mid v \ F \mid F[\sigma]$
Values	$v ::= x \mid \lambda x : \sigma. e \mid \Lambda \alpha^k. v$	$E ::= F$	
(app)	$(\lambda^\epsilon x : \sigma. e) \ v \longrightarrow e[x := v]$		
(tapp)	$(\Lambda \alpha^k. v) [\sigma] \longrightarrow v[\alpha := \sigma]$		
<hr/>			
$\frac{x : \sigma \in \Gamma}{\Gamma \vdash_F x^\sigma : \sigma}$	$\frac{\Gamma \vdash_F v : \sigma}{\Gamma \vdash_F \Lambda \alpha^k. v : \forall \alpha^k. \sigma}$	$\frac{\Gamma, x : \sigma_1 \vdash_F e : \sigma_2}{\Gamma \vdash_F \lambda x : \sigma_1. e : \sigma_1 \rightarrow \sigma_2}$	$\frac{\Gamma \vdash_F v : \sigma}{\Gamma \vdash_F v^\sigma : \sigma}$
<hr/>			
$\frac{\Gamma \vdash_F e_1 : \sigma_1 \rightarrow \sigma \quad \Gamma \vdash_F e_2 : \sigma}{\Gamma \vdash_F e_1 \ e_2 : \sigma}$	$\frac{\Gamma \vdash_F e : \forall \alpha^k. \sigma_1 \quad \vdash_{wf} \sigma : k}{\Gamma \vdash_F e[\sigma] : \sigma_1[\alpha := \sigma]}$		

**Fig. 8.** System  $F^v$ : explicitly typed (higher kinded) polymorphic lambda calculus with strict evaluation. Types as in Figure 3 with no effects on the arrows.

then the translation takes the following  $F^{ev}$  reduction:

$$\text{handle}_m^w h \cdot E' \cdot \text{perform } op \ [\bar{\sigma}] \ w' \ v' \longrightarrow f' \ [\bar{\sigma}] \ w \ v' \ w \ k' \text{ with } k' = \text{guard}^w (\text{handle}_m^w h' \cdot E') \ \sigma'$$

At this point the translation of  $f \ \bar{\sigma} \ v \ k$  will be of the form  $f' \ [\bar{\sigma}] \ w' \ v' \ w' \ k''$  where

$$k'' = \lambda^\epsilon z : \text{evv } \epsilon, x. \text{handle}^\epsilon h' \cdot E'' \cdot x$$

i.e. the resumption  $k$  is translated as a regular lambda now and not as guard! Also, since  $E$  is translated now under a lambda, the resulting  $E''$  differs in all evidence terms  $w$  in  $E'$  which will be  $z$  instead.

However, we know that if the resumption  $k'$  is ever applied, the argument is either exactly  $w$ , in which case  $E''[z := w] = E'$ , or not equal to  $w$  in which case the evidence translated program gets stuck. This is captured by  $\cong$  relation which is the smallest transitive and reflexive congruence among well-typed  $F^{ev}$  expressions, up to renaming of unique markers, satisfying the EQ-GUARD rule, which captures the notion of guarded context instantiation.

$$\frac{e[z := w] \cong E[x]}{\lambda^\epsilon z, x : \sigma. e \cong \text{guard}^w E \ \sigma} \text{ [EQ-GUARD]}$$

Now, is this definition of equivalence strong enough? Yes, because we can show that if two translated expressions are equivalent, then they stay equivalent under reduction (or get stuck):

**Lemma 8.** (Operational semantics preserves equivalence, or gets stuck)

If  $e_1 \cong e_2$ , and  $e_1 \longrightarrow e'_1$ , then either  $e_2$  is stuck, or we have  $e'_2$  such that  $e_2 \longrightarrow e'_2$  and  $e'_1 \cong e'_2$ .

This establishes the full coherence of our evidence translation: if a translated expression reduces under  $F^{ev}$  without getting stuck, the final value is equivalent to the value reduced under System  $F^\epsilon$ . Moreover, the only way an evidence translated expression can get stuck is if it uses non-scoped resumptions.

Note that the evidence translation never produces guard terms, so the translated expression can always take an evaluation step; however, subsequent evaluation steps may lead to guard terms, so after the first step, it may get stuck if a resumption is applied under a different handler context than where it was captured.

## 5 TRANSLATION TO CALL-BY-VALUE POLYMORPHIC LAMBDA CALCULUS

**Fig. 9.** Monadic translation to System- $F^v$ . ( $((\triangleright))$  is monadic bind).

There has been previous work that performs such translation [Forster et al. 2019; Hillerström et al. 2017; Leijen 2017c], as well as various libraries that embed effect handlers as monads [Kammar et al. 2013; Wu et al. 2014] but without evidence translation such embeddings require either a



sophisticated runtime system [Dolan et al. 2017 2015; Leijen 2017a], or are not quite as efficient as one might hope. The translation presented here allows for better optimization as it maintains evidence and has no special runtime requirements (it is just F!).

## 5.1 Translating to Multi-Prompt Delimited Continuations

As a first step, we show that we do not need explicit handle frames anymore that carry around the handler operations  $h$ , but can translate to multi-prompt delimited continuations [Brachthäuser and Schuster 2017]. Gunter, Rémy, and Riecke [1995] present the set and cupto operators for named prompts  $m$  with the following “control-upto” rule:

$$\text{set } m \text{ in } \cdot E \cdot \text{cupto } m \text{ as } k \text{ in } e \longrightarrow (\lambda k. e) (\lambda x. E \cdot x) \quad m \notin [E]^m$$

This effectively exposes “shallow” multi-prompts: for our purposes, we always need “deep” handling where the resumption evaluates under the same prompt again and we define:

$$\begin{aligned} \text{prompt}_m e &\doteq \text{set } m \text{ in } e \\ \text{yield}_m f &\doteq \text{cupto } m \text{ as } k \text{ in } (f (\lambda x. \text{set } m \text{ in } (k x))) \end{aligned}$$

which gives us the following derived evaluation rule:

$$\text{prompt}_m \cdot E \cdot \text{yield}_m f \longrightarrow f (\lambda x. \text{prompt}_m \cdot E \cdot x) \quad m \notin [E]^m$$

This is almost what we need, except that we need a multi-prompt that also takes evidence  $w$  into account and uses guard instead of a plain lambda to apply the resumption, i.e.,

$$\text{prompt}_m^w \cdot E \cdot \text{yield}_m f \longrightarrow f \ w (\text{guard}^w (\text{prompt}_m^w \cdot E)) \quad m \notin [E]^m$$

Therefore, we take prompt and yield as primitive control operators with the above evaluation rule.

Using the correspondence property (Theorem 5), we can use the evidence to inspect the handler locally at the perform and no longer need to keep it in the handle frame. We can now translate both perform  $op \ v \ w$  and handle $_m^w h$  in terms of the simpler yield $_m$  and prompt $_m^w$ , as:

$$\begin{aligned} [\text{handle}_m^w h] &= \text{prompt}_m^w \\ [\text{perform } op \ w' \ v] &= \text{yield}_m (\lambda w \ k. f \ w \ v \ w \ k) \quad \text{with } (m, h) = w'.l \text{ and } (op \rightarrow f) \in h \end{aligned}$$

We prove that this is a sound interpretation of effect handling:

**Theorem 9.** (Evidence Translation to Multi-Prompt Delimited Continuations is Sound)

For any evaluation step  $e_1 \mapsto e_2$  in  $F^{ev}$ , we have  $[e_1] \mapsto^* [e_2]$  with multi-prompt delimited continuations.

Dolan et al. [2015] describe the multi-core OCaml runtime system with *split stacks*; in such setting we could use the pointers to a split point as markers  $m$ , and directly yield to the correct handler with constant time capture of the context. Even with such runtime, it may still be beneficial to do a monadic translation as well in order to be able to use standard compiler optimizations.

## 5.2 Monadic Multi-Prompt Translation to System $F^v$

With the relation to multi-prompt delimited control established, we can now translate  $F^{ev}$  to  $F^v$  in a monadic style, where we use standard techniques [Dybvig et al. 2007] to implement the delimited control as a monad. Assuming notation for data types and matching, we can define a multi-prompt monad mon as follows:

$$\begin{aligned} \text{data mon } \mu \ \alpha &= \\ | \text{pure} : \alpha &\rightarrow \text{mon } \mu \ \alpha \\ | \text{yield} : \forall \beta \ r \ \mu'. &\text{marker } \mu' \ r \rightarrow (\text{evv } \mu' \rightarrow (\text{evv } \mu' \rightarrow \beta \rightarrow \text{mon } \mu' \ r) \rightarrow \text{mon } \mu' \ r) \\ &\rightarrow (\text{mon } \mu \ \beta \rightarrow \text{mon } \mu \ \alpha) \rightarrow \text{mon } \mu \ \alpha \end{aligned}$$

$$\begin{aligned} \text{pure } x &= \text{pure } x \\ \text{yield } m \text{ clause} &= \text{yield } m \text{ clause id} \end{aligned}$$

The pure case is used for value results, while the *yield* implements yielding to a prompt. A *yield*  $m f cont$  has three arguments, (1) the marker  $m$  : marker  $\mu' r$  bound to a prompt in some context with effect  $\mu'$  and *answer type*  $r$ ; (2) the operation clause which receives the resumption (of type  $\beta \rightarrow \text{mon } \mu' r$ ) where  $\beta$  is the type of the operation result; and finally (3) the current continuation  $cont$  which is the runtime representation of the context. When binding a *yield*, the continuation keeps being extended until the full context is captured:

$$\begin{aligned} (f \circ g) x &= f (g x) && \text{(function composition)} \\ (f \bullet g) x &= g x \triangleright f && \text{(Kleisli composition)} \\ (\text{pure } x) \triangleright g &= g x && \text{(monadic bind)} \\ (\text{yield } m f cont) \triangleright g &= \text{yield } m f (g \bullet cont) \end{aligned}$$

The hoisting of *yields* corresponds closely to operation hoisting as described by Bauer and Pretnar [2015]. The *prompt* operation has three cases to consider:

$$\begin{aligned} \text{prompt} &: \forall \mu \alpha. \text{marker } \mu \alpha \rightarrow \text{evv } \mu \rightarrow \text{mon } \langle l \mid \mu \rangle \alpha \rightarrow \text{mon } \mu \alpha \\ \text{prompt } m w (\text{pure } x) &= \text{pure } x \\ \text{prompt } m w (\text{yield } m' f cont) &= \text{yield } m' f (\text{prompt } m w \circ cont) \quad \text{if } m \neq m' \\ \text{prompt } m w (\text{yield } m f cont) &= f w (\text{guard } w (\text{prompt } m w \circ cont)) \end{aligned}$$

In the pure case, we are at the (*value*) rule and return the result as is. If we find a *yield* that yields to another prompt we also keep yielding but remember to restore our prompt when resuming in its current continuation, as (*prompt*  $m w \circ cont$ ). The final case is when we *yield* to the prompt itself, in that case we are in the (*yield*) transition and continue with  $f$  passing the context evidence  $w$  and a guarded resumption<sup>3</sup>.

The *guard* operation simply checks if the evidence matches and either continues or gets stuck:

$$\text{guard } w_1 cont w_2 x = \text{if } (w_1 == w_2) \text{ then } cont (\text{pure } x) \text{ else stuck}$$

Note that due to the uniqueness property (Theorem 6) we can check the equality  $w_1 == w_2$  efficiently by only comparing the markers  $m$  (and ignoring the handlers). The *handle* and *perform* can be translated directly into *prompt* and *yield* as shown in the previous section, where we generate a *handler*<sup>*l*</sup> definition per effect  $l$ , and a *perform*<sup>*op*</sup> for every operation:

$$\begin{aligned} \text{handler}^l &: \forall \mu \alpha. \text{hnd}^l \mu \alpha \rightarrow \text{evv } \mu \rightarrow (\text{evv } \langle l \mid \mu \rangle \rightarrow ()) \rightarrow \text{mon } \langle l \mid \mu \rangle \alpha \rightarrow \text{mon } \mu \alpha \\ \text{perform}^{op} &: \forall \mu \bar{\alpha}. \text{evv } \langle l \mid \mu \rangle \rightarrow \sigma_1 \rightarrow \text{mon } \langle l \mid \mu \rangle \sigma_2 \quad \text{with } op : \forall \bar{\alpha}. \sigma_1 \rightarrow \sigma_2 \in \Sigma(l) \\ \text{handler}^l h w f &= \text{freshm } (\lambda m \rightarrow \text{prompt } m w (f \langle l : (m, h) \mid w \rangle ())) \\ \text{perform}^{op} w x &= \text{let } (m, h) = w.l \text{ in yield } m (\lambda w k. ((h.op) w x \triangleright (\lambda f. f w k))) \end{aligned}$$

The *handler* creates a fresh marker and passes on new evidence under a new *prompt*. The *perform* can now directly select the evidence  $(m, h)$  from the passed evidence vector and *yield* to  $m$  directly. The function passed to *yield* is a bit complex since each operation clause is translated normally and has a nested monadic type, so we need to bind the first partial application to  $x$  before passing the continuation  $k$ .

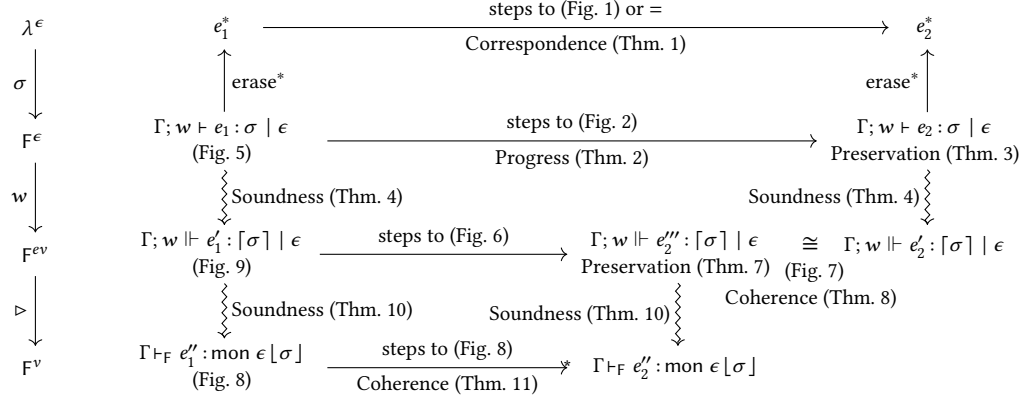
Finally, for every effect signature  $l : \text{sig} \in \Sigma$  we declare a corresponding data type  $\text{hnd}^l \epsilon r$  that is a record of operation clauses:

$$\begin{aligned} l : \{ op_1 : \forall \bar{\alpha}_1. \sigma_1 \rightarrow \sigma'_1, \dots, op_n : \forall \bar{\alpha}_n. \sigma_n \rightarrow \sigma'_n \} \\ \rightsquigarrow \text{data hnd}^l \mu r = \text{hnd}^l \{ op_1 : \forall \bar{\alpha}_1. op \sigma_1 \sigma'_1 \mu r, \dots, op_n : \forall \bar{\alpha}_n. op \sigma_n \sigma'_n \mu r \} \end{aligned}$$

where operations *op* are a type alias defined as:

$$\text{alias } op \alpha \beta \mu r \doteq \text{evv } \mu \rightarrow \alpha \rightarrow \text{mon } \mu (\text{evv } \mu \rightarrow (\text{evv } \mu \rightarrow \beta \rightarrow \text{mon } \mu r) \rightarrow \text{mon } \mu r)$$

<sup>3</sup>Typing the third case needs a dependent match on the markers  $m' : \text{marker } \mu' r$  and  $m = \text{marker } \mu \alpha$  where their equality implies  $\mu = \mu'$  and  $r = \alpha$ . This can be done in Haskell with the *Equal* GADT, or encoded in  $F^\vee$  using explicit equality witnesses [Baars and Swierstra 2002].



**Fig. 10.** An overview how the various theorems establish the relations between the different calculi

With these definitions in place, we can do a straightforward type directed translation from  $F^{ev}$  to  $F^v$  by just lifting all operations into the prompt monad, as shown in Figure 9. Types are translated by making all effectful functions monadic:

$$\begin{aligned} \llbracket \forall \bar{\alpha}. \sigma \rrbracket &= \forall \bar{\alpha}. \llbracket \sigma \rrbracket & \llbracket \sigma_1 \Rightarrow \epsilon \sigma_2 \rrbracket &= \text{evv } \epsilon \rightarrow \llbracket \sigma_1 \rrbracket \rightarrow \text{mon } \epsilon \llbracket \sigma_2 \rrbracket \\ \llbracket \alpha \rrbracket &= \alpha & \llbracket c \sigma_1 \dots \sigma_n \rrbracket &= c \llbracket \sigma_1 \rrbracket \dots \llbracket \sigma_n \rrbracket \end{aligned}$$

We prove that these definitions are correct, and that the resulting translation is fully coherent, where a monadic program evaluates to the same result as a direct evaluation in  $F^{ev}$ .

**Theorem 10. (Monadic Translation is Sound)**

If  $\emptyset; \langle \rangle; \langle \rangle \Vdash e : \sigma \mid \langle \rangle \rightsquigarrow e'$ , then  $\emptyset \vdash_F e' : \text{mon } \langle \rangle \llbracket \sigma \rrbracket$ .

**Theorem 11. (Coherence of the Monadic Translation)**

If  $\emptyset; \langle \rangle; \langle \rangle \Vdash e_1 : \sigma \mid \langle \rangle \rightsquigarrow e'_1$  and  $e_1 \longrightarrow e_2$ , then also  $\emptyset; \langle \rangle; \langle \rangle \Vdash e_2 : \sigma \mid \langle \rangle \rightsquigarrow e'_2$  where  $e'_1 \longrightarrow^* e'_2$ .

Together with earlier results we establish full soundness and coherence from the original typed effect handler calculus  $F^e$  to the evidence based monadic translation into plain call-by-value polymorphic lambda calculus  $F^v$ . See Figure 10 for how our theorems relate these systems to each other.

## 6 OPTIMIZATIONS

With a fully coherent evidence translation to plain polymorphic lambda calculus in hand, we can now apply various transformations in that setting to optimize the resulting programs.

### 6.1 Partially Applied Handlers

In the current *perform<sup>op</sup>* implementation, we yield with a function that takes evidence  $w$  to pass on to the operation clause  $f$ , as:

$$\lambda w k. ((h.op) w x \triangleright (\lambda f. f w k))$$

However, the  $w$  that is going to be passed in is always that of the  $\text{handle}_m^w$  frame, as explained in Section 4.4. When we instantiate the  $\text{handle}_m^w$  we can in principle map the  $w$  in advance over all operation clause so these can be partially evaluated over the evidence vector:

$$\text{handler}^l h w f = \text{freshm} (\lambda m \rightarrow \text{prompt } m w (f \ll l : (m, \text{pmap}^l w h \mid w) \gg ()))$$

$$\begin{aligned} \text{pmap}^l w (\text{hnd}^l f_1 \dots f_n) &= \text{phnd}^l (\text{partial } w f_1) \dots (\text{partial } w f_n) \\ \text{partial} &:: \text{evv } \mu \rightarrow \text{op } \alpha \beta \mu r \rightarrow \text{pop } \alpha \beta \mu r \\ \text{partial } w f &= \lambda x k. (f w x \triangleright (\lambda f'. f' w k)) \end{aligned}$$

The  $\text{pmap}^l$  function creates a new handler data structure  $\text{phnd}^l$  where every operation is now partially applied to the evidence which results in simplified type for each operation (as expressed by the  $\text{pop}$  type alias):

$$\text{alias pop } \alpha \beta \mu r \doteq \alpha \rightarrow (\text{evv } \mu \rightarrow \beta \rightarrow \text{mon } \mu r) \rightarrow \text{mon } \mu r$$

The  $\text{perform}$  is now simplified as well as it no longer needs to bind the intermediate application:

$$\text{perform}^{\text{op}} w x = \text{let } (m, h) = w.l \text{ in } \text{yield } m (\lambda k. (h.op) x k)$$

Finally, the  $\text{prompt}$  case where the marker matches no longer needs to pass evidence as well:

$$\dots$$

$$\text{prompt } m w (\text{yield } m f \text{ cont}) = f (\text{guard } w (\text{prompt } m w \circ \text{cont}))$$

By itself, the impact of this optimization will be modest, just allowing inlining of evidence in  $f$  clauses, but it opens up the way to do tail resumptive operations in-place.

## 6.2 Evaluating Tail Resumptive Operations In Place

In practice, almost all important effects are tail-resumptive. The main exceptions we know of are asynchronous I/O (but that is dominated by I/O anyways) and the ambiguity effect for resuming multiple times. As such, we expect the vast majority of operations to be tail-resumptive, and being able to optimize them well is important. We can extend the partially evaluated handler approach to optimize tail resumptions as well. First we extend the  $\text{pop}$  type to be a data type that signifies if an operation clause is tail resumptive or not:

$$\begin{aligned} \text{data pop } \alpha \beta \mu r &= \text{tail} : (\alpha \rightarrow \text{mon } \mu \beta) \rightarrow \text{pop } \alpha \beta \mu r \\ &\mid \text{normal} : (\alpha \rightarrow (\beta \rightarrow \text{mon } \mu r) \rightarrow \text{mon } \mu r) \rightarrow \text{pop } \alpha \beta \mu r \end{aligned}$$

The  $\text{partial}$  function now creates tail terms for any clause  $f$  that the compiler determined to be tail resumptive (i.e. of the form  $\lambda x k. k e$  with  $k \notin \text{fv}(e)$ ):

$$\begin{aligned} \text{partial } w f &= \text{tail } (\lambda x. (f w x \triangleright (\lambda f'. f' w \text{pure}))) \quad \text{if } f \text{ is tail resumptive} \\ \text{partial } w f &= \text{normal } (\lambda x k. (f w x \triangleright (\lambda f'. f' w k))) \quad \text{otherwise} \end{aligned}$$

Note that even if  $f$  is tail resumptive, it may still use  $x$ . Moreover, since  $f$  has already captured its evidence  $w$ , it can still perform operations itself.

Instead of passing in an “real” resumption function  $k$ , we just pass  $\text{pure}$  directly, leading to  $\lambda x. (e \triangleright \text{pure})$  – and such clause we can now evaluate in-place without needing to yield and capture our resumption context explicitly. The  $\text{perform}^{\text{op}}$  can directly inspect the form of the operation clause from its evidence, and evaluate in place when possible:

$$\begin{aligned} \text{perform}^{\text{op}} w x &= \text{let } (m, h) = w.l \text{ in } \text{case } h.op \text{ of } \mid \text{tail } f \rightarrow f x \\ &\mid \text{normal } f \rightarrow \text{yield } m (f x) \end{aligned}$$

Moreover, if a known handler is applied over some expression, regular optimizations like inlining and known-case evaluation, can often inline the operations fully. As everything has been translated to regular functions and regular data types without any special evaluation rules, there is no need for special optimization rules for handlers either.

### 6.3 Using Constant Offsets in Evidence Vectors

The  $\text{perform}^{op}$  operation is now almost as efficient as a virtual method call for tail resumptive operations (just check if it is tail and do in indirect call), except that it still needs to do a dynamic lookup for the evidence as  $w.l$ .

The trick here is to take advantage of the canonical order of the evidence in a vector, where the location of the evidence in a vector of a closed effect type is fully determined. In particular, for any evidence vector  $w$  of type  $\text{evv } \langle l \mid \epsilon \rangle$  where  $\epsilon$  is closed, we can replace  $w.l$  by a direct index  $w[\text{ofs}]$  where  $(l \text{ in } \epsilon) = \text{ofs}$ , defined as:

$$\begin{aligned} l \text{ in } \langle \rangle &= 0 \\ l \text{ in } \langle l' \mid \epsilon \rangle &= l \text{ in } \epsilon \quad \text{iff } l \leq l' \\ l \text{ in } \langle l' \mid \epsilon \rangle &= 1 + (l \text{ in } \epsilon) \quad \text{iff } l > l' \end{aligned}$$

This means for any functions with a closed effect, the offset of all evidence is constant. Only functions that are polymorphic in the effect tail need to index dynamically. It is beyond the scope of this paper to discuss this in detail but we believe that even in those cases we can index by a direct offset: following the same approach as TREX [Gaster and Jones 1996], we can use qualified types internally to propagate  $(l \text{ in } \mu)$  constraints where the “dictionary” is simply the offset in the evidence vector (and these constraints can be hidden from the user as we can always solve them).

### 6.4 Reducing Continuation Allocation

The monadic translation still produces inefficiencies as it captures the continuation at every point where an operation may yield. For example, when calling an effectful function  $\text{foo}$ , as in  $x \leftarrow \text{foo } (); e$ , the monadic translation produces a bind which takes an allocated lambda as a second argument to represent the continuation  $e$  explicitly, as  $\text{foo } () \triangleright (\lambda x. e)$ .

First of all, we can do a *selective* monadic translation [Leijen 2017c] where we leave out the binds if the effect of a function can be guaranteed to never produce a yield, e.g. total functions (like arithmetic), all effects provided by the host platform (like I/O), and all effects that are statically guaranteed to be tail resumptive (called *linear* effects). It turns out that many (leaf) functions satisfy this property so this removes the vast majority of binding.

Secondly, since we expect the vast majority of operations to be tail resumptive, almost always the effectful functions will not yield at all. It therefore pays off to always inline the bind operation and perform a direct match on the result and inline the continuation directly, e.g., we can expand  $x \leftarrow \text{foo } (); e$  to:

$$\begin{aligned} \text{case } \text{foo } () \text{ of } & \mid \text{yield } m f \text{ cont} \rightarrow \text{yield } m f ((\lambda x. e) \bullet \text{cont}) \\ & \mid \text{pure } x \rightarrow e \end{aligned}$$

This can be done very efficiently, and is close to what a C or Go programmer would write: returning a (yielding) flag from every function and checking the flag before continuing. Of course, this is also a dangerous optimization as it duplicates the expression  $e$ , and more research is needed to evaluate the impact of code duplication and finding a good inlining heuristic.

As a closing remark, the above optimization is why we prefer the monadic approach over continuation passing style (CPS). With CPS, our example would pass the continuation directly as  $\text{foo } () (\lambda x. e)$ . This style may be more efficient if one often yields (as the continuation is more efficiently composed versus bubbling up through the binds [Ploeg and Kiselyov 2014]) but it prohibits our optimization where we can inspect the result of  $\text{foo}$  (without knowing its implementation) and to only allocate a continuation if it is really needed.

## 6.5 Implementation

We have an initial implementation of the evidence passing translation in the Koka language [Leijen 2019] using the JavaScript backend<sup>4</sup>. The original runtime implementation uses a combination of CPS translation [Leijen 2017c] and an internal shadow stack of handlers where the operations propagate through this stack. The runtime part is about 1000 lines of JavaScript. The new implementation based on evidence passing translation requires just a few primitives though (about 100 lines of JavaScript to handle evidence vectors efficiently).

While a systematic evaluation of efficient implementation strategies using evidence translation is beyond the scope of this paper, the initial benchmark results look promising. In particular, using benchmarks from [Kiselyov et al. 2013] and [Kammar et al. 2013], we compiled the same source with the original compiler (*runtime*) and with the new evidence translating compiler (*evidence*), and compare against the *direct* implementation of the benchmark that uses no handlers. We summarize the benchmark results in the following table. The table presents the relative speed of each implementation compared to the *runtime* compiler (where we adjusted the *evidence* relative to any difference between the *direct* versions to correct further for any other differences between the compiler versions).

	runtime	evidence	direct	description
counter	1.00×	1.94×	2.14×	A counter in a loop.
count-mod5	1.00×	1.28×	0.83×	Fold over a list and increment a counter on every 5th element.
layered	1.00×	2.23×	2.34×	Use a state handler above five other reader effect handlers.
nqueens	1.00×	46.09×	76.20×	The n-queens problem.

In *counter*, *evidence* is almost twice as fast as *runtime*, executing the tail resumptive increment in place. It is also getting close to *direct*. In *count-mod5*, the improvement is more modest. Note that *direct* is slower here due to the need to propagate the counter explicitly as an extra argument. The *layered* benchmark can impact performance if searching linearly for a handler. This time *evidence* is more than twice as fast, and again close to *direct*. Finally, on *nqueens*, *evidence* is much faster and about two thirds the speed of *direct*. Since evidence translation *as such* does not speed up the capturing and restoring of backtracking resumptions as used in *nqueens*, this result is a bit surprising. We conjecture that the explicit representation of continuations and evidence also helps the JavaScript compiler to optimize well, while the internal shadow stack handling in the *runtime* implementation may prohibit such optimizations here.

## 7 RELATED WORK

In the paper, we compare to related work mostly inline when we discuss various aspects. In what follows, we briefly discuss more related work.

The work by Forster et al. [2019] is close to our work as it shows how delimited control, monads, and effect handlers can express each other. They show in particular a monadic semantics for effect handlers, but also prove that there does not exist a typed translation in their monomorphic setting. They conjecture a polymorphic translation may exist, and this paper proves that such translation is indeed possible.

Recent work by Biernacki et al. [2019] introduces labeled effect handlers where a handler can be referred to by name; the generative semantics with global labels is similar to our runtime markers *m*, but these labels are not guaranteed to be unique in the evaluation context (and they use the

<sup>4</sup>In the dev-ev branch in the Koka repository [Leijen 2019].



innermost handler in such case). Similar to this work they also distinguish between the generative handler (as  $\text{handle}_a$ ), and the expression form handler ( $\text{handle}_m$  (as  $\text{handle}_l$ )).

Brachthäuser et al. [2020] use capability passing to perform operations directly on a specific handler [Brachthäuser and Schuster 2017; Brachthäuser et al. 2018]. This is also similar to the work of Zhang and Myers [2019] where handlers are passed by name as well. Both of these approaches can be viewed as programming with explicit evidence (capabilities) and we can imagine extending our calculus to allow a programmer to refer explicitly to the evidence (name) of a handler.

## 8 CONCLUSION

We have shown a full formal and coherent translation from a polymorphic core calculus for effect handlers ( $F^\epsilon$ ) to a polymorphic lambda calculus ( $F^\nu$ ) based on evidence translation (through  $F^{\epsilon\nu}$ ), and we have characterized the relation to multi-prompt delimited continuations precisely. Besides giving a new framework to reason about semantics of effect handlers, we are also hopeful that these techniques will be used to create efficient implementations of effect handlers in practice. Moreover, from a language design perspective, we expect that the restriction to scoped resumptions will be more widely adopted. As future work, we would like to further implement these techniques and integrate them into the Koka language, with systematic evaluations of efficient implementation strategies using evidence translation.

## REFERENCES

- Arthur I. Baars, and S. Doaitse Swierstra. 2002. Typing Dynamic Typing. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*, 157–166. Pittsburgh, PA, USA. doi:[10.1145/581478.581494](https://doi.org/10.1145/581478.581494).
- Andrej Bauer, and Matija Pretnar. 2014. An Effect System for Algebraic Effects and Handlers. *Logical Methods in Computer Science* 10 (4).
- Andrej Bauer, and Matija Pretnar. 2015. Programming with Algebraic Effects and Handlers. *J. Log. Algebr. Meth. Program.* 84 (1): 108–123. doi:[10.1016/j.jlamp.2014.02.001](https://doi.org/10.1016/j.jlamp.2014.02.001).
- Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. Dec. 2017. Handle with Care: Relational Interpretation of Algebraic Effects and Handlers. *Proc. ACM Program. Lang.* 2 (POPL'17 issue): 8:1–8:30. doi:[10.1145/3158096](https://doi.org/10.1145/3158096).
- Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. Dec. 2019. Binders by Day, Labels by Night: Effect Instances via Lexically Scoped Handlers. *Proc. ACM Program. Lang.* 4 (POPL). Association for Computing Machinery, New York, NY, USA. doi:[10.1145/3371116](https://doi.org/10.1145/3371116).
- Jonathan Immanuel Brachthäuser, and Philipp Schuster. Oct. 2017. Effekt: Extensible Algebraic Effects in Scala. In *Scala'17*. Vancouver, CA.
- Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. Oct. 2018. Effect Handlers for the Masses. *Proc. ACM Program. Lang.* 2 (OOPSLA). Association for Computing Machinery, New York, NY, USA. doi:[10.1145/3276481](https://doi.org/10.1145/3276481).
- Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. 2020. Effekt: Capability-Passing Style for Type- and Effect-Safe, Extensible Effect Handlers in Scala. *Journal of Functional Programming*. Cambridge University Press.
- Stephen Dolan, Spiros Eliopoulos, Daniel Hillerström, Anil Madhavapeddy, KC Sivaramakrishnan, and Leo White. May 2017. Concurrent System Programming with Effect Handlers. In *Proceedings of the Symposium on Trends in Functional Programming*. TFP'17.
- Stephen Dolan, Leo White, KC Sivaramakrishnan, Jeremy Yallop, and Anil Madhavapeddy. Sep. 2015. Effective Concurrency through Algebraic Effects. In *OCaml Workshop*.
- R Kent Dyvbig, Simon Peyton Jones, and Amr Sabry. 2007. A Monadic Framework for Delimited Continuations. *Journal of Functional Programming* 17 (6). Cambridge University Press: 687–730. doi:[10.1017/S0956796807006259](https://doi.org/10.1017/S0956796807006259).
- Yannick Forster, Ohad Kammar, Sam Lindley, and Matija Pretnar. 2019. On the Expressive Power of User-Defined Effects: Effect Handlers, Monadic Reflection, Delimited Control. *Journal of Functional Programming* 29. Cambridge University Press: 15. doi:[10.1017/S0956796819000121](https://doi.org/10.1017/S0956796819000121).
- Ben R. Gaster, and Mark P. Jones. 1996. *A Polymorphic Type System for Extensible Records and Variants*. NOTTCS-TR-96-3. University of Nottingham.
- Carl A. Gunter, Didier Rémy, and Jon G. Riecke. 1995. A Generalization of Exceptions and Control in ML-like Languages. In *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture*, 12–23. FPCA '95. ACM, La Jolla, California, USA. doi:[10.1145/224164.224173](https://doi.org/10.1145/224164.224173).

- Daniel Hillerström, and Sam Lindley. 2016. Liberating Effects with Rows and Handlers. In *Proceedings of the 1st International Workshop on Type-Driven Development*, 15–27. TyDe 2016. Nara, Japan. doi:[10.1145/2976022.2976033](https://doi.org/10.1145/2976022.2976033).
- Daniel Hillerström, and Sam Lindley. 2018. Shallow Effect Handlers. In *APLAS'18*.
- Daniel Hillerström, Sam Lindley, Bob Atkey, and KC Sivaramakrishnan. Sep. 2017. Continuation Passing Style for Effect Handlers. In *Proceedings of the Second International Conference on Formal Structures for Computation and Deduction*. FSCD'17.
- Mark P. Jones. Feb. 1992. A Theory of Qualified Types. In *4th. European Symposium on Programming (ESOP'92)*, 582:287–306. Lecture Notes in Computer Science. Springer-Verlag, Rennes, France. doi:[10.1007/3-540-55253-7\\_17](https://doi.org/10.1007/3-540-55253-7_17).
- Ohad Kammar, Sam Lindley, and Nicolas Oury. 2013. Handlers in Action. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, 145–158. ICFP '13. ACM, New York, NY, USA. doi:[10.1145/2500365.2500590](https://doi.org/10.1145/2500365.2500590).
- Ohad Kammar, and Matija Pretnar. Jan. 2017. No Value Restriction Is Needed for Algebraic Effects and Handlers. *Journal of Functional Programming* 27 (1). Cambridge University Press. doi:[10.1017/S0956796816000320](https://doi.org/10.1017/S0956796816000320).
- Oleg Kiselyov, Amr Sabry, and Cameron Swords. 2013. Extensible Effects: An Alternative to Monad Transformers. In *Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell*, 59–70. Haskell '13. Boston, Massachusetts, USA. doi:[10.1145/2503778.2503791](https://doi.org/10.1145/2503778.2503791).
- Oleg Kiselyov, and Chung-chieh Shan. 2009. Embedded Probabilistic Programming. In *Domain-Specific Languages*. doi:[10.1007/978-3-642-03034-5\\_17](https://doi.org/10.1007/978-3-642-03034-5_17).
- Oleg Kiselyov, Chung-chieh Shan, and Amr Sabry. 2006. Delimited Dynamic Binding. In *Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming*, 26–37. ICFP '06. Association for Computing Machinery, New York, NY, USA. doi:[10.1145/1159803.1159808](https://doi.org/10.1145/1159803.1159808).
- Daan Leijen. 2005. Extensible Records with Scoped Labels. In *Proceedings of the 2005 Symposium on Trends in Functional Programming*, 297–312.
- Daan Leijen. 2014. Koka: Programming with Row Polymorphic Effect Types. In *MSFP'14, 5th Workshop on Mathematically Structured Functional Programming*. doi:[10.4204/EPTCS.153.8](https://doi.org/10.4204/EPTCS.153.8).
- Daan Leijen. Aug. 2016. *Algebraic Effects for Functional Programming*. MSR-TR-2016-29. Microsoft Research technical report. Extended version of [Leijen 2017c].
- Daan Leijen. 2017a. Implementing Algebraic Effects in C: Or Monads for Free in C. Edited by Bor-Yuh Evan Chang. *Programming Languages and Systems*, LNCS, 10695 (1). Springer International Publishing, Suzhou, China: 339–363. APLAS'17, Suzhou, China.
- Daan Leijen. 2017b. Structured Asynchrony with Algebraic Effects. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Type-Driven Development*, 16–29. TyDe 2017. Oxford, UK. doi:[10.1145/3122975.3122977](https://doi.org/10.1145/3122975.3122977).
- Daan Leijen. Jan. 2017. Type Directed Compilation of Row-Typed Algebraic Effects. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL'17)*, 486–499. Paris, France. doi:[10.1145/3009837.3009872](https://doi.org/10.1145/3009837.3009872).
- Daan Leijen. 2019. Koka Repository. <https://github.com/koka-lang/koka>.
- Sam Lindley, and James Cheney. 2012. 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. doi:[10.1145/2103786.2103798](https://doi.org/10.1145/2103786.2103798).
- Sam Lindley, Connor McBride, and Craig McLaughlin. Jan. 2017. Do Be Do Be Do. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL'17)*, 500–514. Paris, France. doi:[10.1145/3009837.3009897](https://doi.org/10.1145/3009837.3009897).
- Simon L Peyton Jones, and John Launchbury. 1995. State in Haskell. *Lisp and Symbolic Comp.* 8 (4): 293–341. doi:[10.1007/BF01018827](https://doi.org/10.1007/BF01018827).
- Andrew M. Pitts. 1998. Existential Types: Logical Relations and Operational Equivalence. In *Proceedings of the 25th International Colloquium on Automata, Languages and Programming*, 309–326. Springer-Verlag.
- Atze van der Ploeg, and Oleg Kiselyov. 2014. Reflection without Remorse: Revealing a Hidden Sequence to Speed up Monadic Reflection. In *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell*, 133–144. Haskell'14. Gothenburg, Sweden. doi:[10.1145/2633357.2633360](https://doi.org/10.1145/2633357.2633360).
- Gordon D. Plotkin, and John Power. 2003. Algebraic Operations and Generic Effects. *Applied Categorical Structures* 11 (1): 69–94. doi:[10.1023/A:1023064908962](https://doi.org/10.1023/A:1023064908962).
- Gordon D. Plotkin, and Matija Pretnar. 2013. Handling Algebraic Effects. In *Logical Methods in Computer Science*, volume 9. 4. doi:[10.2168/LMCS-9\(4:23\)2013](https://doi.org/10.2168/LMCS-9(4:23)2013).
- Matija Pretnar. Dec. 2015. An Introduction to Algebraic Effects and Handlers. Invited Tutorial Paper. *Electron. Notes Theor. Comput. Sci.* 319 (C). Elsevier Science Publishers: 19–35. doi:[10.1016/j.entcs.2015.12.003](https://doi.org/10.1016/j.entcs.2015.12.003).
- Matija Pretnar, Amr Hany Shehata Saleh, Axel Faes, and Tom Schrijvers. 2017. *Efficient Compilation of Algebraic Effects and Handlers*. CW Reports. Department of Computer Science, KU Leuven; Leuven, Belgium. <https://lirias.kuleuven.be/retrieve/472230>.
- Didier Rémy. 1994. Type Inference for Records in Natural Extension of ML. In *Theoretical Aspects of Object-Oriented Programming*, 67–95. doi:[10.1.1.48.5873](https://doi.org/10.1.1.48.5873).

- Philipp Schuster, Jonathan Immanuel Brachthäuser, and Klaus Ostermann. 2019. *Staging Effects and Handlers – A Shift to Zero-Cost Effect Handlers*. Technical Report. University of Tübingen, Germany. <http://ps.informatik.uni-tuebingen.de/publications/schuster19zero>.
- Sebastian Ullrich, and Leonardo de Moura. Sep. 2019. Counting Immutable Beans – Reference Counting Optimized for Purely Functional Programming. In *Proceedings of the 31st Symposium on Implementation and Application of Functional Languages (IFL'19)*. Singapore.
- Andrew Wright. 1995. Simple Imperative Polymorphism. In *LISP and Symbolic Computation*, 343–356.
- Nicolas Wu, Tom Schrijvers, and Ralf Hinze. 2014. Effect Handlers in Scope. In *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell*, 1–12. Haskell '14. Gothenburg, Sweden. doi:[10.1145/2633357.2633358](https://doi.org/10.1145/2633357.2633358).
- Yizhou Zhang, and Andrew C. Myers. Jan. 2019. Abstraction-Safe Effect Handlers via Tunneling. *Proc. ACM Program. Lang.* 3 (POPL). ACM. doi:[10.1145/3290318](https://doi.org/10.1145/3290318).