Generalized Evidence Passing for Effect Handlers

Efficient Compilation of Effect Handlers to C

NINGNING XIE, University of Hong Kong, China
DAAN LEIJEN, Microsoft Research, USA

This paper studies compilation techniques for algebraic effect handlers. In particular, we present a sequence of refinements of algebraic effects, going via multi-prompt delimited control, *generalized evidence passing*, yield bubbling, and finally a monadic translation into plain lambda calculus which can be compiled efficiently to many target platforms. Along the way we explore various interesting points in the design space. We provide two implementations of our techniques, one as a library in Haskell, and one as a C backend for the Koka programming language. We show that our techniques are effective, by comparing against three other best-in-class implementations of effect handlers: multi-core OCaml, the *Ev.Eff* Haskell library, and the libhandler C library. We hope this work can serve as a basis for future designs and implementations of algebraic effects.

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

Additional Key Words and Phrases: Algebraic Effects, Handlers, Evidence Passing

ACM Reference Format:
Ningning Xie and Daan Leijen. 2021. Generalized Evidence Passing for Effect Handlers: Efficient Compilation of Effect Handlers to C. *Proc. ACM Program. Lang.* 5, ICFP, Article 71 (August 2021), 30 pages. https://doi.org/10.1145/3473576

1 INTRODUCTION

Algebraic effects and handlers [Plotkin and Power 2003; Plotkin and Pretnar 2013] provide a powerful and flexible way to add structured control-flow abstraction to programming languages. Unfortunately, it is not straightforward to compile effect handlers into efficient code: effect operations are generally able to capture- and resume a delimited continuation, which usually requires special runtime support to do efficiently. For example, the effect handler implementation in multi-core OCaml [Dolan et al. 2017; Sivaramakrishnan et al. 2021] relies on a runtime system that uses segmented stacks which can be captured efficiently [Farvardin and Reppy 2020]. Then, a natural question that arises is whether it is possible to compile effect handlers efficiently where the target platform does not directly support delimited continuations, for example, when compiling to C/LLVM, WASM [Haas et al. 2017], JavaScript, Java VM, .NET, etc.

In this paper we give a formalized translation and evaluation semantics from a typed effect handler calculus into a plain typed lambda calculus as a sequence of refinements:

(1) First we show how effect handler semantics can be expressed using standard *multi-prompt delimited control* semantics [Forster et al. 2019; Gunter et al. 1995] (Section 2.4).

Authors' addresses: Ningning Xie, University of Hong Kong, China, xnning@hku.hk; Daan Leijen, Microsoft Research, USA, daan@microsoft.com.

2475-1421/2021/8-ART71
https://doi.org/10.1145/3473576

(2) We refine this semantics further to *evidence passing semantics* (EPS) where the *evidence* for a handler prompt in the evaluation context is pushed down to each effect operation as an *evidence vector* (Section 2.5 and 3.1). This makes performing an operation a *local* transition that no longer needs to search through the evaluation context (Section 2.6 and 3.2).
(3) Next we also localize yielding to a handler prompt by *bubbling* each yield through the evaluation context instead of capturing in one step (Section 2.7 and 4.1). This closely follows the effect handler semantics as given by Pretnar [2015].
(4) With all evaluation transitions localized, we can now define a direct *monadic translation* of effect handlers into a plain typed lambda calculus using a multi-prompt monad (Section 2.9, 2.11, and 4). Such program can be directly compiled to any target platform (including C/LLVM, WASM, JavaScript, Java VM, .NET, etc) without requiring special runtime mechanisms.

Aside from the novel evidence passing semantics, many parts of the refinements are known compilation techniques for effect handlers – but we believe we are the first to formalize each within a single polymorphically typed framework (combined with evidence passing semantics). Specifically, we make the following contributions:

- We formalize each refinement and translation, and show they are sound and semantics preserving (Section 3 and 4). Along the way, we explore various interesting points in the design space:
  - The use of *segmented stacks* for implementing effect handlers in a *direct* way (as used by multi-core OCaml [Sivaramakrishnan et al. 2021]) versus translation into a multi-prompt monad (Section 2.4): segmented stacks need a dedicated runtime system but can capture and resume an operation in constant time (for *one-shot* resumptions), while a multi-prompt monad is linear in the continuation points.
  - Using *insertion-* versus *canonical* ordered evidence vectors (Section 2.5): the former is efficient to construct but needs a linear lookup for each operation, while a canonical vector is more expensive to construct upfront but can use constant time lookup for operations.
  - Using *short-cut resumptions* to minimize the stack usage of a resumption while increasing sharing of continuation points (Section 2.8); a similar technique is used in [Kiselyov and Ishii 2015] to compose monadic binds in an effect monad.
  - Using *bind-inlining* and *join-point sharing* for improved efficiency when translating into the multi-prompt monad (Section 2.10).
- Our evidence passing *semantics* (EPS) is a generalization of the work on evidence passing *translation* (EPT) [Xie et al. 2020]. In particular, EPT can only express a subset of full effect handlers that are restricted to *scoped* resumptions only, whereas EPS lifts the restriction and can fully express effect handlers (Section 2.12 and 3.1).
- We give the first formal account of optimized *tail-resumptive* operation semantics and show how this can evaluate an operation in-place and avoid performing an expensive yield-and-resume cycle in the majority of effect operations (Section 2.6 and 3.2). The tail-resumptive optimization is surprisingly subtle to get correct – in particular in combination with *unscoped resumptions* which we illustrate in Section 2.12.2. We prove the correctness of the tail-resumptive optimization by showing that an optimized program is contextually equivalent to the original one.
- We have implemented our techniques as a monadic library for effect handlers in Haskell, called *Mp.Eff* (for "multi-prompt effect") [Xie and Leijen 2021b], generalizing the *Ev.Eff* library based on EPT [Xie and Leijen 2020]. Our implementation is based on insertion-ordered evidence vectors.

- We have also implemented our techniques in the Koka programming language [Leijen 2020] compiling to standard C code (Section 2.11). The implementation uses canonical evidence vectors, short-cut resumptions, bind-inlining, and join-point sharing.
- We benchmarked the Koka implementation against four other implementations of effect handlers that compile to native code: the current state-of-the-art *direct* implementation of effect handlers in multi-core OCaml which uses a dedicated runtime system based on segmented stacks; our *Mp.Eff* Haskell library; the *Ev.Eff* Haskell library which has been shown by Xie and Leijen [2020] to perform very well compared to other Haskell effect handler libraries [Schrijvers et al. 2019; Wu and Schrijvers 2015; Wu et al. 2014]; and finally the *libhandler* C library which implements effect handlers directly in C by copying fragments of the stack [Leijen 2017a]. Comparing across systems and languages is always tricky but the results clearly indicate that our approach can have competitive performance (Section 5).

The metatheory proofs are available in the technical report [Xie and Leijen 2021a], and the *Mp.Eff* Haskell library and benchmarks are available online [Leijen 2021; Xie and Leijen 2021b].

2 OVERVIEW

We start with a short discussion and examples of basic effect handlers and follow with an overview of each of our semantic refinements and translation techniques. We refer to other work [Hillerström and Lindley 2016; Leijen 2017b; Pretnar 2015] for further examples of effect handlers.

2.1 Algebraic Effects

With algebraic effect handlers, an effect l defines a set of operations *op*. For example, we can have a reader effect with an *ask* operation

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

and we can perform the *ask* operation writing perform *ask* (). A handler (handler h v) takes a list of operation clauses in h, and a computation v to be handled. Each operation clause in h takes the form $\mathit{op} \mapsto f$, providing the implementation f for the operation *op* from the handled effect where the implementation f is of form $\lambda x.\ \lambda k.\ e$: x binds the operation argument, and k binds the captured *resumption* that can be used to resume to the original call-site with the operation result. For example, we can handle the reader effect by always resuming with the constant 1:

$$h^{\mathit{read}} \ = \ \{\ \mathit{ask} \mapsto \lambda x.\ \lambda k.\ k\ 1\ \}$$

where the expression handler h^{read} ($\lambda\_$. perform *ask* () + perform *ask* ()) evaluates to 2. The following evaluation rules give the essence of the untyped semantics for algebraic effect handlers [Xie and Leijen 2020]:

$$\begin{array}{llll}
(\mathit{app}) & (\lambda x.\ e)\ v & \longrightarrow & e[x{:=}v] \\
(\mathit{handler}) & \mathsf{handler}\ h\ f & \longrightarrow & \mathsf{handle}\ h\ (f\ ()) \\
(\mathit{return}) & \mathsf{handle}\ h\ v & \longrightarrow & v \\
(\mathit{perform}) & \mathsf{handle}\ h\ \mathsf{E}[\mathsf{perform}\ \mathit{op}\ v] & \longrightarrow & f\ v\ (\lambda x.\ \mathsf{handle}\ h\ \mathsf{E}[x]) \\
 & & & \quad \text{iff } \mathit{op} \notin \mathsf{bop}(\mathsf{E}) \wedge (\mathit{op} \mapsto f) \in h
\end{array}$$

Rule (*app*) is standard β-reduction and applies a function to a value v by substituting x for the argument v in the function body. The (*handler*) takes a computation f, and applies the computation to a unit value under a new frame handle h. The computation to be handled (f) is always a unit-taking function as in [Xie et al. 2020], which essentially corresponds to a suspended computation as in the call-by-push-value approach [Levy 2006] used in several algebraic effect systems [Kammar and Pretnar 2017; Plotkin and Pretnar 2013].

The handle frame is only generated by handler, and treated as a strictly internal frame. When handling a computation under a handle h frame, there are two possible situations. In the first case, the computation evaluates to a value and the (*return*) transition discards the handle h frame and propagates the value. The second case captures the essence of algebraic effects handlers where an operation is *handled*. In rule (*perform*), perform $op\ v$ calls an effect operation op by providing the operation argument v. The handle h frame handles the operation by applying the operation implementation f to the operation argument v, and the *resumption* ($\lambda x.$ handle $h\ \mathsf{E}[x]$). The resumption captures the original handle, as well as the whole *evaluation context* E between handle and the operation call.

An evaluation context E is essentially an expression with a hole ($\square$) in it, and the notation $\mathsf{E}[e]$ represents the expression obtained by plugging e into the hole of E (e.g., $(f\ (g\ \square))[x] = f\ (g\ x)$). In this rule, the condition $op \notin \mathsf{bop}(\mathsf{E})$ indicates that op is not in the bound operations of E, i.e. not handled by any handle frames in E, ensuring that h is always the *innermost* handle frame for the effect that handles the operation.

2.2 Examples

Here we consider some standard examples of algebraic effects, and we refer the reader to other work for more examples as well as practical uses of effect handlers [Bauer and Pretnar 2015; Hillerström and Lindley 2016; Kammar et al. 2013; Leijen 2017b; Pretnar 2015; Xie et al. 2020]. In the examples, we use $x \leftarrow e_1;\ e_2$ as a shorthand for $(\lambda x.\ e_2)\ e_1$, and use $e_1; e_2$ for $(\lambda\_.\ e_2)\ e_1$, where $\lambda\_$ denotes a lambda whose binding is not used in the body.

Exceptions. The following definition defines an effect *exn* with one operation *throw*.

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

Given a datatype Maybe with two constructors Just and Nothing, we can define a handler for exceptions that reifies any exceptional computation with a Maybe result to return Nothing on an exception:

$$h^{exn} = \{\ throw \mapsto \lambda x.\ \lambda k.\ \mathsf{Nothing}\ \}$$

For example, suppose we define safe division as:

$$safediv = \lambda x\ y.\ \mathsf{if}\ (y == 0)\ \mathsf{then\ perform}\ throw\ ()\ \mathsf{else}\ x/y$$

then we have

$$\begin{array}{ll}
\quad \mathsf{handler}\ h^{exn}\ (\lambda_.\ \mathsf{Just}\ (safediv\ 42\ 2)) & \quad \mathsf{handler}\ h^{exn}\ (\lambda_.\ \mathsf{Just}\ (safediv\ 42\ 0)) \\
\longmapsto^* \mathsf{handle}\ h^{exn}\ (\mathsf{Just}\ (42/2)) & \longmapsto^* \mathsf{handle}\ h^{exn}\ (\mathsf{Just}\ (\mathsf{perform}\ throw\ ())) \\
\longmapsto \mathsf{handle}\ h^{exn}\ (\mathsf{Just}\ 21) & \longmapsto (\lambda x.\ \lambda k.\ \mathsf{Nothing})\ ()\ (\lambda x.\ \mathsf{handle}\ h^{exn}\ (\mathsf{Just}\ x)) \\
\longmapsto \mathsf{Just}\ 21 & \longmapsto^* \mathsf{Nothing}
\end{array}$$

We use the notation $\longmapsto$ to allow expressions to take steps ($\longrightarrow$) inside evaluation contexts, where $\longmapsto^*$ is the transitive reflexive closure of $\longmapsto$, and $\longmapsto^+$ is the transitive closure of $\longmapsto$.

Reader. In the previous example we did not make use of the operation argument (x) or the resumption (k). Let's consider this time the evaluation of our first example with the *reader* effect:

$$\begin{array}{ll}
 & \mathsf{handler}\ h^{read}\ (\lambda_.\ \mathsf{perform}\ ask\ ()\ +\ \mathsf{perform}\ ask\ ()) \\
\longmapsto^* & \mathsf{handle}\ h^{read}\ (\mathsf{perform}\ ask\ ()\ +\ \mathsf{perform}\ ask\ ()) \\
\longmapsto & (\lambda x.\ \lambda k.\ k\ 1)\ ()\ (\lambda x.\ \mathsf{handle}\ h^{read}\ (x\ +\ \mathsf{perform}\ ask\ ())) \\
\longmapsto^* & (\lambda x.\ \mathsf{handle}\ h^{read}\ (x\ +\ \mathsf{perform}\ ask\ ()))\ 1 \\
\longmapsto & \mathsf{handle}\ h^{read}\ (1\ +\ \mathsf{perform}\ ask\ ()) \quad \longmapsto^* (\lambda x.\ \mathsf{handle}\ h^{read}\ (1\ +\ x))\ 1 \quad \longmapsto^* 2
\end{array}$$

where both *ask* operations resume back to the original calling context with a result.

State. We can define a state handler using the monadic encoding [Kammar and Pretnar 2017], where performing an operation returns a function that takes in the current state.

$$st\ \{\ get : ()\to\alpha,\ set : \alpha\to()\ \} \qquad h^{st} = \{get \mapsto \lambda x.\ \lambda k.\ (\lambda y.\ k\ y\ y),\ set \mapsto \lambda x.\ \lambda k.\ (\lambda y.\ k\ ()\ x)\ \}$$

The following program starts with an initial state 0.

$$\begin{array}{ll} & (\text{handler } h^{st}\ (\lambda\_.\ \text{perform } set\ 21;\ w \leftarrow \text{perform } get\ ();\ (\lambda z.\ w\ +\ w))\ ))\ 0 \\ \longmapsto & (\text{handle } h^{st}\ (\text{perform } set\ 21;\ w \leftarrow \text{perform } get\ ();\ (\lambda z.\ w\ +\ w))\ ))\ 0 \end{array}$$

In the following derivation, we make use of the *dot notation* [Xie and Leijen 2020]. Specifically, the notation $\mathsf{E}_1 \bullet \mathsf{E}_2$ composes two evaluation contexts by plugging E_2 into the hole of E_1, resulting in a new evaluation context. The $(\bullet)$ notation is right-associative and has the lowest precedence, so we often write $\mathsf{E}_1 \bullet \mathsf{E}_2$ instead of $(\mathsf{E}_1) \bullet \mathsf{E}_2$. The notation $\mathsf{E} \bullet e$ has the same meaning as $\mathsf{E}[e]$, which plugs e into the hole of E, resulting in a new expression. Using the dot notation, the evaluation order of expressions becomes more apparent, and it is now easier to discuss one specific frame in the chain of evaluation contexts. We start by rewriting the last expression using the dot notation as:

$$= \quad \square\ 0 \bullet \text{handle } h^{st}\ \square \bullet (\square;\ w \leftarrow \text{perform } get\ ();\ (\lambda z.\ w\ +\ w)) \bullet \text{perform } set\ 21$$

For conciseness, we also often omit a trailing $\square$ in an application context $e\ \square \bullet \mathsf{E}$ and write instead $e \bullet \mathsf{E}$; this is usually the case for handle expressions:

$$= \quad \square\ 0 \bullet \text{handle } h^{st} \bullet (\square;\ w \leftarrow \text{perform } get\ ();\ (\lambda z.\ w\ +\ w)) \bullet \text{perform } set\ 21$$

Writing contexts this way, it shows more clearly the stack of evaluation frames with the expression under evaluation at the end. We can now continue evaluating as:

$$\begin{array}{ll} \longmapsto^* & \square\ 0 \bullet (\lambda y.\ k\ ()\ 21) \quad \text{with } k = \lambda x.\text{handle } h^{st} \bullet (\square;\ w \leftarrow \text{perform } get\ ();\ (\lambda z.w + w)) \bullet x \\ = & (\lambda y.\ k\ ()\ 21)\ 0 \quad \longmapsto k\ ()\ 21 \\ \longmapsto & \square\ 21 \bullet \text{handle } h^{st} \bullet (\square;\ w \leftarrow \text{perform } get\ ();\ (\lambda z.\ w\ +\ w)) \bullet () \\ = & \square\ 21 \bullet \text{handle } h^{st} \bullet ((\ )\ ;\ w \leftarrow \text{perform } get\ ()\ ;\ (\lambda z.\ w\ +\ w)) \\ \longmapsto & \square\ 21 \bullet \text{handle } h^{st} \bullet (w \leftarrow \square;\ (\lambda z.\ w\ +\ w)) \bullet \text{perform } get\ () \quad \longmapsto^* 42 \end{array}$$

While this is a nice example of the expressiveness of effect handlers, it is clearly not the most efficient way to express mutable state. In practice, state can be implemented more efficiently using *parameterized handlers* [Plotkin and Pretnar 2009] or a primitive state handler [Xie and Leijen 2020]. Moreover, using the more efficient implementations allow state handlers to be *tail-resumptive* (Section 2.6).

Non-determinism. By having the resumption k available when handling, we can actually resume more than once. In the handler of *amb*, we implement non-determinism by collecting all possible results in a list by resuming the resumption twice, each time with one boolean result.

$$\begin{array}{l} amb\ \{\ flip : () \to bool\ \} \\ h^{amb} = \{\ flip \mapsto \lambda\_\ k.\ \ xs \leftarrow k\ \text{True};\ ys \leftarrow k\ \text{False};\ xs\ ++\ ys\ \} \end{array} \qquad \begin{array}{l} \text{handler } h^{amb}\ (\lambda\_.\ \ x \leftarrow \text{perform } flip\ ();\ y \leftarrow \text{perform } flip\ ();\ [x\ \&\&\ y]) \\ \longmapsto^* [\text{True}, \text{False}, \text{False}, \text{False}] \end{array}$$

2.3 Compiling Effect Handlers

As the examples show, algebraic effect handlers can be very expressive. Unfortunately, their expressive power also makes it not easy to compile them efficiently. The main culprit is the (*perform*) rule:

$$\text{handle } h\ \mathsf{E}[\text{perform } op\ v] \longrightarrow f\ v\ (\lambda x.\ \text{handle } h\ \mathsf{E}[x]) \quad \text{iff } op \notin \text{bop}(\mathsf{E})\ \wedge\ (op \mapsto f) \in h$$

This single rule combines two potentially expensive runtime operations:

(1) *Searching*: The innermost handler for *op* must be found which usually requires a linear search through the current handlers in the evaluation context (i.e. search up through the stack frames).
(2) *Capturing*: After finding the handler clause f, we need to capture the evaluation context (i.e. stack and registers) up to the found handler, and create a *resumption* function ($\lambda x.$ handle $h\ \mathsf{E}[x]$) which restores the captured context when invoked with a result. An added complication is that in the general case such resumption may never be called (as in h^{exn}), or invoked more than once (as in h^{amb}), which can present difficulties in the runtime (for scanning GC roots for example).

Capturing and restoring resumptions can be done relatively efficiently if the target runtime system implements segmented stacks [Farvardin and Reppy 2020] – this is used in multi-core OCaml [Dolan et al. 2015] for example, where segmented stacks split the stack at each handler so that a one-shot resumption can be implemented efficiently by switching back to a previous stack segment [Sivaramakrishnan et al. 2021]). However, many target platforms do not support directly capturing parts of the stack at all, like compilation to C (as in Koka), WASM, .NET, the Java VM, JavaScript, etc, and in these cases it is not even possible to implement (*perform*) in any direct way.

In this paper we address these compilation and runtime challenges by presenting various refinements of the operational semantics in combination with source translations. Each of these steps enables further optimizations and implementations, and we explore various interesting points in the design space along the way.

2.4 Multi-Prompt Semantics

As a first step, we are going to split the (*perform*) operation into two parts where we separate the searching for a handler from capturing and restoring a resumption. To capture and restore a resumption we are going to use standard (typed) multi-prompt delimited control [Gunter et al. 1995]: instead of a handle h frame, we install a prompt $m\ h$ frame that is uniquely identified with a *marker* m, and performing an operation will use a yield $m\ f$ frame to yield to such prompt.

As an example, consider again the *reader* effect handler $h^{read} = \{ask \mapsto f\}$ with $f = \lambda x.\lambda k.\ k\ 1$, where we have the following evaluation (rewritten using the dot notation):

$$
\begin{array}{ll}
 & \mathsf{handler}\ h^{read}\ (\lambda\_.\ \mathsf{perform}\ ask\ ()\ +\ \mathsf{perform}\ ask\ ()) \\
\longmapsto^* & \mathsf{handle}\ h^{read}\ \bullet\ (\Box + \mathsf{perform}\ ask\ ())\ \bullet\ \mathsf{perform}\ ask\ () \\
\longmapsto & f\ ()\ (\lambda x.\ \mathsf{handle}\ h^{read}\ \bullet\ (\Box + \mathsf{perform}\ ask\ ())\ \bullet\ x) \\
\ldots &
\end{array}
$$

When using multi-prompt semantics, the first transition now installs a prompt $m\ h^{read}$ frame *instead* of a handle frame, where m is a unique *marker* identifying the prompt:

$$
\begin{array}{ll}
 & \mathsf{handler}\ h^{read}\ (\lambda\_.\ \mathsf{perform}\ ask\ ()\ +\ \mathsf{perform}\ ask\ ()) \\
\longmapsto^* & \mathsf{prompt}\ m\ h^{read}\ (\mathsf{perform}\ ask\ +\ \mathsf{perform}\ ask\ ()) \\
= & \mathsf{prompt}\ m\ h^{read}\ \bullet\ (\Box + \mathsf{perform}\ ask\ ())\ \bullet\ \mathsf{perform}\ ask\ ()
\end{array}
$$

The next transition shows how we separate searching from capturing – perform *ask* () now only finds the handler clause f but defers yielding to the prompt by using an explicit yield frame:

$$\longmapsto \mathsf{prompt}\ m\ h^{read}\ \bullet\ (\Box + \mathsf{perform}\ ask\ ())\ \bullet\ \mathsf{yield}\ m\ (\lambda k.\ f\ ()\ k)$$

The yield $m\ g$ has two arguments: the marker m that uniquely identifies the prompt to yield to, and a function g that is applied to the resumption when reaching the prompt. Through the marker m, we can yield directly to the corresponding prompt which captures and applies the resumption:

$\longmapsto (\lambda k.\ f\ ()\ k)\ (\lambda x.\ \text{prompt}\ m\ h^{read}\ \bullet\ (\square +\ \text{perform}\ ask\ ())\ \bullet\ x)$
$\longmapsto f\ ()\ (\lambda x.\ \text{prompt}\ m\ h^{read}\ \bullet\ (\square +\ \text{perform}\ ask\ ())\ \bullet\ x)$
$\ldots$

This separation of concerns does not immediately buy us much but, as we will see, it opens up the way for optimizing each part individually by (1) using evidence passing semantics to avoid searching, and (2) using a monadic translation to enable capturing without requiring a special runtime system. Moreover, multi-prompt delimited control is one of the lowest level control operations that can be typed in the simply typed lambda calculus.

If one controls the target platform, it is possible to efficiently implement multi-prompt delimited control directly. This is done for example in multi-core OCaml using segmented stacks: here the call stack is split in segments where each prompt frame starts a fresh segment. The marker m can be implemented directly as the runtime pointer to that frame. Yielding up to a parent stack segment is now a constant time operation as only the stack segment pointer needs to be adjusted. Resuming once can also be done in constant time this way, but supporting multi-shot resumptions still requires a linear copy of the resumption stack segments (and one of the reasons why multi-shot resumptions are not directly supported in multi-core OCaml).

2.5 Evidence Passing Semantics

The (*perform*) operation is still a non-local transition as it searches through the evaluation context to find the innermost handler. We can make it local using *evidence passing semantics*, where we pass the current handlers in the evaluation context explicitly as an extra *evidence vector* w down to the perform operations. Instead of searching through the context, we can now look up the handler locally. Essentially, if the current evidence vector is w, then the (*perform*) rule becomes:

$$\text{perform}\ op\ v \longrightarrow \text{yield}\ m\ (\lambda k.\ f\ v\ k) \quad \text{where}\ (m, h) = w.l \wedge (op \mapsto f) \in h$$

The expression $w.l$ directly looks up the marker and handler (called *evidence*) for effect l from the evidence vector w. We apply the idea to our example, where we use the $\frown$ notation to indicate the current evidence vector and we sometimes omit the notation when it is irrelevant or obvious from the context. Evaluation always starts with an empty evidence vector $\langle\!\langle\rangle\!\rangle$:

$$\overbrace{\text{handler}\ h^{read}\ (\lambda\_.\ \text{perform}\ ask\ ()\ +\ \text{perform}\ ask\ ())}^{\langle\!\langle\rangle\!\rangle}$$

which evaluates into:

$$\longmapsto^* \overbrace{\text{prompt}\ m\ h^{read}}^{\langle\!\langle\rangle\!\rangle}\ \bullet\ \overbrace{(\square +\ \text{perform}\ ask\ ())\ \bullet\ \text{perform}\ ask\ ()}^{w\ =\ \langle\!\langle read : (m, h^{read})\rangle\!\rangle}$$

where the prompt frame modifies the evidence for rest of the evaluation context. At this point perform evaluates under an evidence vector $\langle\!\langle read : (m, h^{read})\rangle\!\rangle$, and we get:

$\longmapsto \text{prompt}\ m\ h^{read} \bullet (\square + \text{perform}\ ask\ ()) \bullet \text{yield}\ m\ (\lambda k.f\ ()\ k) \quad \text{where}\ (m,\ h^{read}) = w.read$
$\longmapsto (\lambda k.\ f\ ()\ k)\ (\lambda x.\ \text{prompt}\ m\ h^{read}\ \bullet\ (\square + \text{perform}\ ask\ ())\ \bullet\ x)$
$\longmapsto f\ ()\ (\lambda x.\ \text{prompt}\ m\ h^{read}\ \bullet\ (\square + \text{perform}\ ask\ ())\ \bullet\ x)$
$\ldots$

Using evidence passing semantics makes the (*perform*) transition localized which can potentially be more efficient than searching through the evaluation context. When we treat the evidence vector as an abstract datatype there are two interesting variants depending on how the vectors are ordered:

(1) *Insertion order*: Insert handler evidence in the order of the actual handlers in the evaluation context. This is straightforward and also the approach we take in the associated Haskell library. However, it means that the lookup operation $w.l$ still needs to search linearly through the vector for the "innermost" handler. One way to implement such vector is as a linked list where each handler pushes itself on the list. Since evidence vectors are not first-class values, we can actually allocate this list on the evaluation stack directly and as such it becomes a linked list of handlers at runtime – this is exactly how various languages (e.g. C++ compilers used to do this) and systems (e.g. Windows structured exception handling) implement exception handlers where the w parameter is a pointer to the head of the exception handler list.
(2) *Canonical order*: Use a lexicographic order of the handler evidence based on their effect label. This requires a strongly typed calculus but it means that if the effect type is fully known at compile time, we can *statically* determine the index for a particular effect in the runtime evidence vector. For example, in systems that keep track of the effect type of expressions using *row types* [Hillerström and Lindley 2016; Leijen 2017b], the effect type of our example perform *ask* () is the singleton effect row $\langle read \rangle$, and we know statically that the dynamic runtime evidence vector will have the form $\langle\!\langle read : \_ \rangle\!\rangle$. We can thus replace the linear runtime lookup $w.read$ with a *constant-time* array access $w[0]$ instead. This is the approach used in the Koka compiler.

2.6 Tail-Resumptive Operations

With evidence semantics in place, the only expensive operation left is yielding and capturing a resumption. Fortunately, we can often avoid doing a full yield: almost all common operations in practice happen to be *tail resumptive* where the operation clause has the form:

$$op \mapsto \lambda x.\ \lambda k.\ k\ e \qquad \text{where } k \notin \mathsf{fv}(e)$$

For example, the *ask* operation in our h^{read} handler is of this form[2]. It turns out we can perform such operations *in place*: instead of yielding up and eventually resuming with the final result, we can directly evaluate e on the current stack without doing an expensive yield followed by a resume. To this end, we extend each evidence in the evidence vector to store a triple (m, h, w) (instead of a tuple (m, h)), where the third component w is the *evidence context*: this is the evidence vector under which the handler h is defined and is used for the evaluated-in-place expression. We illustrate the use of this in our running example:

$$\overbrace{\text{handler } h^{read}\ (\lambda\_.\ \text{perform } ask\ ()\ +\ \text{perform } ask\ ())}^{\langle\rangle}$$
$$\longmapsto^* \overbrace{\text{prompt } m\ h^{read}}^{\langle\rangle}\ \bullet\ \overbrace{(\square +\ \text{perform } ask\ ())\ \bullet\ \text{perform } ask\ ()}^{\langle\!\langle read : (m, h^{read}, \langle\rangle) \rangle\!\rangle}$$

[2]While h^{state} is not tail-resumptive here, implementations of state in practice are usually based on parameterized handlers [Plotkin and Pretnar 2009] or primitive state [Xie and Leijen 2020], both of which are tail-resumptive. The h^{exn} and h^{amb} handlers are not tail-resumptive because of their special nature (aborting the computation and non-determinism, respectively). Furthermore, in practice we can also allow any clause that can be rewritten into the tail-resumptive form – for example $\lambda x\ k.$ if $x == 0$ then $k\ 1$ else $k\ 2$ which can be transformed to $\lambda x\ k.\ k$ (if $x == 0$ then 1 else 2).

Here, the evidence vector for perform is $\langle\!\langle read : (m, h^{read}, \langle\!\rangle) \rangle\!\rangle$ and we can locally find the operation clause $ask \to \lambda x\ k.\ k\ 1 \in h^{read}$ and determine that it is tail-resumptive. Instead of generating yield as before, we instead evaluate e (as in $\lambda x.\ \lambda k.\ k\ e$, with e being 1 in this case) *in-place*:

$$
\begin{aligned}
&\longmapsto \mathsf{prompt}\ m\ h^{read} \bullet (\square + \mathsf{perform}\ ask\ ()) \bullet \mathsf{under}\ read \bullet 1 \\
&\longmapsto \mathsf{prompt}\ m\ h^{read} \bullet (\square + \mathsf{perform}\ ask\ ()) \bullet 1 \\
&\longmapsto \mathsf{prompt}\ m\ h^{read}\ (1 + \mathsf{perform}\ ask\ ()) \\
&\ldots
\end{aligned}
$$

The operation clause is now evaluated in-place – but note it needs to be evaluated under an under l frame. Such frame ensures that if the operation clause e itself performs operations, these are resolved correctly with respect to the actual handler up in the evaluation context. Consider for example the following reader handler:

$$h_2 = \{\ ask \mapsto \lambda x.\lambda k.\ k\ (\mathsf{perform}\ ask\ ()\ +\ 1)\ \}$$

Here the operation clause is tail-resumptive, and itself performs an *ask* operation. Now consider:

$$
\begin{aligned}
&\quad \mathsf{handler}\ h^{read}\ (\lambda\_.\ \mathsf{handler}\ h_2\ (\lambda\_.\ \mathsf{perform}\ ask()))\\
&\longmapsto^* \overbrace{\mathsf{prompt}\ m_1\ h^{read}}^{\langle\rangle} \bullet \overbrace{\mathsf{prompt}\ m_2\ h_2}^{w_1 = \langle\!\langle read : (m_1, h^{read}, \langle\rangle) \rangle\!\rangle} \bullet \overbrace{\mathsf{perform}\ ask\ ()}^{w_2 = \langle\!\langle read : (m_2, h_2, w_1),\ read : (m_1, h^{read}, \langle\rangle) \rangle\!\rangle}
\end{aligned}
$$

At this point, the evidence vector at the second prompt is $w_1 = \langle\!\langle read : (m_1, h^{read}, \langle\!\rangle) \rangle\!\rangle$, but the evidence vector at the perform contains two entries: $w_2 = \langle\!\langle read : (m_2, h_2, w_1),\ read : (m_1, h^{read}, \langle\!\rangle) \rangle\!\rangle$. Here we see how the third member of the evidence always points to the "previous" evidence vector (e.g., w_1) under which a particular hander (e.g., h_2) is defined. If using insertion-ordered evidence vectors as a linked list, this is always just the tail of the list, but for canonical evidence vectors the previous vector must be kept explicitly. Since the operation clause is tail-resumptive, we get:

$$
\begin{aligned}
&\longmapsto \overbrace{\mathsf{prompt}\ m_1\ h^{read}}^{\langle\rangle} \bullet \overbrace{\mathsf{prompt}\ m_2\ h_2}^{w_1} \bullet \overbrace{\mathsf{under}\ read}^{w_2} \bullet \overbrace{\mathsf{perform}\ ask\ ()\ +\ 1}^{w_1} \\
&= \quad \mathsf{prompt}\ m_1\ h^{read} \bullet \mathsf{prompt}\ m_2\ h_2 \bullet \mathsf{under}\ read \bullet (\square + 1) \bullet \mathsf{perform}\ ask\ ()
\end{aligned}
$$

The evidence vector for the perform $ask\ ()\ +\ 1$ is now w_1 and *not* the unchanged w_2. Indeed, it would be incorrect to use w_2 or otherwise we would invoke the operation clause of h_2 again! The under *read* frame prevents this from happening and adjusts the evidence vector to the one under which the *read* handler is itself defined: this is exactly the third component of the evidence, $w_2.read.thd$, which is w_1 in our example. We now continue as:

$$
\begin{aligned}
&\longmapsto \overbrace{\mathsf{prompt}\ m_1\ h^{read}}^{\langle\rangle} \bullet \overbrace{\mathsf{prompt}\ m_2\ h_2}^{w_1} \bullet \overbrace{\mathsf{under}\ read}^{w_2} \bullet \overbrace{(\square + 1) \bullet \mathsf{under}\ read}^{w_1} \bullet \overbrace{1}^{\langle\rangle} \\
&\longmapsto \mathsf{prompt}\ m_1\ h^{read} \bullet \mathsf{prompt}\ m_2\ h_2 \bullet \mathsf{under}\ read \bullet (\square + 1) \bullet 1 \\
&\longmapsto \mathsf{prompt}\ m_1\ h^{read} \bullet \mathsf{prompt}\ m_2\ h_2 \bullet \mathsf{under}\ read \bullet 2 \quad \longmapsto^* 2
\end{aligned}
$$

Note that the second under *read* frame adjusts the evidence further to $\langle\!\rangle$ (which is $w_1.read.thd$). The correct formalization of under is subtle, and we will come back to this in Section 2.12.

2.7 Bubbling Yields

Using evidence semantics, perform is a local transition which only leaves yields as a non-local transition for non-tail-resumptive operations. We can further make yield m local by *bubbling* it up until it meets its corresponding prompt m frame in the evaluation context. That is, instead of capturing the delimited evaluation context E wholesale, we are going to build a resumption function piecemeal while bubbling up. To this end, we extend yield $m\ v$ with an extra argument as yield $m\ v\ k$ where k is the current partially built up continuation (starting out as identity).

Consider our earlier exception effect in Section 2.2 where:

$$\begin{array}{ll} & \mathsf{handler}\ h^{exn}\ (\lambda\_.\ \mathit{safediv}\ 42\ 0) \\ \longmapsto^* & \mathsf{handle}\ h^{exn}\ \bullet\ \mathsf{Just}\ \square\ \bullet\ \mathsf{perform}\ \mathit{throw}\ () \\ \longmapsto & (\lambda x\ k.\ \mathsf{Nothing})\ ()\ (\lambda x.\ \mathsf{handle}\ h^{exn}\ (\mathsf{Just}\ x)) \end{array}$$

When using yield bubbling we evaluate instead as (writing f for $\lambda x\ k.$ Nothing):

$$\begin{array}{ll} & \mathsf{handler}\ h^{exn}\ (\lambda\_.\ \mathit{safediv}\ 42\ 0) \\ \longmapsto^* & \mathsf{prompt}\ m\ h^{exn}\ \bullet\ \mathsf{Just}\ \square\ \bullet\ \mathsf{perform}\ \mathit{throw}\ () \\ \longmapsto & \mathsf{prompt}\ m\ h^{exn}\ \bullet\ \mathsf{Just}\ \square\ \bullet\ \mathsf{yield}\ m\ (\lambda k.\ f\ ()\ k)\ \mathit{id} \end{array}$$

At this point, the yield does a local transition and bubbles up only one step through the Just application, resulting in

$$\longmapsto \mathsf{prompt}\ m\ h^{exn}\ \bullet\ \mathsf{yield}\ m\ (\lambda k.\ f\ ()\ k)\ (\mathsf{Just} \circ \mathit{id})$$

Note how the resumption function changed from the initial identity id to the composition $\mathsf{Just} \circ \mathit{id}$. Generally, yields keep bubbling up this way extending their current resumption until they meet their target prompt:

$$\longmapsto^* f\ ()\ (\lambda x.\ \mathsf{prompt}\ m\ h^{exn}\ ((\mathsf{Just} \circ \mathit{id})\ x)) \quad \longmapsto^* \mathsf{Nothing}$$

Using bubbling removes any direct manipulation of the evaluation context E and only regular functions are used instead. The bubbling technique for implementing delimited continuations is well known [Felleisen et al. 1986; Parigot 1992] and used for example to give direct semantics to effect handlers [Kiselyov and Sivaramakrishnan 2017; Pretnar 2015].

2.8 Short-cut Resumptions

When bubbling up, a resumption is built up as a composition of continuations, $f_1 \circ \ldots \circ f_n \circ \mathit{id}$, and when resuming it is applied as $(f_1 \circ \ldots \circ f_n \circ \mathit{id})\ x$ which will recreate all f_1 to f_n application frames on the evaluation stack which can be expensive. Instead, in an implementation we can represent the composition as a list $[f_1, \ldots, f_n]$, and resume instead as $\mathit{resume}\ [f_1, \ldots, f_n]\ x$ where resume folds through the list from the end:

$$\mathit{resume}\ []\ x\ =\ x \qquad\qquad \mathit{resume}\ (\mathit{fs}\ +\!\!+\ [f])\ x\ =\ \mathit{resume}\ \mathit{fs}\ (f\ x)$$

This can be done efficiently by using a queue or array representation (as done in Koka) and also uses minimal stack space by evaluating just one f continuation at a time. Moreover, any *further* yields in a frame f_i will bubble up directly through the current resume and thus capture all f_1 to f_{i-1} continuations in one go (and will itself share those continuations through the various yields). We call these *short-cut resumptions* as these can be resumed by immediately starting at the deepest continuation point. This uses minimal stack space while increasing the use of shared continuations.

Note that while bubbling up we can also encounter prompt and under frames besides regular applications; for example, the final resumption may be of the form $f_1 \circ \ldots \circ f_i \circ \mathsf{prompt}\ m\ h \circ f_{i+1} \circ \ldots \circ f_n$. When resuming, we need to ensure that such prompt and under frames are properly restored and cannot use short-cut's for those. Of course we can still use resume for the application fragments surrounding the prompt/under frames, e.g. $\mathit{resume}\ [f_1, \ldots, f_i] \circ \mathsf{prompt}\ m\ h \circ \mathit{resume}\ [f_{i+1}, \ldots, f_n]$.

2.9 Monadic Translation

At this point all transitions are local and no longer capture the evaluation context explicitly. This means we are now able to translate our core calculus into a pure lambda calculus together with a multi-prompt delimited control monad. This is a straightforward transformation where every (effectful) expression is sequenced through a monadic bind. Our running example:

$$\mathsf{handler}\ h^{read}\ (\lambda\_.\ \mathsf{perform}\ \mathit{ask}\ ()\ +\ \mathsf{perform}\ \mathit{ask}\ ())$$

translates to the following monadic expression:

$$\text{handler } h^{read}\ (\lambda\_.\ \text{perform } ask\ ()\ \triangleright\ (\lambda x.\ \text{perform } ask\ ()\ \triangleright\ (\lambda y.\ \text{Pure}\ (x+y))))$$

where we write $(\triangleright)$ for monadic binding, and Pure for lifting pure expressions into the monad. Through the bind operation, the current continuation becomes explicit (as a function argument) and can be captured and resumed using regular function application, where bind is implemented essentially[3] as:

$$\begin{aligned} e \triangleright g\ =\ &\text{case } e \text{ of } \text{Pure } x \longrightarrow g\ x \\ &\quad\ \text{Yield } m\ f\ k \longrightarrow \text{Yield } m\ f\ (\lambda x.\ k\ x \triangleright g) \end{aligned}$$

Pure values are directly propagated while a yield bubbles up (upto its matching prompt) and appends each explicit continuation g to the built up resumption. Since all of this can be expressed in plain typed lambda calculus, this can be directly translated to almost any target platform – all control flow is now fully explicit.

2.10 Bind-Inlining and Join-Point Sharing

However, if done naively there may be a high cost to this translation: since every bind operation takes a lambda as its second argument this may lead to many closure allocations even for non-yielding code. Moreover, any direct tail-recursive calls are no longer directly tail-recursive as they occur under a lambda now!

To improve this we need two techniques: bind-inlining and join-point sharing. To avoid always allocating a lambda, we can use bind-inlining to simply inline every bind operation, expanding our example expression to:

$$\begin{aligned} &\text{handler } h^{read}\ (\lambda\_.\ \text{case perform } ask\ () \text{ of} \\ &\quad \text{Yield } m\ f\ k \longrightarrow \text{Yield } m\ f\ (\lambda z.\ k\ z \triangleright (\lambda x.\ \text{perform } ask\ () \triangleright (\lambda y.\ \text{Pure}\ (x+y)))) \\ &\quad \text{Pure } x \longrightarrow \text{case perform } ask\ () \text{ of } \text{Yield } m\ f\ k \longrightarrow \text{Yield } m\ f\ (\lambda z.\ k\ z \triangleright (\lambda y.\ \text{Pure}\ (x+y))) \\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{Pure } y \longrightarrow \text{Pure}\ (x+y)) \end{aligned}$$

For clarity, we did not inline the bind operations in an expansion itself. Nevertheless, we can already see that at every original bind operation, we duplicated the g argument in the expansion. This means that if we have a sequence of N statements, we may end up with 2^N duplications.

To avoid such expansion, we need to use *join-point sharing*: we consider every g argument as a join point, and rewrite the initial translation to make this sharing explicit:

$$\begin{aligned} &join_1\ =\ \lambda x\ y.\ \text{Pure}\ (x\ +\ y) \qquad\qquad join_2\ =\ \lambda x.\ \text{perform } ask\ () \triangleright (\lambda y.\ join_1\ x\ y) \\ &\text{hander } h^{read}\ (\lambda\_.\ \text{perform } ask\ () \triangleright join_2) \end{aligned}$$

From there, we perform bind-inlining only for non-*join* definitions, but also aggressively inline *join*-definitions for the Pure branches. This results effectively in a fully inlined *fast path* along the Pure branches:

$$\begin{aligned} &\text{handler } h^{read}\ (\lambda\_.\ \text{case perform } ask\ () \text{ of} \\ &\quad \text{Yield } m\ f\ k \longrightarrow \text{Yield } m\ f\ (\lambda z.\ k\ z \triangleright join_2) \\ &\quad \text{Pure } x \longrightarrow \text{case perform } ask\ () \text{ of } \text{Yield } m\ f\ k \longrightarrow \text{Yield } m\ f\ (\lambda z.\ k\ z \triangleright join_1\ x) \\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{Pure } y \longrightarrow \text{Pure}\ (x+y)) \end{aligned}$$

Note how the $join_1$ join point is shared by the $join_2$ definition as well, and the code expansion for N statements is now reduced from 2^N to $2N$. In practice, the Koka compiler does a type-selective transformation and leaves out monadic binds for functions that are total (since those will never yield) which further reduces code expansion by a large factor.

This strategy ensures that we have a *fast path* along each Pure branch: if no operation performs a full yield, no allocation happens along this path and tail-recursive calls are preserved (and as

[3] As shown in Section 4 the actual definition also propagates the current evidence vector as part of the monad.

such, this optimization works best when used together with tail-resumptive optimization). Only in the (hopefully rare) case that full yield is needed, the slow path along the Yield case is taken and a resumption is constructed on demand. When such a resumption is resumed, the execution is a bit slower as well as it takes the code path along the $join_n$ definitions where the binds are *not* inlined – this is the price we pay for limiting the expansion. Note though that if the function is recursive, any further recursive calls will again start at the fast path.

2.11 Compiling to C

At this point we can use regular compilation techniques to compile the plain lambda calculus to a target platform. As an example, we show here how Koka compiles to standard C. In our final calculus all effectful functions return a monadic result, either Pure or Yield. Since this monad is internal to the compiler we can optimize its representation: we always return results normally assuming Pure, and set a (thread-local) flag to indicate yielding (in which case the actual returned value is ignored). Moreover, every function has one extra parameter that holds the (thread-local) context `ctx` which contains the current evidence vector (`ctx→w`), and the yielding flag (`ctx→is_yielding`). For example, the expression $\lambda\_.$ perform *ask* () + perform *ask* () translates essentially as:

```
int expr( unit_t u, context_t* ctx) {
  int x = perform_ask( ctx→w[0], unit, ctx );
  if (ctx→is_yielding) { yield_extend(&join2,ctx); return 0; }
  int y = perform_ask( ctx→w[0], unit, ctx );
  if (ctx→is_yielding) { yield_extend(alloc_closure_join1(x,ctx),ctx); return 0; }
  return (x+y); }
```

Here we see how the evidence for the *read* handler is selected from the current evidence vector as `ctx→w[0]`. Here the offset 0 is known as the effect type is $\langle read \rangle$ and Koka uses canonical evidence vectors. If the effect row type was not fully known, e.g., a polymorphic row type $\langle read \mid \mu \rangle$, the code would instead be `find_ev(ctx→w,tag_read)` to find the evidence dynamically. When yielding, the `yield_extend` calls are used to extend the currently build up resumption (as part of the `ctx`) with the current continuation (which is usually a join point).

There is still an overhead in always needing to check after every effectful call if we are yielding or not. Fortunately, this seems quite cheap on modern processors and the condition can be predicted well. In the future we would like to leverage C compiler primitives to implement the `is_yielding` flag in the processor *carry* flag as suggested by recent C++ proposals for error handling [Sutter 2019].

2.12 Generalized Evidence Passing

The closest related work to our approach is [Xie et al. 2020], which uses *evidence-passing translation* (EPT). Even though similar in its purpose, EPT differs fundamentally from our approach. First, while our evidence-passing semantics provides a set of direct evaluation rules for the algebraic effect calculus, EPT is defined via *elaboration* from the algebraic effect calculus into an *evidence calculus*. Second, our generalized evidence-passing semantics works for all algebraic effect handler programs, whereas in EPT resumptions are limited to *scoped* resumptions only – that is, resumptions can only be used under the same handler context as captured by the handler.

Specifically, in EPT, as the evidence vector is passed *statically* during elaboration, it is determined *before* running the program. However, the statically passed-in evidence vector may, as the program evaluates, no longer match the handlers in the current dynamic evaluation context (and in such case, EPT raises a runtime error). Scoped resumptions restrict the expressiveness of algebraic effects, including the use of *shallow handlers* [Hillerström and Lindley 2018] and code migration that resumes continuations on a different host [Kiselyov et al. 2006].

2.12.1 Non-Scoped Resumptions. We illustrate the problem of non-scoped resumptions using the following *evil* effect as shown by Xie et al. [2020]:

$$\mathit{evil}\ \{\ \mathit{evil}\ :\ () \to ()\ \} \qquad\qquad h^{\mathit{evil}} = \{\ \mathit{evil} \mapsto \lambda x\ k.\ k\ \}$$

The handler h^{evil} illustrates again the expressiveness of effect handlers: the captured resumption is a first-class value and thus can be returned directly, and in this example we are going to resume it later under a changed handler context. Suppose we have another reader handler that always returns 2:

$$h^{\mathit{read}_2} = \{\ \mathit{ask} \mapsto \lambda x\ k.\ k\ 2\ \}$$

Consider the following program, where $f = (\lambda k.\ \text{handler}\ h^{\mathit{read}_2}\ (\lambda\_.\ k\ ()))$, which takes a continuation and resumes it under a new handler (ignoring tail-resumptive optimization for now):

$$\begin{aligned} & f\ (\text{handler}\ h^{\mathit{read}}\ (\lambda\_.\ \text{handler}\ h^{\mathit{evil}}\ (\lambda\_.\ \text{perform}\ \mathit{ask}\ ();\ \text{perform}\ \mathit{evil}\ ();\ \text{perform}\ \mathit{ask}\ ()))) \\ \longmapsto\ & \overbrace{f\ \bullet\ \text{prompt}\ m_1\ h^{\mathit{read}}}^{\langle\rangle}\ \bullet\ \overbrace{\text{prompt}\ m_2\ h^{\mathit{evil}}}^{w_1 = \langle \mathit{read} : (m_1,\ h^{\mathit{read}}) \rangle} \\ & \qquad \bullet\ \overbrace{(\square\ ;\ \text{perform}\ \mathit{evil}\ ()\ ;\ \text{perform}\ \mathit{ask}\ ())}^{w_2 = \langle \mathit{evil} : (m_2,\ h^{\mathit{evil}}),\ \mathit{read} : (m_1, h^{\mathit{read}}) \rangle}\ \bullet\ \text{perform}\ \mathit{ask}\ () \end{aligned}$$

It may seem that both *ask* operations will return 1 as they both have $\mathit{read} : (m_1, h^{\mathit{read}})$ in the evidence vector w_2 but, as we will see, that is not the case! The first *ask* returns 1 as expected though:

$$\longmapsto f\ \bullet\ \text{prompt}\ m_1\ h^{\mathit{read}}\ \bullet\ \text{prompt}\ m_2\ h^{\mathit{evil}}\ \bullet\ (\square\ ;\ \text{perform}\ \mathit{evil}\ ()\ ;\ \text{perform}\ \mathit{ask}\ ())\ \bullet\ 1$$

However, before we can handle the second *ask*, the operation *evil* is performed, which captures the second *ask* in the resumption k:

$$\begin{aligned} & \longmapsto f\ \bullet\ \text{prompt}\ m_1\ h^{\mathit{read}}\ \bullet\ \text{prompt}\ m_2\ h^{\mathit{evil}}\ \bullet\ (\square\ ;\ \text{perform}\ \mathit{ask}\ ())\ \bullet\ \text{perform}\ \mathit{evil}\ () \\ & \longmapsto \overbrace{f \bullet \text{prompt}\ m_1\ h^{\mathit{read}}}^{\langle\rangle}\ \bullet\ \overbrace{k}^{w_1} \quad \text{with}\ k = \lambda x.\ \text{prompt}\ m_2\ h^{\mathit{evil}} \bullet (\square\ ;\ \text{perform}\ \mathit{ask}\ ()) \bullet x \end{aligned} \tag{1}$$

As k is a value, it is propagated through the prompt m_1 frame:

$$\longmapsto f\ k \quad \longmapsto \text{handler}\ h^{\mathit{read}_2}\ (\lambda\_.\ k\ ())$$

At this point, the reader handler in the context is now changed to h^{read_2}:

$$\begin{aligned} & \longmapsto \overbrace{\text{prompt}\ m_0\ h^{\mathit{read}_2}}^{\langle\rangle}\ \bullet\ \overbrace{k\ ()}^{w_3 = \langle \mathit{read} : (m_0,\ h^{\mathit{read}_2}) \rangle} \\ & \longmapsto \overbrace{\text{prompt}\ m_0\ h^{\mathit{read}_2}}^{\langle\rangle}\ \bullet\ \overbrace{\text{prompt}\ m_2\ h^{\mathit{evil}}}^{w_3}\ \bullet\ \overbrace{(\square\ ;\ \text{perform}\ \mathit{ask}\ ())\ \bullet\ ()}^{w_4 = \langle \mathit{evil} : (m_2,\ h^{\mathit{evil}}),\ \mathit{read} : (m_0,\ h^{\mathit{read}_2}) \rangle} \end{aligned} \tag{2}$$

and the *ask* operation is performed under w_4 using the new h^{read_2} and thus evaluates to 2 (and not 1)!

EPT rejects this program at runtime by detecting the *non-scoped* resumption k: k is captured under w_1 at (1), but is later applied under w_3 at (2). In particular, in EPT, both *ask* operations *statically* receive w_2 as the evidence vector during elaboration to the evidence calculus. As such resuming k under a changed evidence vector means the statically received evidence vector does no longer match the dynamic handler context anymore, and is thus not allowed in their system. In contrast, our generalized evidence passing semantics correctly models the dynamic behavior of the evidence vector, and can express the full semantics of algebraic effect handlers.

2.12.2 Non-Scoped Resumptions with Tail-Resumptive Optimization. Xie et al. [2020] also describe the tail-resumptive optimization, and argue that tail-resumptive operations are examples of scoped resumptions, but do not provide any formalization of the optimization.

It turns out that the tail-resumptive optimization is more challenging with generalized evidence passing semantics, and our formalization goes beyond what is sketched in [Xie et al. 2020]. In particular, the interaction between non-scoped resumptions and tail-resumptive operations is subtle and the formalization of under is tricky to get right. We illustrate this by performing the previous *evil* example from *inside* a tail-resumptive operation:

$$tl\ \{\ tl : () \to Int\ \} \qquad h^{tl} = \{\ tl \mapsto \lambda x\ k.\ k\ (\text{perform } ask\ ();\ \text{perform } evil\ ();\ \text{perform } ask\ ())\ \}$$

Here we have the same sequence of operations as before, but this time these happen from inside an operation. Note that this operation is tail-resumptive, despite all effects performed before resuming. Now consider the following program, which performs tl under three handlers, and passes the result to f.

$$\begin{aligned} & f\ (\text{handler } h^{read}\ (\lambda\_.\ \text{handler } h^{evil}\ (\lambda\_.\ \text{handler } h^{tl}\ (\lambda\_.\ \text{perform } tl\ ()\ )))) \\ \longrightarrow^* & f \bullet \text{prompt } m_1\ h^{read} \bullet \text{prompt } m_2\ h^{evil} \bullet \text{prompt } m_3\ h^{tl} \bullet \text{perform } tl\ () \end{aligned}$$

We evaluate tl in-place under w_2, as h^{tl} is itself defined under w_2.

$$\begin{aligned} \longrightarrow^* & \overbrace{f \bullet \text{prompt } m_1\ h^{read}}^{\langle\rangle} \bullet \overbrace{\text{prompt } m_2\ h^{evil}}^{w_1 = \langle read : (m_1, h^{read}, \langle\rangle)\rangle} \bullet \overbrace{\text{prompt } m_3\ h^{tl}}^{w_2 = \langle evil : (m_2,\ h^{evil}, w_1), read : (m_1, h^{read}, \langle\rangle)\rangle} \\ & \quad \underset{w_3 = \langle tl : (m_3,\ h^{tl}, w_2),\ evil : (m_2,\ h^{evil}, w_1)\ ,\ read : (m_1, h^{read}, \langle\rangle)\rangle}{\bullet} \overbrace{\text{under } tl}^{w_2} \bullet \overbrace{(\square; \text{perform } evil\ (); \text{perform } ask\ ()) \bullet \text{perform } ask\ ()}^{w_2} \end{aligned}$$

ask is also tail-resumptive and gets evaluated in-place.

$$\begin{aligned} \longrightarrow & f \bullet \text{prompt } m_1\ h^{read} \bullet \text{prompt } m_2\ h^{evil} \bullet \text{prompt } m_3\ h^{tl} \\ & \bullet \text{under } tl \bullet (\square\ ;\ \text{perform } evil\ ();\ \text{perform } ask\ ()) \bullet \text{under } read \bullet 1 \end{aligned} \tag{3}$$

We then perform *evil*, which again captures the resumption and passes it to f.

$$\begin{aligned} \longrightarrow^* & f \bullet \text{prompt } m_1\ h^{read} \bullet \text{prompt } m_2\ h^{evil} \bullet \text{prompt } m_3\ h^{tl} \\ & \bullet \text{under } tl \bullet (\square;\ \text{perform } ask\ ()) \bullet \text{perform } evil \\ \longrightarrow^* & f\ k \ \text{ where } k = \lambda x.\text{prompt } m_2\ h^{evil} \bullet \text{prompt } m_3\ h^{tl} \bullet \text{under } tl \bullet (\square; \text{perform } ask\ ()) \bullet x \end{aligned} \tag{4}$$

f applies k under a reader handler h^{read_2}:

$$\begin{aligned} \longrightarrow^* & \overbrace{\text{prompt } m_0\ h^{read_2}}^{\langle\rangle} \bullet \overbrace{\text{prompt } m_2\ h^{evil}}^{w_4 = \langle read : (m_0,\ h^{read_2},\ \langle\rangle)\rangle} \bullet \overbrace{\text{prompt } m_3\ h^{tl}}^{w_5 = \langle evil : (m_1, h^{evil}, w_4),\ read : (m_0,\ h^{read_2},\ \langle\rangle)\rangle} \\ & \quad \underset{w_6 = \langle tl : (m_3,\ h^{tl}, w_5),\ evil : (m_2,\ h^{evil}, w_4)\ ,\ read : (m_0, h^{read_2}, \langle\rangle)\rangle}{\bullet} \overbrace{\text{under } tl}^{w_5} \bullet \overbrace{\text{perform } ask\ ()}^{w_5} \\ \longrightarrow^* & 2 \end{aligned} \tag{5}$$

The evaluation is quite subtle in several places. First, at (3) we introduced under *tl*. As shown at (4), the under frame can itself be captured by a resumption. This explains why we cannot directly apply the optimization but require an extra under frame: inside the resumption we still need to remember that operations happening after under *tl* can only reach handlers beyond h^{tl}.

However, at (3), it might be tempting to introduce the frame as under w_2 instead of *under tl*, as that would be enough to ensure that all operations afterwards are evaluated under w_2. By doing so, under could be formalized in a simpler way: under w_2 could simply ignore the current evidence vector and always pass w_2 to future operations. Our initial formalization did this but unfortunately this turns out to be unsound.

$$
\begin{array}{llll}
\text{Expression} & e & ::= & v \mid e\ e \mid e\ \sigma \\
 & & \mid & \text{prompt}\ m\ h\ e \\
 & & \mid & \text{yield}\ m\ v \\
\text{Value} & v & ::= & x \mid \lambda^{\epsilon} x\!:\!\sigma.\ e \mid \Lambda\alpha^{\kappa}.\ v \\
 & & \mid & \text{handler}\ h \\
 & & \mid & \text{perform}\ op\ \epsilon\ \overline{\sigma} \\
\text{Handler} & h & ::= & \{\ \overline{op_i \mapsto f_i}\ \} \\
\text{Evaluation ctx.} & \mathsf{E} & ::= & \square \mid \mathsf{E}\ e \mid v\ \mathsf{E} \mid \mathsf{E}\ \sigma \\
 & & \mid & \text{prompt}\ m\ h\ \mathsf{E} \\
 & \mathsf{F} & ::= & \square \mid \mathsf{F}\ e \mid v\ \mathsf{F} \mid \mathsf{F}\ \sigma
\end{array}
\qquad
\begin{array}{llll}
\text{Type} & \sigma & ::= & \alpha^{\kappa} \mid c^{\kappa}\ \overline{\sigma} \mid \sigma \to \epsilon\ \sigma \\
 & & \mid & \forall\alpha^{\kappa}.\ \sigma \\
\text{Effect row} & \epsilon & ::= & \langle\rangle \mid \langle l \mid \epsilon\rangle \mid \alpha^{\text{eff}} \\
\text{Kind} & \kappa & ::= & * \mid \kappa \to \kappa \mid \text{lab} \mid \text{eff} \\
\text{Type env.} & \Gamma & ::= & \varnothing \mid \Gamma,\ x\!:\!\sigma \\
\text{Effect ctx.} & \Sigma & ::= & \{\ \overline{l_i : sig_i}\ \} \\
\text{Effect sig.} & sig & ::= & \{\ \overline{op_i : \forall\overline{\alpha^{\kappa}}.\ \sigma_i \to \sigma_i'}\ \} \\
\text{Evidence} & ev & ::= & (m,\ h,\ w) \\
\text{Evidence vec.} & w & ::= & \langle\!\langle\rangle\!\rangle \mid \langle\!\langle l : ev \mid w\rangle\!\rangle
\end{array}
$$

$$
\begin{array}{lllll}
(app) & (\lambda^{\epsilon} x\!:\!\sigma.\ e)\ v & \longrightarrow & e[x{:=}v] & \\
(tapp) & (\Lambda\alpha^{\kappa}.\ v)\ \sigma & \longrightarrow & v[\alpha{:=}\sigma] & \\
(handler) & \text{handler}\ h\ v & \longrightarrow & \text{prompt}\ m\ h\ (v\ ()) & \text{with unique}\ m \\
(promptv) & \text{prompt}\ m\ h\ v & \longrightarrow & v & \\
(prompt) & \text{prompt}\ m\ h\ \mathsf{E}[\text{yield}\ m\ f] & \longrightarrow & f\ (\lambda^{\epsilon} x\!:\!\sigma_2.\ \text{prompt}\ m\ h\ \mathsf{E}[x]) & \\
 & & \text{with} & \varnothing \vdash_{\text{val}} f : (\sigma_2 \to \epsilon\ \sigma) \to \epsilon\ \sigma & \\
(perform) & w \vdash \text{perform}\ op\ \epsilon_0\ \overline{\sigma}\ v & \longrightarrow & \text{yield}\ m\ (\lambda^{\epsilon} k\!:\!\sigma_2[\overline{\alpha}{:=}\overline{\sigma}] \to \epsilon\ \sigma.\ f\ \overline{\sigma}\ v\ k) & \\
 & & \text{with} & (m, h, \_) = w.l \wedge (op \mapsto f) \in h & \\
 & & & (op : \forall\overline{\alpha}.\ \sigma_1 \to \sigma_2) \in \Sigma(l) \wedge \varnothing \vdash h : \sigma \mid l \mid \epsilon &
\end{array}
$$

$$
\frac{e \longrightarrow e'}{w \vdash \mathsf{F}[e] \longmapsto \mathsf{F}[e']}\,(step)
\qquad
\frac{w \vdash e \longrightarrow e'}{w \vdash \mathsf{F}[e] \longmapsto \mathsf{F}[e']}\,(stepw)
\qquad
\frac{}{w \vdash e \longmapsto^{*} e}
$$

$$
\frac{\langle\!\langle l : (m, h, w) \mid w\rangle\!\rangle \vdash e \longmapsto e'}{w \vdash \mathsf{F}[\text{prompt}\ m\ h\ e] \longmapsto \mathsf{F}[\text{prompt}\ m\ h\ e']}\,(promptw)
\qquad
\frac{w \vdash e \longmapsto^{*} e' \quad w \vdash e' \longmapsto e''}{w \vdash e \longmapsto^{*} e''}
$$

Fig. 1. F^{pw}: multi-prompt with evidence-passing semantics.

As shown at (5), the evidence vector for *under tl* itself has changed, from w_3 to w_6, and thus the evidence vector passed by *under tl* has also changed from w_2 to w_5, so that the last *ask* is handled by h^{read_2} and returns 2 (the reader can check that 2 is indeed the desired result of the program by evaluating without tail-resumptive optimization). If we would have used under w_2, the *ask* would wrongly return 1!

Proving the correctness of under is also challenging, as it essentially requires us to show that a program with tail-resumptive optimization will produce the same result as of the same program without the optimization. To this end, we show that the optimized program is *contextual equivalent* to the original program.

3 SEMANTICS

This section presents System F^{pw}, which features algebraic effects using multi-prompt and evidence passing semantics. The system is designed based on System F^{ϵ} [Xie et al. 2020], an explicitly typed polymorphic algebraic effect calculus.

3.1 Multi-Prompt with Evidence Passing Semantics

Syntax. Figure 1 defines the syntax. Expressions e include values v, applications $e\ e$, type applications $e\ \sigma$ and the internal frames prompt $m\ h$ and yield $m\ v$. Values include variables x, lambdas $\lambda^{\epsilon}\ x : \sigma.e$, which is annotated with the effect ϵ that may be performed when the lambda is applied, type lambdas $\Lambda\alpha^{k}.\ v$, and handler h and perform $op\ \epsilon\ \overline{\sigma}$. Since the calculus is explicitly typed and an operation signature can be polymorphic, performing an operation perform $op\ \epsilon\ \overline{\sigma}$ needs to indicate its context effect ϵ, as well as to explicitly pass the type arguments $\overline{\sigma}$. A handler h contains a list of operation clauses $op \mapsto f$, where f denotes a function expression. As we have seen before, an evaluation context E is essentially an expression with a hole in it, which indicates explicitly the evaluation order of an expression. A *pure* evaluation context F has no prompt frame.

Types. Types σ include type variables α^{κ} of kind κ, type constructors $c^{\kappa}\ \overline{\sigma}$ where c^{κ} of kind κ is applied to the arguments $\overline{\sigma}$, function types $\sigma \to \epsilon\ \sigma$ annotated with the effect ϵ that may be performed when the function is applied, and polymorphic types $\forall\alpha^{\kappa}.\ \sigma$. Types of kind eff are called *effect rows* and we write them as ϵ. Such row can be either empty $\langle\rangle$ (i.e. the type constructor $\langle\rangle^{\mathsf{eff}}$) which denotes the *total* effect, an extension $\langle l \mid \epsilon\rangle$ (i.e. the type constructor $\langle \_ \mid \_ \rangle^{\mathsf{lab} \to \mathsf{eff} \to \mathsf{eff}}$), which extends ϵ with effect label l (i.e. a type constructor l^{lab}), or a type variable α^{eff} (often written as μ). Effect rows that end with such a type variable (e.g., $\langle l \mid \mu\rangle$) are called *open*, while effect rows ending with an empty effect (e.g., $\langle l \mid \langle\rangle\rangle$) are called *closed*.

Equivalence between row types ($\equiv$) is defined as follows. Row equivalence is reflexive, transitive, and can freely reorder distinct labels.

$$\frac{}{\epsilon \equiv \epsilon} \qquad \frac{\epsilon_1 \equiv \epsilon_2 \quad \epsilon_2 \equiv \epsilon_3}{\epsilon_1 \equiv \epsilon_3} \qquad \frac{\epsilon_1 \equiv \epsilon_2}{\langle l \mid \epsilon_1\rangle \equiv \langle l \mid \epsilon_2\rangle} \qquad \frac{l_1 \neq l_2 \quad \epsilon_1 \equiv \epsilon_2}{\langle l_1, l_2 \mid \epsilon_1\rangle \equiv \langle l_2, l_1 \mid \epsilon_2\rangle}$$

To distinguish among types, System F^{pw} uses a basic kind system. Kinds κ include the basic kind ($*$), functions ($\kappa \to \kappa$), the kind of labels (lab), and the kind of effects (eff). The judgment $\vdash_{\mathsf{wf}}\ \sigma : \kappa$ checks the kind of types, whose definition is standard and is given in the technical report [Xie and Leijen 2021a].

The term context Γ is standard. A global effect context Σ maps each effect l to its signature sig^{l}, which gives every operation its input and output types, i.e., $op_i : \forall\overline{\alpha}_i.\ \sigma_i \to \sigma'_i$ (where $\overline{\alpha}_i = \mathsf{ftv}(\sigma_i \to \sigma'_i)$). We assume each op is uniquely named, and we use $op : \forall\overline{\alpha}.\ \sigma_1 \to \sigma_2 \in \Sigma(l)$ to denote the type of op that belongs to effect l.

Evidence Vectors. System F^{pw} incorporates *evidence passing semantics*. In particular, *evidence* ev is a triple consisting of a marker m, its corresponding handler h and the evidence vector w where h is defined. An *evidence vector* w is a map from effect labels to evidence. It can be either empty $\langle\!\langle\rangle\!\rangle$, or an extension $\langle\!\langle ev \mid w\rangle\!\rangle$ which extends w with evidence ev. We also write $\langle\!\langle w_1 \mid w_2\rangle\!\rangle$ for the concatenation of w_1 and w_2. We use the notation $w.l$ to select evidence of label l from w. As we have discussed in Section 2.5, we treat the evidence vector as an abstract datatype, as it can be either canonical or insertion ordered, depending on how the extension operation $\langle\!\langle ev \mid w\rangle\!\rangle$ is implemented. Importantly though, for correctness of the evidence passing semantics, selection and extension should satisfy the following laws, so that $w.l$ always finds the most recent evidence of l, which corresponds to the dynamic semantics of algebraic effects where an operation is handled by its innermost handler.

$$\langle\!\langle l : ev \mid w\rangle\!\rangle.l = ev \qquad \langle\!\langle l' : ev \mid w\rangle\!\rangle.l = w.l \ \text{ iff } l \neq l'$$

3.1.1 Operational Semantics. The operational semantics rules of System F^{pw} (Figure 1) include three definitions: $\longrightarrow$ provides a primitive evaluation step, $\longmapsto$ evaluates expressions under evaluation contexts, and $\longmapsto^{*}$ defines the *transitive closure* of $\longmapsto$. In practice, evaluating an expression always

start with an empty evidence vector. For clarify, we use a *lighter* color for all type information, which is needed for type soundness, but not directly for the dynamic semantics of algebraic effects.

($\longrightarrow$). During evaluation, we pass the current handlers down as an evidence vector. However, the evidence vector only matters when performing an operation, and many evaluation steps do not need to inspect the evidence vector. To make the difference clear, we separate the evaluation step into two categories: plain $e_1 \longrightarrow e_2$, and evaluation under an evidence vector $w \vdash e_1 \longrightarrow e_2$.

Rule (*app*) and (*tapp*) are standard. In rule (*handler*), handler installs a prompt m frame, with a fresh unique marker m, so that the marker can later be used to find the specific prompt. Values are propagated through the prompt frame (rule (*promptv*)).

As this system models the multi-prompt semantics, we split performing an operation into two parts: searching for a handler (rule (*perform*)), and capturing and restoring a resumption (rule (*prompt*)). Rule (*perform*) captures the essence of evidence passing semantics. Specifically, given the evidence vector w, performing an operation directly gets the handler h by selecting out the corresponding evidence by $w.l$. The operation implementation f from h is then used to handle the operation. As we will see shortly, the notation $\varnothing \vdash_{\mathsf{ops}} h : \sigma \mid l \mid \epsilon$ says that h is a handler for effect l, which has result type σ and may itself perform effect ϵ. Notice the difference between ϵ_0 and $\epsilon - \epsilon_0$ is the effect where perform is defined, and ϵ is the effect where prompt h is defined. Finally, in rule (*prompt*), yield captures the resumption $(\lambda^{\epsilon} x : \sigma_2.\ \mathsf{prompt}\ m\ h\ \mathsf{E}[x])$, to which f is applied.

($\longmapsto$). When evaluating expressions under evaluation contexts, each rule is given the current evidence vector w. Rule (*step*) and (*stepw*) correspond respectively to a plain $\longrightarrow$ and a $w \vdash \longrightarrow$ step. Both rules evaluate under an F. That is because as shown in rule (*promptw*), the prompt $m\ h\ e$ frame modifies the evidence vector by inserting the new evidence $l : (m, h, w)$ and uses the evidence vector $\langle\!\langle l : (m,\ h,\ w) \mid w \rangle\!\rangle$ for evaluating e. Here the evaluation context is again an F ensuring that the evidence vectors always match the prompt frames in the context.

Example. In Section 2, for better illustration, we have used the $\overset{w}{\frown}$ notation to indicate the current evidence vector. In the formal system, we always use $w \vdash$. The following example shows the evaluation derivation of handler $h^{read}\ (\lambda\_.\ \mathsf{perform}\ ask\ ()) \longmapsto^* 1$. We have omitted type annotations, and details regarding $\langle\!\langle\rangle\!\rangle \vdash e_1 \longmapsto e_3$ and $\langle\!\langle\rangle\!\rangle \vdash e_6 \longmapsto^* 1$.

(1) $e_1 = \mathsf{handler}\ h^{read}\ (\lambda\_.\ \mathsf{perform}\ ask\ ())$
(2) $e_2 = \mathsf{perform}\ ask\ ()$
(3) $e_3 = \mathsf{prompt}\ m\ h^{read}\ e_2$
(4) $e_4 = \mathsf{yield}\ m\ (\lambda k.\ (\lambda x\ k.\ 1)\ ()\ k)$
(5) $e_5 = \mathsf{prompt}\ m\ h^{read}\ e_4$
(6) $e_6 = (\lambda k.\ (\lambda x\ k.\ 1)\ ()\ k)\ (\lambda x.\ \mathsf{prompt}\ m\ h^{read}\ x)$

$$\dfrac{\dfrac{\dfrac{\dots}{\langle\!\langle\rangle\!\rangle \vdash e_1 \longmapsto^* e_3} \quad \dfrac{\langle\!\langle m,\ h^{read},\ \langle\!\langle\rangle\!\rangle\rangle\!\rangle \vdash e_2 \longrightarrow e_4}{\langle\!\langle\rangle\!\rangle \vdash e_3 \longmapsto e_5}}{\langle\!\langle\rangle\!\rangle \vdash e_1 \longmapsto^* e_5} \quad \dfrac{e_5 \longrightarrow e_6}{\langle\!\langle\rangle\!\rangle \vdash e_5 \longmapsto e_6}}{\langle\!\langle\rangle\!\rangle \vdash e_1 \longmapsto^* e_6} \dots}{\langle\!\langle\rangle\!\rangle \vdash e_1 \longmapsto^* 1}$$

3.1.2 Typing Rules. Figure 2 defines the typing rules for System F^{pw}. The judgment $\Gamma \vdash e : \sigma \mid \epsilon$ reads that, under the typing context Γ, the expression e has type σ and may perform effect ϵ. Values are not effectful and thus the typing judgment takes the form $\Gamma \vdash_{\mathsf{val}} v : \sigma$. The judgment $\Gamma \vdash_{\mathsf{ops}} h : \sigma \mid l \mid \epsilon$ type-checks a handler h for effect l, with result type σ and effect ϵ. For clarity of presentation we do not maintain an explicit kind environment for type variables; instead, as a well-formedness condition, we assume that all occurrences of a type variable α always have the same kind κ (subject to alpha-renaming).

Most rules are standard. Rule VAL can take in any effect. In rule ABS, the effect annotation from the lambda is passed to the body derivation, and the rule produces type $\sigma_1 \to \epsilon\ \sigma_2$, where ϵ indicates the effect that may be performed by the lambda body. In rule APP, we require three effects to match: the effect ϵ in the function $\sigma_1 \to \epsilon\ \sigma_2$, the effect ϵ of e_1 and of e_2. The rules TAPP and TABS handle

$$\boxed{\Gamma \vdash e : \sigma \mid \epsilon \qquad \Gamma \vdash_{\mathsf{val}} v : \sigma \qquad \Gamma \vdash_{\mathsf{ops}} h : \sigma \mid l \mid \epsilon}$$

$$\frac{x{:}\sigma \in \Gamma}{\Gamma \vdash_{\mathsf{val}} x : \sigma}\ \textsc{var} \qquad \frac{\Gamma \vdash_{\mathsf{val}} v : \sigma}{\Gamma \vdash v : \sigma \mid \epsilon}\ \textsc{val}$$

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

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

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

$$\frac{\Gamma \vdash_{\mathsf{ops}} h : \sigma \mid l \mid \epsilon}{\Gamma \vdash_{\mathsf{val}} \mathsf{handler}\ h : (() \to \langle l \mid \epsilon \rangle\ \sigma) \to \epsilon\ \sigma}\ \textsc{handler}$$

$$\frac{\Gamma \vdash_{\mathsf{ops}} h : \sigma \mid l \mid \epsilon \quad \Gamma \vdash e : \sigma \mid \langle l \mid \epsilon \rangle}{\Gamma \vdash \mathsf{prompt}\ m\ h\ e : \sigma \mid \epsilon}\ \textsc{prompt} \qquad \frac{\Gamma \vdash_{\mathsf{val}} f : (\sigma \to \epsilon'\ \sigma') \to \epsilon'\ \sigma'}{\Gamma \vdash \mathsf{yield}\ m\ f : \sigma \mid \epsilon}\ \textsc{yield}$$

$$\frac{op_i : \forall\overline{\alpha}.\, \sigma_1 \to \sigma_2 \in \Sigma(l) \quad \overline{\alpha} \not\cap \mathsf{ftv}(\epsilon, \sigma) \quad \Gamma \vdash_{\mathsf{val}} f_i : \forall\overline{\alpha}.\, \sigma_1 \to \epsilon\ ((\sigma_2 \to \epsilon\ \sigma) \to \epsilon\ \sigma)}{\Gamma \vdash_{\mathsf{ops}} \{\ op_1 \mapsto f_1,\ \ldots,\ op_n \mapsto f_n\ \} : \sigma \mid l \mid \epsilon}\ \textsc{ops}$$

Fig. 2. Typing Rules for System F^{pw}.

type application- and abstraction and are mostly standard except that type abstraction requires the kind is not lab: this ensures that lab types are always a constant (l) which ensures that unification for row equivalence is decidable [Leijen 2005].

Performing an operation introduces effects. In rule PERFORM, perform $op\ \epsilon\ \overline{\sigma}$ first gets the type of the operation from $\Sigma(l)$, and adds l to the context effect ϵ, generating $\langle l \mid \epsilon \rangle$. Dually, handling eliminates effects. In rule HANDLER, given a handler h for l, the rule takes an action with effect $\langle l \mid \epsilon \rangle$, and produces the result effect ϵ. Rule PROMPT is similar, but directly takes an expression e of effect $\langle l \mid \epsilon \rangle$. Rule OPS types a handler, where we assume $\{op_1, \ldots, op_n\} = \Sigma(l)$. Note that all operation implementations must have the same effect (ϵ) and type result (σ).

Rule YIELD is more subtle. Recall that the operational rule (*perform*) (in Figure 1) turns perform into yield, So we expect the result type of yield to match that of perform. Note that the result type of perform is the same as the argument type of the resumption k, and the type of the resumption k itself is the argument type of f in yield $m\ f$. Therefore, in rule YIELD, we directly get the result type from the type of f. To be more precise, we could also set the result effect of yield to match that of perform. But since yield is an internal frame, the current form is sufficient for type soundness.

3.1.3 Correctness, Preservation and Progress. In rule (*perform*), we refer to w as the current evidence vector, and we select out the handler from the evidence vector (instead of searching for it in the evaluation context). This means that for the correctness of evidence passing semantics, the current evidence vector w *must correspond exactly to the actual handlers in the dynamic evaluation context* – so that the handler selected from the evidence vector is indeed exactly the innermost handler that would be found with the original semantics of algebraic effects.

We use the notation $\lceil \mathsf{E} \rceil$ to extract all evidence from an evaluation context E. Specifically, if E is $\mathsf{F}_0 \bullet \mathsf{prompt}\ m_1\ h_1 \bullet \mathsf{F}_1 \bullet \ldots \bullet \mathsf{prompt}\ m_n\ h_n \bullet \mathsf{F}_n$, where each h_i is a handler for l_i, we have $\lceil \mathsf{E} \rceil = \langle\!\langle l_n : (m_n,\ h_n,\ \_) \mid \ldots \mid l_1 : (m_1,\ h_1, \_) \rangle\!\rangle$ (we ignore the third component as it is not used). In

order to prove correctness, we show that a $\longmapsto$ step can be reasoned in terms of a $\longrightarrow$ step, where for a $w \vdash \longrightarrow$ step, the evidence vector is the original evidence vector extended by all evidence from the evaluation context:

Lemma 1. (*Inversion of* $\longmapsto$). If $w \vdash e_1 \longmapsto e_2$, then either

- $e_1 = \mathsf{E}[e_1']$, $e_2 = \mathsf{E}[e_2']$, and $e_1' \longrightarrow e_2'$; or
- $e_1 = \mathsf{E}[e_1']$, $e_2 = \mathsf{E}[e_2']$, and $\langle\!\langle \lceil \mathsf{E} \rceil \mid w \rangle\!\rangle \vdash e_1' \longrightarrow e_2'$.

Based on Lemma 1, we can now show that the marker m and the handler h found by evidence-passing semantics is indeed the innermost handler found dynamically from the evaluation context. The following theorem establishes the correctness of evidence passing semantics.

Theorem 1. (*Evidence corresponds to the evaluation context*).

$$\text{If } \langle\!\langle\rangle\!\rangle \vdash \mathsf{E}[\mathsf{perform}\ op^l\ \overline{\sigma}\ v] \longmapsto \mathsf{E}[\mathsf{yield}\ m\ (\lambda k.\ f\ \overline{\sigma}\ v\ k)], \text{ then } \lceil \mathsf{E} \rceil.l = (m, h, \_), \text{and } (op \mapsto f) \in h.$$

Preservation and progress do not hold immediately for our system; instead we need to consider both prompt and yield as strictly *internal* frames that cannot be written directly by the programmer (and only occur during evaluation). For example, if we can write yield m ourselves, we can use an arbitrary m that does not match with any prompt in the context (and thus lose progress); similarly, we can write a yield $m\ f$ where the result type of f does not match the type expected by the prompt m in the context (and lose preservation).

By treating both prompt and yield as strictly internal frames we can ensure by construction that the previous problematic examples cannot occur, and can prove progress and preservation. In particular, we use a similar definition as the *handle-safe* definition from [Xie et al. 2020]:

Definition 1. (*Internal-safe expressions*). An *internal-safe* expression is a well-typed closed expression that either (1) contains no internal construct; or (2) is itself reduced from an internal-safe expression.

Internal-safe expressions maintain two important invariants: (1) each prompt owns a unique m generated at rule (*handler*); and (2) when perform generates yield m in (*perform*), it has found the handler with the right type (and therefore, yield m will find the right prompt m in rule (*prompt*)). We prove that internal-safe System F^{pw} enjoys preservation and progress.

Theorem 2. (*Preservation of Internal-safe System* F^{pw}). If $\varnothing \vdash e_1 : \sigma \mid \langle\rangle$ where e_1 is internal-safe, and $\langle\!\langle\rangle\!\rangle \vdash e_1 \longmapsto e_2$, then $\varnothing \vdash e_2 : \sigma \mid \langle\rangle$.

The progress theorem is more tricky, as perform does not find the handler from the evaluation context but instead from the evidence vector. Fortunately, from Lemma 1, we can show that the handler found from the evidence vector is always available in the evaluation context.

Theorem 3. (*Progress of Internal-safe System* F^{pw}). If $\varnothing \vdash e_1 : \sigma \mid \langle\rangle$ where e_1 is internal-safe, then either e_1 is a value, or $\langle\!\langle\rangle\!\rangle \vdash e_1 \longmapsto e_2$.

We can further prove that markers cannot be duplicated in the evaluation context.

Theorem 4. (*Uniqueness of Handlers for Internal-safe System* F^{pw}). For any internal-safe F^{pw} expression prompt $m_1\ h_1\ (\mathsf{E}_2[\mathsf{prompt}\ m_2\ h_2\ e])$, we have $m_1 \neq m_2$.

3.2 Tail-Resumptive Optimization

With evidence passing semantics, we are now ready to formalize the tail-resumptive optimization, which is given in Figure 3. We extend the definition of expressions with $\mathsf{under}^{\epsilon,\epsilon}\ l\ e$, and the definition of evaluation contexts with $\mathsf{under}^{\epsilon,\epsilon}\ l\ \mathsf{E}$.

$$
\begin{array}{llll}
\text{Expression} & e & ::= & \ldots \mid \mathsf{under}^{\epsilon,\epsilon}\ l\ e \\
\text{Evaluation context} & \mathsf{E} & ::= & \ldots \mid \mathsf{under}^{\epsilon,\epsilon}\ l\ \mathsf{E}
\end{array}
$$

$$
\begin{array}{llll}
(\mathit{performt}) & w \vdash \mathsf{perform}\ \mathit{op}\ \epsilon_0\ \overline{\sigma}\ v & \longrightarrow & (\Lambda\overline{\alpha}.\ \lambda^{\langle l \mid \epsilon_0\rangle} x : \sigma_1.\ \mathsf{under}^{\epsilon_0,\epsilon}\ l\ e)\ \overline{\sigma}\ v \\
 & & \text{with} & (m, h, w') = w.l \\
 & & & (\mathsf{op} \mapsto \Lambda\overline{\alpha}.\ \lambda^{\epsilon} x : \sigma_1.\ k : \sigma_2 \to \epsilon\ \sigma.\ k\ e) \in h \ \wedge\ k \notin \mathsf{fv}(e) \\
(\mathit{underv}) & \mathsf{under}^{\epsilon_0,\epsilon}\ l\ v & \longrightarrow & v
\end{array}
$$

$$
\frac{w' \vdash e \longmapsto e' \quad (m, h, w') = w.l}{w \vdash \mathsf{F}[\mathsf{under}^{\epsilon_0,\epsilon}\ l\ e] \longmapsto \mathsf{F}[\mathsf{under}^{\epsilon_0,\epsilon}\ l\ e']}\ (\mathit{underw}) \qquad \frac{\Gamma \vdash e : \sigma \mid \epsilon}{\Gamma \vdash \mathsf{under}^{\epsilon_0,\epsilon}\ l\ e : \sigma \mid \langle l \mid \epsilon_0\rangle}\ \textsc{under}
$$

Fig. 3. Tail resumptive operations

3.2.1 Operational Semantics. Rule (*performt*) is the key to apply the tail-resumptive optimization. First, it gets the handler h from the evidence vector as before. But it then detects that the operation implementation ($\Lambda\overline{\alpha}.\ \lambda^{\epsilon} x : \sigma_1.\ \lambda.k : \sigma_2 \to \epsilon\sigma.\ k\ e$) is tail-resumptive (with $k \notin \mathsf{fv}(e)$), and so instead of yielding up, it generates $\mathsf{under}^{\epsilon_0,\epsilon}\ l\ e$, which directly evaluates e *in-place* with the type arguments $\overline{\sigma}$ and value argument v. When the expression evaluates to a value, the value is propagated through the under frame (rule (*underv*)).

Importantly, under needs to modify the evidence vector, so that operations happening after it can find the right handler. In rule (*underw*), given the current evidence vector w, under first finds the innermost evidence for l in the evidence vector, i.e., $(m,\ h,\ w')$, and then passes the evidence vector w', under which h is defined, to e. In other words, under skips the whole evidence fragment between h to itself, which should not be accessible to e.

3.2.2 Typing. Rule UNDER types an $\mathsf{under}^{\epsilon_0,\epsilon}\ l\ e$ expression. Note that the effect ϵ corresponds to the effect of e, while under itself produces $\langle l \mid \epsilon_0\rangle$. As with yield, in a more refined system, we can further state that ϵ_0 contains ϵ (as when generated in (*performt*)), but as under is internal, the current typing rule is sufficient for establishing soundness.

3.2.3 Correctness, Preservation and Progress. In what sense is the tail-resumptive optimization correct? Only if the optimized expression can produce an equivalent result as of the original expression. However, the equivalence is not so obvious. To illustrate the subtlety, consider evaluating the expression $(\mathsf{prompt}\ m\ h \bullet \mathsf{E} \bullet \mathsf{perform}\ \mathit{op}\ \overline{\sigma}\ v)$ under the evidence vector w. Assume that the op operation is handled by $\mathsf{prompt}\ m\ h$, where $(\mathsf{op} \mapsto \Lambda\overline{\alpha}.\ \lambda x\ k.\ k\ e) \in h$ with $k \notin \mathsf{fv}(e)$, i.e., the implementation is tail-resumptive. If we evaluate the expression without tail-resumptive optimization, we get (for clarify we omit w in the derivation):

$$
\begin{array}{ll}
 & \mathsf{prompt}\ m\ h \bullet \mathsf{E} \bullet \mathsf{perform}\ \mathit{op}\ \epsilon_0\ \overline{\sigma}\ v \\
\longmapsto & \mathsf{prompt}\ m\ h \bullet \mathsf{E} \bullet \mathsf{yield}\ m\ (\lambda k.\ (\Lambda\overline{\alpha}.\ \lambda x\ k.\ k\ e)\ \overline{\sigma}\ v\ k) \\
\longmapsto^* & (\Lambda\overline{\alpha}.\ \lambda x\ k.\ k\ e)\ \overline{\sigma}\ v\ (\lambda x.\ \mathsf{prompt}\ m\ h\ \mathsf{E}[x]) \\
\longmapsto^* & (\lambda x.\ \mathsf{prompt}\ m\ h\ \mathsf{E}[x])\ e[\overline{\alpha} := \overline{\sigma}, x := v]
\end{array}
$$

while with tail optimization we end up with:

$$
\begin{array}{ll}
 & \mathsf{prompt}\ m\ h \bullet \mathsf{E} \bullet \mathsf{perform}\ \mathit{op}^l\ \epsilon_0\ \overline{\sigma}\ v \\
\longmapsto & \mathsf{prompt}\ m\ h \bullet \mathsf{E} \bullet (\Lambda\overline{\alpha}.\ \lambda x.\ \mathsf{under}^{\epsilon_0,\epsilon}\ l \bullet e)\ [\overline{\sigma}]\ v \\
\longmapsto^* & \mathsf{prompt}\ m\ h \bullet \mathsf{E} \bullet \mathsf{under}^{\epsilon_0,\epsilon}\ l \bullet e[\overline{\alpha} := \overline{\sigma},\ x := v]
\end{array}
$$

The two expressions are now quite different. Nevertheless, intuitively these two result expressions are *equivalent*: they both first evaluate $e[\overline{\alpha} := \overline{\sigma},\ x := v]$, and then pass the result to $\mathsf{prompt}\ m\ h\ \mathsf{E}$, via beta-reduction and via propagation through under, respectively. The situation is a bit more tricky though as $e[\overline{\alpha} := \overline{\sigma},\ x := v]$ may perform an operation. However, even in that case, the operation will

find the same handler: in the first case, it is obvious that the evidence vector passed to $e[\overline{\alpha}:=\overline{\sigma},\ x:=v]$ is w; in the second case, w is first extended by evidence from prompt $m\ h\ \bullet\ \mathsf{E}$, but then $\mathsf{under}^{\epsilon}\ l$ changes the evidence vector back to w! To capture the observation, we formalize an equivalent relation $e_1 \cong e_2$ between expressions (and evaluation contexts respectively) where e_1 has no under, and e_2 may have under. The relation $\cong$ is mostly structural, up to renaming of fresh markers, with the following rule:

$$\frac{e_1 \cong e_2 \quad \mathsf{E}_1 \cong \mathsf{E}_2 \quad l \notin \mathsf{bl}(\mathsf{E}_1) \quad \varnothing \vdash_{\mathsf{ops}} h : \sigma \mid l \mid \epsilon}{(\lambda x.\ \mathsf{prompt}\ m\ h\ \mathsf{E}_1[x])\ e_1 \cong \mathsf{prompt}\ m\ h\ \bullet\ \mathsf{E}_2\ \bullet\ \mathsf{under}^{\epsilon_0,\epsilon}\ l\ e_2}$$

We can prove that evaluation preserves the equivalent relation, except that expressions need to take several reduction steps to become equivalent again, as evaluating prompt under the two semantics takes different number of steps to reach the desired equivalent form.

Lemma 2. (*Evaluation Preserves* $\cong$). Given two closed internal-safe expressions $\varnothing \vdash e_1 : \sigma \mid \langle\rangle$ and $\varnothing \vdash e_2 : \sigma \mid \langle\rangle$, if $e_1 \cong e_2$, then either e_1 and e_2 are values, or there exist e_1', e_2' such that $\langle\!\langle\rangle\!\rangle \vdash e_1 \longmapsto^+ e_1'$, $\langle\!\langle\rangle\!\rangle \vdash e_2 \longmapsto^+ e_2'$, and $e_1' \cong e_2'$.

Based on Lemma 2, we show that the optimized and unoptimized expressions are *contextual equivalent*, with the intuition that we cannot tell them apart in any context.

Definition 2. (*Contextual Equivalence*).

$$\begin{aligned} e_1 \cong_{ctx} e_2 \triangleq\ & \varnothing \vdash e_1 : \sigma \mid \epsilon \wedge \varnothing \vdash e_2 : \sigma \mid \epsilon \\ & \wedge \forall C.\ \varnothing \vdash C : (\sigma \mid \epsilon) \to (\mathit{Int} \mid \langle\rangle) \implies (\forall n.\ C[e_1] \longmapsto^* n \iff C[e_2] \longmapsto^* n) \end{aligned}$$

where C is the standard definition of a *program context* that is under-free, and $C[e_1]$ is evaluated under the original semantics while $C[e_2]$ is with tail-resumptive optimization. The notation $\varnothing \vdash C : (\sigma_1 \mid \epsilon_1) \to (\sigma_2 \mid \epsilon_2)$ type-checks an program context, so that if $\varnothing \vdash e : \sigma_1 \mid \epsilon_1$, we have $\varnothing \vdash C[e] : \sigma_2 \mid \epsilon_2$.

Theorem 5. (*Tail-resumptive Optimization is Sound*). If $\varnothing \vdash e : \sigma \mid \epsilon$, then $e \cong_{ctx} e$.

The theorem may seem trivial, but given that $\cong_{ctx}$ uses different evaluation strategies for the left expression and the right one, the theorem states exactly what we want: starting from the same expression e, evaluating without and with tail-resumptive optimization produces the same result. We have also proved that Theorem 2 (Preservation) and Theorem 3 (Progress) remain valid for internal-safe System F^{pw} extended with under.

4 TRANSLATION TO POLYMORPHIC LAMBDA CALCULUS

In order to compile to standard lambda calculus from our evidence passing effect handler calculus, we first need to ensure that all transitions are *local* and no longer manipulate evaluation contexts explicitly. The only operation that it is non-local with evidence passing semantics is the yield. As discussed in Section 2.7 we can make this local by bubbling up the yields step-by-step through the context while constructing a resumption.

4.1 Bubbling Yields

We briefly introduce System F^{pb} (Figure 4), which extends the semantics of F^{pw} where yield builds the resumption locally and bubbles up to its corresponding prompt frame. In this section, we focus on the dynamic semantics of System F^{pb}, with its full typed formalization and preservation and progress theorems given in the technical report [Xie and Leijen 2021a].

First, expressions now include a new form of yielding expression $\mathsf{yield}\ m\ v\ v$ that takes an extra argument: the first v is a function that will be applied to the resumption (like before), while the second v is the current resumption that is extended step-by-step while bubbling up. We replace the

$$
\begin{array}{lllll}
\text{Expressions} & e & ::= & v \mid e\ e \mid e\ \sigma \mid \mathsf{prompt}\ m\ h\ e \mid \mathsf{yield}\ m\ v\ v & \\
(app_1) & v\ \square \bullet \mathsf{yield}\ m\ f\ k & \longrightarrow & \mathsf{yield}\ m\ f\ (\lambda x.\ v\ (k\ x)) & \\
(app_2) & \square\ e \bullet \mathsf{yield}\ m\ f\ k & \longrightarrow & \mathsf{yield}\ m\ f\ (\lambda x.\ (k\ x)\ e) & \\
(under) & \mathsf{under}\ l\ \square \bullet \mathsf{yield}\ m\ f\ k & \longrightarrow & \mathsf{yield}\ m\ f\ (\lambda x.\ \mathsf{under}\ l\ (k\ x)) & \\
(prompt_1) & \mathsf{prompt}\ m\ h\ \square \bullet \mathsf{yield}\ m\ f\ k & \longrightarrow & f\ (\lambda x.\ \mathsf{prompt}\ m\ h\ (k\ x)) & \\
(prompt_2) & \mathsf{prompt}\ n\ h\ \square \bullet \mathsf{yield}\ m\ f\ k & \longrightarrow & \mathsf{yield}\ m\ f\ (\lambda x.\ \mathsf{prompt}\ n\ h\ (k\ x)) & \text{iff } n \neq m \\
(perform) & w \vdash \mathsf{perform}\ op\ \epsilon_0\ \overline{\sigma}\ v & \longrightarrow & \mathsf{yield}\ m\ (\lambda k.\ f\ \overline{\sigma}\ v\ k)\ (\lambda x.\ x) & \\
 & & & \text{with}\ (m, h, \_) = w.l \wedge (op \mapsto f) \in h &
\end{array}
$$

Fig. 4. F^{pb}: Multi-prompt with bubble semantics.

original rules (*prompt*) and (*perform*) in System F^{pw} (Figure 1) with rules in Figure 4. The new rule (*perform*) builds the continuation and initial resumption, which is then bubbled up by the other rules. In (*perform*) the yield now gets an extra argument $(\lambda x.\ x)$ which is the initial *partially built* resumption – at this time just an identity function. The resumption is now gradually extended as yield bubbles up through every evaluation frame, as in rule (app_1), (app_2) and $(prompt_2)$. In rule (app_1), the frame $v\ \square$ is added to the current partially built resumption k, generating $(\lambda x.\ v\ (k\ x))$. Rule (app_2) is similar. Rule $(prompt_2)$ compares markers and finds that $n \neq m$ and adds the prompt frame to the resumption. The yield keeps bubbling up until it finds its matching handler in rule $(prompt_1)$, where we finally apply the continuation f to now completed resumption.

4.2 A Multi-prompt Delimited Control Monad

All transitions in the bubbling semantics are now *local* transitions, and we can implement these semantics using a multi-prompt delimited control monad, where each algebraic effect specific construct can be implemented directly as a regular function. In this section, we first establish the multi-prompt delimited control monad and then discuss the type directed translation from System F^{pb} into a polymorphic lambda calculus. We use standard techniques [Dybvig et al. 2007] to implement delimited control as a monad. For better readability, throughout this section we use Haskell-like syntax. First, we define our monad Mon as:

$$\mathsf{type}\ \mathsf{Mon}\ \mu\ \alpha\ =\ \mathsf{Evv}\ \mu \to \mathsf{Ctl}\ \mu\ \alpha$$

The evidence-passing semantics is established by taking an argument of type Evv μ, which corresponds to the current evidence vector for an effect μ, and returning in the control monad Ctl. The control monad is defined as[4]:

$$
\begin{array}{l}
\mathsf{data}\ \mathsf{Ctl}\ \mu\ \alpha\ = \\
\quad \mid \mathsf{Pure} : \alpha \to \mathsf{Ctl}\ \mu\ \alpha \\
\quad \mid \mathsf{Yield} : \forall \beta\ \mu'\ r.\ \mathsf{Marker}\ \mu'\ r \to ((\beta \to \mathsf{Mon}\ \mu'\ r) \to \mathsf{Mon}\ \mu'\ r) \to (\beta \to \mathsf{Mon}\ \mu\ \alpha) \to \mathsf{Ctl}\ \mu\ \alpha
\end{array}
$$

The Pure case returns a value result, while Yield implements yielding to a prompt (corresponding to yield $m\ f\ k$ in System F^{pb}). Markers carry explicit types and can later serve as the witness to type equality. When binding a yield, the resumption keeps being extended:

$$
\begin{array}{lll}
(f \circ g)\ x = f\ (g\ x) & & \text{(function composition)} \\
(f \star g)\ x = g\ x \rhd f & & \text{(Kleisli composition)} \\
e \rhd g = \lambda w.\ \mathsf{case}\ e\ w\ \mathsf{of}\ \mathsf{Pure}\ x \to g\ x\ w & & \text{(monadic bind)} \\
\qquad\qquad \mathsf{Yield}\ m\ f\ k \to \mathsf{Yield}\ m\ f\ (g \star k) & & ((app_1), (app_2)\ \text{Fig. 4, APP Fig. 5})
\end{array}
$$

[4]This monad is used exactly in the *Mp.Eff* library [Xie and Leijen 2021b], but the Ctl is different from that of *Ev.Eff* [Xie et al. 2020] as the continuation and resumption in Yield both return in Mon, whereas in *Ev.Eff* these return in Ctl (again because in EPT the evidence vector is statically determined).

With the multi-prompt monad, we can now define the monadic translation of types from System F^{pb}, where all effectful functions are made monadic:

$$\begin{array}{lll} \lfloor \forall \overline{\alpha}^{\overline{\kappa}}.\, \sigma \rfloor = \forall \overline{\alpha}^{\overline{\kappa}}.\, \lfloor \sigma \rfloor & & \lfloor \sigma_1 \to \epsilon\ \sigma_2 \rfloor = \lfloor \sigma_1 \rfloor \to \mathsf{Mon}\ \epsilon\ \lfloor \sigma_2 \rfloor \\ \lfloor \alpha \rfloor = \alpha & & \lfloor c^{\kappa}\ \sigma_1 \ldots \sigma_n \rfloor = c^{\kappa}\ \lfloor \sigma_1 \rfloor \ldots \lfloor \sigma_n \rfloor \end{array}$$

We then implement prompt as a family of $prompt^l$ functions for each effect l:

$$\begin{array}{l} prompt^l : \forall \mu\ \alpha.\, \mathsf{Marker}\ \mu\ \alpha \to \mathsf{Hnd}^l\ \mu\ \alpha \to \mathsf{Mon}\ \langle l \mid \mu \rangle\ \alpha \to \mathsf{Mon}\ \mu\ \alpha \\ prompt^l\ m\ h\ e = \lambda w.\, \mathsf{case}\ e\ \langle\!\langle l : (m, h, w) \mid w \rangle\!\rangle\ \mathsf{of} \\ \quad \mathsf{Pure}\ x \to \mathsf{Pure}\ x \qquad ((promptv)\ \text{in Fig. 1}) \\ \quad \mathsf{Yield}\ m'\ f\ k \mid m \neq m' \to \mathsf{Yield}\ m'\ f\ (prompt^l\ m\ h \circ k) \qquad ((prompt_2)\ \text{in Fig. 4}) \\ \quad \mathsf{Yield}\ m'\ f\ k \mid m = m' \to f\ (prompt^l\ m\ h \circ k)\ w \qquad ((prompt_1)\ \text{in Fig. 4}) \end{array}$$

Note how the evidence vector is passed as an explicit argument in the monad. The Pure case returns the value as is. For Yield, if it yields to another prompt, we keep building the resumption. In the third case, Yield meets the target prompt and we apply f to the built-up resumption (composed with $prompt^l\ m\ h$ as we use *deep* resumptions). Note that to type check this case, the equality of the markers $m = m'$ implies that $\mu = \mu'$ and $\alpha = r$ (as in the definition of Yield). For example, this can be encoded using explicit equality witnesses [Baars and Swierstra 2002] or equality constraints in Haskell [Sulzmann et al. 2007; Xie and Leijen 2021b].

The *handler* function generates *prompt* with a fresh marker created by a utility function *freshm*.

$$\begin{array}{l} handler^l : \forall \mu\ \alpha.\, \mathsf{Hnd}^l\ \mu\ \alpha \to ((\,) \to \mathsf{Mon}\ \langle l \mid \mu \rangle\ \alpha) \to \mathsf{Mon}\ \mu\ \alpha \\ handler^l\ h\ f = freshm\ (\lambda m \to prompt^l\ m\ h\ (f\ (\,))) \qquad ((handler)\ \text{in Fig. 1}) \end{array}$$

The type of a handler Hnd^l is generated for every effect signature $l : \{ op_1 : \forall \overline{\alpha}_1.\, \sigma_1 \to \sigma_1',\ \ldots,\ op_n : \forall \overline{\alpha}_n.\, \sigma_n \to \sigma_n' \} \in \Sigma$ and is a record of operation definitions:

$$\mathsf{data}\ \mathsf{Hnd}^l\ \mu\ r = \mathsf{Hnd}^l\ (\forall \overline{\alpha}_1.\, \mathsf{Op}\ \lfloor \sigma_1 \rfloor\ \lfloor \sigma_1' \rfloor\ \mu\ r) \ldots (\forall \overline{\alpha}_n.\, \mathsf{Op}\ \lfloor \sigma_n \rfloor\ \lfloor \sigma_n' \rfloor\ \mu\ r)$$

together with a selector for each operation $op_i : \forall \overline{\alpha}.\, \sigma_1 \to \sigma_2 \in \Sigma(l)$:

$$\begin{array}{l} select^{op_i} : \forall \overline{\alpha}\ \mu\ r.\, \mathsf{Hnd}^l\ \mu\ r \to \mathsf{Op}\ \lfloor \sigma_1 \rfloor\ \lfloor \sigma_2 \rfloor\ \mu\ r \\ select^{op_i}\ (\mathsf{Hnd}^l\ op_1 \ldots op_n) = op_i \end{array}$$

where the $\mathsf{Op}\ \alpha\ \beta\ \mu\ r$ type represents operations from α to β defined in a handler with effect μ and result type r (the *answer* type). For example for a reader effect we have:

$$\begin{array}{l} \mathsf{data}\ \mathsf{Hnd}^{read}\ \mu\ r = \mathsf{Hnd}^{read}\ (\mathsf{Op}\ (\,)\ \mathsf{int}\ \mu\ r) \\ select^{ask}\ (\mathsf{Hnd}^{read}\ ask) = ask \end{array}$$

For operations we distinguish between tail-resumptive operation implementations and normal implementations in order to do the tail-resumptive optimization:

$$\begin{array}{l} \mathsf{data}\ \mathsf{Op}\ \alpha\ \beta\ \mu\ r = \mathsf{Tail} : (\alpha \to \mathsf{Mon}\ \mu\ \beta) \to \mathsf{Op}\ \alpha\ \beta\ \mu\ r \\ \qquad\qquad \mid \mathsf{Normal} : (\alpha \to \mathsf{Mon}\ \mu\ ((\beta \to \mathsf{Mon}\ \mu\ r) \to \mathsf{Mon}\ \mu\ r)) \to \mathsf{Op}\ \alpha\ \beta\ \mu\ r \end{array}$$

We can now *perform* an operation by getting the handler from the evidence vector, and selecting the right operation from the handler record (e.g. ask). Depending on the operation constructor, we use $under^l$ for tail-resumptive operations or otherwise generate a Yield.

$$\begin{array}{l} perform^l : \forall \mu\ \alpha\ \beta.\, (\forall \mu'\ r.\, \mathsf{Hnd}^l\ \mu'\ r \to \mathsf{Op}\ \alpha\ \beta\ \mu'\ r) \to \alpha \to \mathsf{Mon}\ \langle l \mid \mu \rangle\ \beta \\ perform^l\ select\ x = \lambda w : \mathsf{Evv}\ \langle l \mid \mu \rangle.\, \mathsf{let}\ (m, h, w') = w.l\ \mathsf{in} \\ \quad \mathsf{case}\ select\ h\ \mathsf{of}\ \mathsf{Tail}\ f \to under^l\ m\ w'\ (f\ x) \qquad ((performt)\ \text{in Fig. 3}) \\ \qquad\qquad \mathsf{Normal}\ f \to \mathsf{Yield}\ m\ (\lambda y.\, f\ x \rhd (\lambda g.\, g\ y))\ (\lambda x\ w.\, \mathsf{Pure}\ x) \qquad ((perform)\ \text{in Fig. 4}) \end{array}$$

$$\textsc{val}\quad \frac{\Gamma \vdash_{\mathsf{val}} v : \sigma \rightsquigarrow v'}{\Gamma \vdash v : \sigma \mid \epsilon \rightsquigarrow \lambda w : \mathsf{Evv}\ \epsilon.\mathsf{Pure}\ \epsilon\ \lfloor\sigma\rfloor\ v'}$$

$$\textsc{app}\quad \frac{\Gamma \vdash e_1 : \sigma_1 \to \epsilon\ \sigma \mid \epsilon \rightsquigarrow e_1' \quad \Gamma \vdash e_2 : \sigma_1 \mid \epsilon \rightsquigarrow e_2'}{\Gamma \vdash e_1\ e_2 : \sigma \mid \epsilon \rightsquigarrow e_1' \rhd (\lambda f : \lfloor \sigma_1 \to \epsilon\ \sigma \rfloor.\ e_2' \rhd f)}$$

$$\textsc{perform}\quad \frac{op : \forall \overline{\alpha}.\ \sigma_1 \to \sigma_2 \in \Sigma(l)}{\Gamma \vdash_{\mathsf{val}} \mathsf{perform}\ op\ \epsilon\ \overline{\sigma} : \sigma_1[\overline{\alpha}{:=}\overline{\sigma}] \to \langle l \mid \epsilon \rangle\ \sigma_2[\overline{\alpha}{:=}\overline{\sigma}] \rightsquigarrow perform^l\ \epsilon\ \lfloor\sigma_1[\overline{\alpha}{:=}\overline{\sigma}]\rfloor\ \lfloor\sigma_2[\overline{\alpha}{:=}\overline{\sigma}]\rfloor\ (\Lambda\mu\ r.\ select^{op}\ \lfloor\overline{\sigma}\rfloor\ \mu\ r)}$$

Fig. 5. Monadic translation of F^{pb} (excerpt).

Finally, *under* can be implemented with two mutually recursive definitions:

$$under^l : \forall \mu\ \beta\ \mu'\ r.\ \mathsf{Marker}\ \mu'\ r \to \mathsf{Evv}\ \mu' \to \mathsf{Mon}\ \mu'\ \beta \to \mathsf{Mon}\ \mu\ \beta$$
$$underk^l : \forall \mu\ \beta\ \mu'\ r.\ \mathsf{Marker}\ \mu'\ r \to (\beta \to \mathsf{Mon}\ \mu'\ r) \to \beta \to \mathsf{Mon}\ \mu\ r$$

The *under* function runs the action e under another evidence vector w', and ensures that any resumption is itself continued under the right evidence through *underk*:

$$\begin{aligned} under^l\ m\ w'\ e\ =\ & \lambda w : \mathsf{Evv}\ \mu.\ \mathsf{case}\ e\ w'\ \mathsf{of} && ((underw)\ \text{in Fig. 3})\\ & \mathsf{Pure}\ x \quad \to \mathsf{Pure}\ x && ((underv)\ \text{in Fig. 3})\\ & \mathsf{Yield}\ n\ f\ k \to \mathsf{Yield}\ n\ f\ (underk^l\ m\ k) && ((under)\ \text{in Fig. 4}) \end{aligned}$$

Note that it is easy to make a mistake here: in the Yield case, a well-typed (!) but semantically wrong implementation of $under^l$ is to return $\mathsf{Yield}\ n\ f\ (\lambda x.\ under^l\ m\ w'\ (k\ x))$ – as described in Section 2.12.2 this wrongly fixes the evidence vector to w'. Instead, we need to use the *underk* function which re-finds the correct evidence vector w' from the current evidence vector w to resume under:

$$\begin{aligned} underk^l\ m\ k\ x\ =\ & \lambda w : \mathsf{Evv}\ \mu.\ \mathsf{let}\ (m', h, w' : \mathsf{evv}\ \epsilon)\ =\ w.l\ \mathsf{in} \\ & \quad \mathsf{if}\ (m\ =\ m')\ \mathsf{then}\ under\ m\ w'\ (k\ x)\ w && ((underw)\ \text{in Fig. 3}) \end{aligned}$$

The marker is passed to under^l and underk^l in order to get the type equality from $m\ =\ m'$ (which should always hold for internal-safe expressions).

4.3 Monadic Translation

Using the multi-prompt monad definition, we can define a type-directed translation of System F^{pb} into a polymorphic lambda calculus (see the technical report [Xie and Leijen 2021a]). The translation takes the form $\Gamma\ \vdash\ e\ :\ \sigma \mid \epsilon\ \rightsquigarrow e'$, where under Γ, the expression e with type σ and effect ϵ is translated to e'. Based on the multi-prompt monad, the translation is mostly straightforward where each construct translates directly to its corresponding function: prompt translates to *prompt*, handler translates to *handler*, etc. Figure 5 shows an excerpt of the translation rules, while the full translation is shown in the technical report [Xie and Leijen 2021a].

During translation, we have made type applications explicit. Rule VAL simply wraps the value translation inside Pure. Rule APP first evaluates e_1', binds the result to f, and then evaluates e_2' and passes the result to f. If any of the expressions evaluates to Yield, the monadic binding ($\rhd$) will bubble it up (according to the rules (app_1) and (app_2) in Figure 4). Rule PERFORM shows how perform is translated using our monadic implementation of $perform^l$ and $select^{op}$.

We prove that our translation is sound, where we use the notation $\vdash_{\mathsf{F}}$ for the typing judgment in the target polymorphic lambda calculus, whose full definition can be found in the technical report [Xie and Leijen 2021a].

Theorem 6. (*Monadic Translation is Sound*). If $\varnothing\ \vdash\ e\ :\ \sigma \mid \langle\rangle\ \rightsquigarrow e'$, then $\varnothing \vdash_{\mathsf{F}}\ e'\ :\ \mathsf{Mon}\ \langle\rangle\ \lfloor\sigma\rfloor$.

4.4 Semantics Preserving

We now show that our sequence of refinements preserve the *original semantics* of polymorphic algebraic effect handlers in System F^{ϵ} [Xie et al. 2020]. In particular, consider a user-provided expression e in F^{ϵ}. As our initial multi-prompt delimited control semantics shares the same *source* language (i.e., without *internal* frames) with System F^{ϵ}, we have two possible dynamic semantics for e: (1) the original direct semantics defined in System F^{ϵ}; and (2) the multi-prompt delimited control semantics described in 2.4. We can prove that these two semantics always give the same result; that is, the multi-prompt delimited control preserves the original algebraic effects semantics.

In fact, each of our further refinement steps is also semantics preserving: (1) the evidence passing semantics preserves the multi-prompt delimited control semantics; (2) the bubbling semantics F^{pb} preserves the evidence passing semantics; and (3) the monadic translation semantics preserves the bubbling semantics. Detailed lemma statements and their proofs are included in in the technical report [Xie and Leijen 2021a].

Building on top of the semantics preserving lemmas of each refinement step, we can show that the final monadic translation preserves the semantics of System F^{ϵ}. Specifically, for a user-provided total expression e of type int, if e evaluates to n in System F^{ϵ}, then its monadic translation evaluates to n in the polymorphic lambda calculus; we use $e \Uparrow$ to denote the case when e diverges.

Theorem 7. (*Semantics Preserving*). Given $\varnothing \vdash e : int \mid \langle\rangle \rightsquigarrow e'$, if $e \longmapsto^* n$ in F^{ϵ}, then $e' \langle\!\langle\rangle\!\rangle \longmapsto^*$ Pure $\langle\rangle$ int n, in the polymorphic lambda calculus and if $e \Uparrow$ in F^{ϵ}, then $e' \langle\!\langle\rangle\!\rangle \Uparrow$ in the polymorphic lambda calculus.

5 BENCHMARKS

In this section we benchmark five implementations of effect handlers [Leijen 2021].

(1) *Koka*: We have a full implementation of our techniques in the Koka compiler [Leijen 2020] which compiles via standard C code. This uses generalized evidence passing with canonical evidence vectors, short-cut resumptions, bind-inlining and join-point sharing.
(2) *multi-core OCaml*: This is a fork of standard OCaml with the current state-of-the-art *direct* implementation of effect handlers based on segmented stacks [Sivaramakrishnan et al. 2021] (but without direct support for multi-shot resumptions).
(3) *Mp.Eff*: This is our implementation of generalized evidence passing effect handlers as a monadic library in Haskell [Xie and Leijen 2021b]. The library uses insertion-ordered evidence vectors and does not use short-cut resumptions.
(4) *Ev.Eff*: A Haskell monadic effect handler library by Xie and Leijen [2020] based on evidence *translation* (and cannot handle non-scoped resumptions). This library performs very well with respect to other effect handler implementations [Kiselyov and Ishii 2015; Schrijvers et al. 2019; Wu and Schrijvers 2015; Wu et al. 2014] and monad transformers.
(5) *libhandler*: a C library that implements effect handlers on top of the regular C stack and uses `longjmp` to transfer control [Leijen 2017a]. This is a *direct* implementation where capturing- and resuming is linear in the stack size as it copies and restores pieces of the C stack directly. It uses the tail-resumptive optimization and insertion ordered "evidence" where it searches through the handler frames on the stack.

Comparing across systems is always difficult as many parts differ – for example, Koka uses Perceus compiler guided reference counting [Reinking et al. 2021] while multi-core OCaml and Haskell use a generational tracing collector, Koka has few standard optimizations while both OCaml and Haskell are sophisticated compilers with decades of development, etc. We selected current best-in-class implementations that compile to native code so execution times are somewhat comparable. As

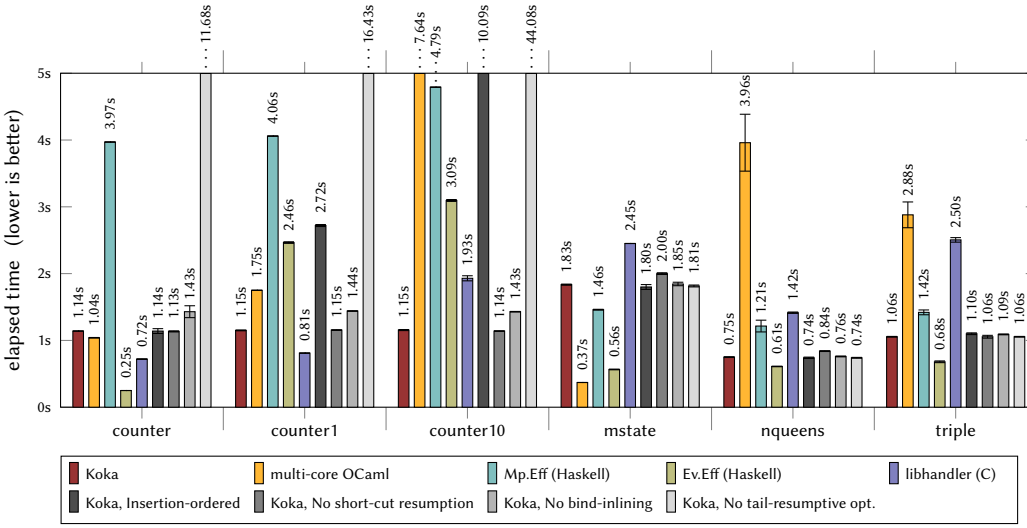

Fig. 6. Execution time averaged over 10 runs

such, the results are meant to establish if the effect handler compilation strategies described in this paper are viable and can be competitive, but should not be interpreted as a measure of absolute performance between systems and languages. Execution times are shown in Figure 6. The execution times are averaged over 10 runs, on an AMD 5950X at 3.4Ghz with 32GiB memory running Ubuntu 20.04, with Koka v2.1.2, multi-core OCaml 4.10, libhandler v0.5, and GHC 8.6.5.

Our benchmarks are taken from [Kiselyov and Ishii 2015], and each is designed to probe specific aspects of effect handling with minimal other computation and allocation overheads:

- *counter* shows how the most common tail-resumptive effects are handled;
- *counter1* and *counter10* emphasize the impact of nested handlers;
- *mstate* demonstrates the use of full first-class resumptions (captured under a lambda);
- *nqueens* and *triple* uses multi-shot resumptions.

Below we discuss the benchmark results.

- *counter*. This benchmark implements a state effect using a mutable reference such that both *get* and *set* operations are tail-resumptive. It then performs 200M *get* and *set* operations in a tight loop. The tail-resumptive optimization in Koka and the fast stack switching in OCaml seem to perform similarly and the execution times are very close. The libhandler C implementation is 1.5× faster than Koka – we believe this is because it does no allocation at all. In contrast, both Koka and OCaml still allocate at each operation (for example, OCaml allocates a continuation object per resumption [Sivaramakrishnan et al. 2021]).
  Moreover, *Mp.Eff* is about 4× slower as Koka, but *Ev.Eff* is 4× faster! This is because GHC is able to fully inline the handler and operations and optimizes almost all effect handling code away. When we remove the inline pragma on the state handler definition, the benchmark takes about 2.02s which is more in line with the results seen in *counter1* and *counter10*. We also ran this benchmark with the tail-resumption optimization turned off; this causes Koka to always allocate a resumption and take the slow path through the monadic bindings making it 10× slower than the optimized version.
- *counter1*. This is the same as *counter* but with one (unused) reader effect handler in between. This time Koka is 1.5× faster than OCaml: due to evidence passing, the execution times of

the tail-resumptive *get* and *set* operations are independent of any other handlers that are in the context (as the handler is found at a constant offset in the canonical evidence vectors). In contrast, multi-core OCaml always yields up one handler stack segment at a time and thus each *get* and *set* operation needs to pass through the reader handler incurring a runtime cost.

- *counter10*. Same as *counter1* but now with 10 reader handlers under the state handler. Again Koka execution is (almost) the same as for *counter1* but we can see that all implementations without tail-resumptive optimization or evidence-passing get slower with each added handler due to the linear search at each operation call.
  The *counter10* benchmark may seem artificial but we believe this pattern to be common in practice. Many uses of effect handlers are to provide contextual state and environments; for example, a type checker may have a current substitution, the type environment, a unique identifier generator, etc. Such nested handlers may thus be quite common in general.
- *mstate*. This is the same as *counter* but now implements the state effect in a monadic way as shown in Section 2.2. This means that the operations are no longer tail-resumptive since the resumption is captured under a lambda. To reduce the execution time, *mstate* only performs 20M *get* and *set* operations (versus 200M in the tail-resumptive *counter* benchmark). This is a worst-case for Koka as it needs to allocate a fresh resumption for each operation call, and it is about 5× slower than multi-core OCaml here. Surprisingly, both *Mp.Eff* and *Ev.Eff* are faster than Koka here – again, the small benchmark can be optimized impressively well by GHC.
- *nqueens*. Calculates all solutions to the *queens* problem of size 12 using a *choice* effect to elegantly express backtracking similar to the non-determinism example in Section 2.2. Multiple resumptions are not directly supported in multi-core OCaml but we can use `Obj.clone_continuation` to manually copy resumptions[6]. Here OCaml is about 5× slower than Koka. We think that this is mostly due to the need to clone a resumption while in Koka (and Haskell) the resumption function is shared over multiple resumes.
- *triple*. Finds Pythogorean triples by using multi-shot resumptions for backtracking, and the performance is therefore very similar to that of *nqueens*.

To better quantify the impact of each optimization individually, we also measured the performance of Koka with various optimizations disabled: (1) Koka using *insertion ordered* evidence (Section 2.5), (2) without fast path *bind inlining* (Section 2.10), (3) without *short-cut resumptions* (Section 2.8), and (4) without *tail-resumptive* optimization (Section 2.6).

As we can see in Figure 6, insertion-order shows the high linear search overhead in *counter1* and *counter10*, while short-cut resumptions offer a modest 10% improvement in *mstate* and *nqueens*. Bind-inlining speeds up the *counter* benchmarks by 25% but has less effect on more allocation intensive benchmarks. Finally, tail-resumptive optimization speeds up the *counter* benchmarks by 10×. As we argued before, most operations in practice are tail-resumptive and we consider this an important optimization.

Overall, the results look promising and show our compilation strategy can be competitive with specialized runtime implementations. With respect to evidence translation versus evidence passing, it seems evidence translation can have the advantage in performance: even though *Mp.Eff* and *Ev.Eff* have very similar implementations, the generalized evidence passing library is about twice as slow as the *Ev.Eff* library over our benchmarks. We believe this is partly caused by the more static nature of evidence in *Ev.Eff* and which makes it more amenable to compiler optimizations.

[6]It works for our particular benchmark, but generally multiple resumptions do not work reliably (as currently implemented) for two reasons: the optimizer is not aware of multiple resumptions and may generate invalid code when optimizing across function calls (this is a problem for libhandler as well [Leijen 2017a]), and cloning a continuation does not compose with other operations that may not clone their own continuation (leading to a runtime crash).

6 RELATED WORK

Throughout the paper, we compare with the most directly related work [Sivaramakrishnan et al. 2021; Xie et al. 2020; Xie and Leijen 2020] inline. Here we briefly discuss other related work.

In contrast to the monadic translation, Hillerström et al. [2017] describe a CPS based translation of effect handlers. Similar to bubbling and evidence passing, this avoids capturing the evaluation context by making all continuations explicit. Forster et al. [2019] show how delimited control, monads, and effect handlers can all be expressed in terms of each other in an untyped setting. However, their encoding of effect handlers in terms of shift-reset does not preserve typeability (due to the lack of answer type polymorphism [Asai and Kameyama 2007; Danvy and Filinski 1989]). In our work typing is preserved by using multi-prompt control with explicitly typed markers. Kiselyov and Sivaramakrishnan [2017] present a direct embedding of effect handlers in OCaml based on shift-reset (using the *delimcc* library), where they use an out-of-band technique [Kiselyov et al. 2006] to work around the lack of answer type polymorphism. Kammar et al. [2013] also embed effect handlers in OCaml using shift-reset, where they use a global mutable variable to hold the current stack of handlers (which can be considered as the insertion-ordered evidence vector).

Capability passing [Brachthäuser et al. 2020; Schuster et al. 2020] is related to algebraic effect handlers. It has the concept of handlers but each handler must be passed explicitly by name and there is no search for the innermost handler when an operation is performed (but the handler is an explicit argument). Schuster et al. [2020] show that capability based handlers can be efficiently compiled using iterated CPS translation (however, the translation requires whole-program monomorphisation). Generally, with capability passing, handler names are captured statically in a resumption and, similar to evidence translation (EPT), one gets either stuck or the "wrong" results for the examples in Section 2.12. Evidence passing avoids this problem by keeping the evidence vector separate from general expressions and not capturing it as part of a resumption.

Zhang and Myers [2019] and Brachthäuser et al. [2020] (using capability passing as a target calculus) develop "lexically scoped effect handlers". It is argued that such handlers avoid accidental capture of operations, and allow better modular reasoning for higher-order abstractions. However, this approach deviates from the semantics of algebraic effect handlers as originally defined by Plotkin and Pretnar [2013]. In particular, common source-to-source transformations are not always valid in this setting. For example, inlining a lambda expression instead of passing it by argument may change the semantics of an operation. In contrast to algebraic effect handlers there is also no untyped dynamic semantics, and types are required to give semantics to a program.

Flatt and Dybvig [2020] extended Racket (and the Chez Scheme runtime) with support for *continuation marks*. These provide efficient access to key-value maps that are bound in the evaluation context. As such, we can view these as a kind of built-in (tail-resumptive) state effect handler.

7 CONCLUSION

Generalized evidence passing is a promising technique for compiling effect handlers to standard target platforms, and can offer competitive performance relative to specialized runtimes. Moreover, our formalization explores the design space of implementation techniques and their trade-offs. We hope our study will lead to further improvements of effect handlers implementations in the future.

ACKNOWLEDGMENTS

We would like to thank Taro Sekiyama, and other anonymous reviewers, for their detailed and insightful feedback on earlier versions of the paper. We also thank Jonathan Brachthäuser his feedback and proof reading of this paper.

REFERENCES

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

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. ICFP'02. Pittsburgh, PA, USA. https://doi.org/10.1145/581478.581494.

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

Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. Nov. 2020. Effects as Capabilities: Effect Handlers and Lightweight Effect Polymorphism. *Proc. ACM Program. Lang.* 4 (OOPSLA). ACM. https://doi.org/10.1145/3428194.

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*, number 30. Cambridge University Press.

Olivier Danvy, and Andrzej Filinski. 1989. *A Functional Abstraction of Typed Contexts*. 89/12. DIKU, University of Copenhagen.

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

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

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

Kavon Farvardin, and John Reppy. 2020. From Folklore to Fact: Comparing Implementations of Stacks and Continuations. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 75–90. PLDI 2020. London, UK. https://doi.org/10.1145/3385412.3385994.

Matthias Felleisen, Daniel P. Friedman, Eugene E. Kohlbecker, and Bruce F. Duba. 1986. Reasoning with Continuations. In *Proceedings of the 1st Symposium on Logic in Computer Science (LICS)*, 131–141.

Matthew Flatt, and R. Kent Dybvig. 2020. Compiler and Runtime Support for Continuation Marks. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 45–58. PLDI 2020. ACM, London, UK. https://doi.org/10.1145/3385412.3385981.

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

Carl A. Gunter, Didier Rémy, and Jon G. Riecke. 1995. A Generalization of Exceptions and Control in ML-like Languages. In *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture*, 12–23. FPCA '95. ACM, La Jolla, California, USA. https://doi.org/10.1145/224164.224173.

Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. 2017. Bringing the Web up to Speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 185–200. PLDI 2017. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3062341.3062363.

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

Daniel Hillerström, and Sam Lindley. 2018. Shallow Effect Handlers. In *16th Asian Symposium on Programming Languages and Systems (APLAS'18)*, 415–435. Springer.

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.

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

Oleg Kiselyov, and Hiromi Ishii. 2015. Freer Monads, More Extensible Effects. In *Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell*, 94–105. Haskell '15. Vancouver, BC, Canada. https://doi.org/10.1145/2804302.2804319.

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. https://doi.org/10.1145/1159803.1159808.

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

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

Daan Leijen. 2017. 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.

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

Daan Leijen. Nov. 2020. The Koka Programming Language. https://koka-lang.github.io.

Daan Leijen. May 2021. Effect Handler Benchmarks. https://github.com/daanx/effect-bench.

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

Michel Parigot. 1992. $\lambda\mu$-Calculus: An Algorithmic Interpretation of Classical Natural Deduction. In *Logic Programming and Automated Reasoning*, edited by Andrei Voronkov, 190–201. Springer Berlin Heidelberg.

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

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

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

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

Alex Reinking, Ningning Xie, Leonardo de Moura, and Daan Leijen. 2021. Perceus: Garbage Free Reference Counting with Reuse. In *42rd ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '21.

Tom Schrijvers, Maciej Piróg, Nicolas Wu, and Mauro Jaskelioff. 2019. Monad Transformers and Modular Algebraic Effects: What Binds Them Together. In *Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell*, 98–113. Haskell 2019. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3331545.3342595.

Philipp Schuster, Jonathan Immanuel Brachthäuser, and Klaus Ostermann. Aug. 2020. Compiling Effect Handlers in Capability-Passing Style. *Proc. ACM Program. Lang.* 4 (ICFP). ACM. https://doi.org/10.1145/3408975.

KC Sivaramakrishnan, Stephen Dolan, Leo White, Tom Kelly, Sadiq Jaffer, and Anil Madhavapeddy. 2021. Retrofitting Effect Handlers onto OCaml. In *42rd ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '21.

Martin Sulzmann, Manuel Chakravarty, Simon Peyton Jones, and Kevin Donnelly. Jan. 2007. System F with Type Equality Coercions. In *ACM SIGPLAN International Workshop on Types in Language Design and Implementation (TLDI'07)*, 53–66. ACM press.

Herb Sutter. 2019. Zero-Overhead Deterministic Exceptions: Throwing Values. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p0709r4.pdf. ISO/IEC WG2.1, P0709R4.

Nicolas Wu, and Tom Schrijvers. 2015. Fusion for Free: Efficient Algebraic Effect Handlers. In *Proceedings of the 12th International Conference on Mathematics of Program Construction*, 9129:302. MPC 2015. Springer, Königswinter, Germany.

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

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

Ningning Xie, and Daan Leijen. Aug. 2020. Effect Handlers in Haskell, Evidently. In *Proceedings of the 2020 ACM SIGPLAN Symposium on Haskell*. Haskell'20. Jersey City, NJ. https://doi.org/10.1145/3406088.3409022.

Ningning Xie, and Daan Leijen. Mar. 2021a. Mp.Eff: Efficient Effect Handlers Based on Evidence Passing Semantics. https://github.com/xnning/MpEff.

Ningning Xie, and Daan Leijen. Mar. 2021b. *Generalized Evidence Passing for Effect Handlers*. MSR-TR-2021-5. Microsoft Research. Extended version with proofs.

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