Continuation Passing Style for Effect Handlers

Daniel Hillerström[1], Sam Lindley[2], Robert Atkey[3], and KC Sivaramakrishnan[4]

1 **The University of Edinburgh, United Kingdom**
`daniel.hillerstrom@ed.ac.uk`
2 **The University of Edinburgh, United Kingdom**
`sam.lindley@ed.ac.uk`
3 **University of Strathclyde, United Kingdom**
`robert.atkey@strath.ac.uk`
4 **University of Cambridge, United Kingdom**
`sk826@cl.cam.ac.uk`

Abstract

We present Continuation Passing Style (CPS) translations for Plotkin and Pretnar's effect handlers with Hillerström and Lindley's row-typed fine-grain call-by-value calculus of effect handlers as the source language. CPS translations of handlers are interesting theoretically, to explain the semantics of handlers, and also offer a practical implementation technique that does not require special support in the target language's runtime.

We begin with a first-order CPS translation into untyped lambda calculus which manages a stack of continuations and handlers as a curried sequence of arguments. We then refine the initial CPS translation first by uncurrying it to yield a properly tail-recursive translation and second by making it higher-order in order to contract administrative redexes at translation time. We prove that the higher-order CPS translation simulates effect handler reduction. We have implemented the higher-order CPS translation as a JavaScript backend for the Links programming language.

1998 ACM Subject Classification D.3.3 Language Constructs and Features, F.3.2 Semantics of Programming Languages

Keywords and phrases effect handlers, delimited control, continuation passing style

Digital Object Identifier 10.4230/LIPIcs.FSCD.2017.18

1 Introduction

Algebraic effects, introduced by Plotkin and Power [25], and their handlers, introduced by Plotkin and Pretnar [26], provide a modular foundation for effectful programming. Though originally studied in a theoretical setting, effect handlers are also of practical interest, as witnessed by a range of recent implementations [2, 4, 8, 11, 13, 15, 17, 20]. Notably, the Multicore OCaml project brings effect handlers to the OCaml programming language as a means for abstracting over different scheduling strategies [8]. As a programming abstraction, effect handlers can be viewed as a more modular alternative to monads [23, 29].

An algebraic effect is a signature of operations along with an equational theory on those operations. An effect handler is a delimited control operator which *interprets* a particular subset of the signature of operations up to equivalences demanded by the equational theory. In practice current implementations do not support equations on operations, and as such, the underlying algebra is the free algebra, allowing handlers maximal interpretative freedom. Correspondingly, in this paper we assume free algebras for effects.

There are many feasible implementation strategies for effect handlers. For instance, Kammar et al.'s libraries make use variously of free monads, continuation monads, and delimited continuations [13]; the server backend of the Links programming language uses an abstract machine [11]; and Multicore OCaml relies on explicit manipulation of the runtime stack [8]. Explicit stack manipulation is appealing when one has complete control over the design of the backend. Similarly, delimited continuations are appealing when the backend already has support for delimited continuations [13, 16]. However, if the backend does not support delimited continuations or explicit stack manipulation, for instance when targeting a backend language like JavaScript, an alternative approach is necessary.

In this paper we study how to translate effect handlers from a rich source lambda calculus into a plain lambda calculus without necessarily requiring additional primitives. Specifically, we study *continuation passing style* (CPS) translations for handlers. CPS does not extend the lambda calculus, rather, CPS amounts to using a restricted subset of the plain lambda calculus. Furthermore CPS is an established intermediate representation used by compilers [1, 14], making it a realistic compilation target, and it provides a general framework for implementing control flow, making it a good fit for implementing control operators such as effect handlers. The only existing CPS translation for effect handlers we are aware of is due to Leijen [17]. His work differs from ours in that he does not CPS translate away operations or handlers, but rather uses a CPS translation to lift code into a free monad, relying on a special handle primitive in the runtime. Leijen's formalism includes features that we do not. In particular, he performs a selective CPS translation in order to avoid overhead in code that does not use algebraic effects. We have implemented a JavaScript backend for Links, based on our formalism, which also performs a selective CPS translation. In this paper we do not formalise the selective aspect of the translation as it is mostly orthogonal.

This paper makes the following contributions:

1. A concise curried CPS translation for effect handlers (§4.2.1). The translation has two shortcomings: it is not properly tail-recursive and it yields administrative redexes.
2. A higher-order uncurried variant of the curried CPS translation, which is properly tail-recursive and partially evaluates administrative redexes at translation time (§4.3).
3. A correctness proof for the higher-order CPS translation (§4.3).
4. An implementation of the higher-order CPS translation as a backend for Links [5] (§5).

The paper proceeds as follows. §2 gives a primer to programming with effect handlers. §3 describes a core calculus of algebraic effects and handlers. §4 presents the CPS translations, correctness proof, and variations. §5 briefly describes our implementation. §6 concludes.

2 Modular interpretation of effectful computations

In this section we give a flavour of programming with algebraic effects and handlers. We use essentially the syntax of our core calculus (§3) along with syntactic sugar to make the examples more readable (in particular, we use direct-style rather than fine-grain call-by-value). We consider two effects: *nondeterminism* and *exceptions*, the former given by a nondeterministic choice operation Choose; the latter by an exception-raising operation Fail.

We combine nondeterminism and exceptions to model a drunk attempting to toss a coin. The coin toss and whether it succeeds is modelled by the Choose operation. The drunk dropping the coin is modelled by the Fail operation.

```
drunkToss : Toss ! {Choose : Bool; Fail : Zero}
drunkToss = if do Choose then
                  if do Choose then Heads else Tails
            else absurd do Fail
```

This code declares an *abstract computation* drunkToss, which potentially invokes two abstract operations Choose and Fail using the **do** primitive. The type signature of drunkToss reads: drunkToss is a computation with effect signature {Choose : Bool; Fail : Zero} and return value Toss, whose constructors are Heads and Tails. The order of operation names in the effect signature is irrelevant. The first invocation of Choose decides whether the coin is caught, while the second invocation decides the face. The Fail operation never returns, so its return type is Zero, which is eliminated by the **absurd** construct.

A possible interpretation of drunkToss uses lists to model nondeterminism, where the return operation lifts its argument into a singleton list, the choose handler concatenates lists of possible outcomes, and the fail operation returns the empty list:

$$
\begin{aligned}
&\mathsf{nondet} : \alpha\ !\ \{\mathsf{Choose} : \mathsf{Bool}; \mathsf{Fail} : \mathsf{Zero}\} \Rightarrow \mathsf{List}\ \alpha\ !\ \{\} \\
&\mathsf{nondet} = \mathbf{return}\ x \mapsto [x] \\
&\qquad\quad \mathsf{Choose}\ r \mapsto r\ \mathsf{True} +\!\!+ r\ \mathsf{False} \\
&\qquad\quad \mathsf{Fail}\ r \mapsto []
\end{aligned}
$$

The type signature conveys that the handler transforms an abstract computation into a concrete computation where the operations Choose and Fail are instantiated. The handler comprises three clauses. The return clause specifies how to handle the return value of the computation. The other two clauses specify how to interpret the operations. The Choose clause binds a *resumption* r, a function which captures the delimited continuation of the operation Choose. The interpretation of Choose explores both alternatives by invoking the resumption twice and concatenating the results. The Fail clause ignores the provided resumption and returns simply the empty list, []. Thus handling drunkToss with nondet yields all possible positive outcomes, i.e. [Heads, Tails].

A key feature of effect handlers is that the use of an operation is *decoupled* from its interpretation. Rather than having one handler which handles every operation, we may opt for a more fine-grained approach using multiple handlers which each instantiate a subset of the abstract operations. For example, we can define handlers for each of the two operations.

$$
\begin{aligned}
&\mathsf{allChoices} : \alpha\ !\ \{\mathsf{Choose} : \mathsf{Bool}; \rho\} \Rightarrow \mathsf{List}\ \alpha\ !\ \{\mathsf{Choose} : \theta; \rho\} \\
&\mathsf{allChoices} = \mathbf{return}\ x \mapsto [x] \\
&\qquad\quad\ \ \mathsf{Choose}\ r \mapsto r\ \mathsf{True} +\!\!+ r\ \mathsf{False}
\end{aligned}
$$

This effect signature differs from the signature of nondet; it mentions only one operation and it mentions an *effect variable* variable ρ which ranges over all unmentioned operations. In addition Choose is mentioned in output effect signature. (This is because we adopt a Remy-style row-type system [28] in which negative information is made explicit. At the cost of slightly less expressivity, it is possible to eliminate these effects in the output type by permitting effect shadowing as in Koka [17] and Frank [20].) The notation $\mathsf{Choose} : \theta$ denotes that the operation may or may not appear again. This handler implicitly *forwards* Fail to another enclosing handler such as the following.

$$
\begin{aligned}
&\mathsf{failure} : \alpha\ !\ \{\mathsf{Fail} : \mathsf{Zero}; \rho\} \Rightarrow \mathsf{List}\ \alpha\ !\ \{\mathsf{Fail} : \theta; \rho\} \\
&\mathsf{failure} = \mathbf{return}\ x \mapsto [x] \\
&\qquad\quad \mathsf{Fail}\ r \mapsto []
\end{aligned}
$$

Now, we have two possible compositions, and we must tread carefully as they are semantically different. For example, **handle** (**handle** drunkToss **with** allChoices) **with** failure yields the empty list. But **handle** (**handle** drunkToss **with** failure) **with** allChoices yields all possible outcomes, i.e. [[Heads], [Tails], []] where the empty list conveys failure. The behaviour of nondet can be obtained by concatenating the result of the latter composition. Effect handlers also permit multiple interpretations of the same abstract computation.

Value types	$A, B ::= A \to C \mid \forall \alpha^K.C$	Types	$T ::= A \mid C \mid E \mid R \mid P \mid F$
	$\mid \langle R \rangle \mid [R] \mid \alpha$	Kinds	$K ::= \mathsf{Type} \mid \mathsf{Row}_{\mathcal{L}} \mid \mathsf{Presence}$
Computation types	$C, D ::= A!E$		$\mid \mathsf{Comp} \mid \mathsf{Effect} \mid \mathsf{Handler}$
Effect types	$E ::= \{R\}$	Label sets	$\mathcal{L} ::= \emptyset \mid \{\ell\} \uplus \mathcal{L}$
Row types	$R ::= \ell : P; R \mid \rho \mid \cdot$	Type environments	$\Gamma ::= \cdot \mid \Gamma, x : A$
Presence types	$P ::= \mathsf{Pre}(A) \mid \mathsf{Abs} \mid \theta$	Kind environments	$\Delta ::= \cdot \mid \Delta, \alpha : K$
Handler types	$F ::= C \Rightarrow D$		

Figure 1 Types, effects, kinds, and environments

3 A calculus of handlers and rows

In this section, we recapitulate our Church-style row-polymorphic call-by-value calculus for effect handlers $\lambda^{\rho}_{\text{eff}}$ (pronounced "lambda-eff-row") [11].

The design of $\lambda^{\rho}_{\text{eff}}$ is inspired by the λ-calculi of Kammar et al. [13], Pretnar [27], and Lindley and Cheney [19]. As in the work of Kammar et al. [13], each handler can have its own effect signature. As in the work of Pretnar [27], the underlying formalism is fine-grain call-by-value [18], which names each intermediate computation like in A-normal form [9], but unlike A-normal form is closed under β-reduction. As in the work of Lindley and Cheney [19], the effect system is based on row polymorphism.

3.1 Types

The syntax of types, kinds, label sets, and type and kind environments is given in Figure 1.

Value types The function type $A \to C$ maps values of type A to computations of type C. The polymorphic type $\forall \alpha^K.\, C$ is parameterised by a type variable α of kind K. The record type $\langle R \rangle$ represents records with fields constrained by row R. Dually, the variant type $[R]$ represents tagged sums constrained by row R.

Computation types The computation type $A!E$ is given by a value type A and an effect type E, which specifies the operations a computation inhabiting this type may perform.

Row types Effect, record, and variant types are defined in terms of rows. A row type embodies a collection of distinct labels, each of which is annotated with a presence type. A presence type indicates whether a label is *present* with some type A ($\mathsf{Pre}(A)$), *absent* (Abs) or *polymorphic* in its presence (θ). Row types are either *closed* or *open*. A closed row type ends in $\cdot$, whilst an open row type ends with a *row variable* ρ. The row variable in an open row can be instantiated with additional labels. We identify rows up to reordering of labels. For instance, we consider the rows $\ell_1 : P_1; \cdots; \ell_n : P_n; \cdot$ and $\ell_n : P_n; \cdots; \ell_1 : P_1; \cdot$ equivalent. Absent labels in closed rows are redundant: if R is closed, then $\ell : \mathsf{Abs}; R$ is equivalent to R. The unit and empty type are definable in terms of row types. We define the unit type as the empty, closed record, that is, $\langle \cdot \rangle$. Similarly, we define the empty type as the empty, closed variant $[\cdot]$. Often we omit the $\cdot$ for closed rows.

Handler types The handler type $C \Rightarrow D$ represents handlers that transform computations of type C into computations of type D.

$$
\begin{array}{lll}
\text{Values} & V, W & ::= x \mid \lambda x^{A}.\, M \mid \Lambda \alpha^{K}.\, M \mid \langle\rangle \mid \langle \ell = V; W \rangle \mid (\ell\ V)^{R} \\
\text{Computations} & M, N & ::= V\ W \mid V\ T \\
& & \quad \mid \mathbf{let}\ \langle \ell = x; y \rangle = V\ \mathbf{in}\ N \mid \mathbf{case}\ V \{ \ell\ x \mapsto M; y \mapsto N \} \mid \mathbf{absurd}^{C}\ V \\
& & \quad \mid \mathbf{return}\ V \mid \mathbf{let}\ x \leftarrow M\ \mathbf{in}\ N \\
& & \quad \mid (\mathbf{do}\ \ell\ V)^{E} \mid \mathbf{handle}\ M\ \mathbf{with}\ H \\
\text{Handlers} & H & ::= \{\mathbf{return}\ x \mapsto M\} \mid \{\ell\ p\ r \mapsto M\} \uplus H
\end{array}
$$

Figure 2 Term syntax

Kinds We have six kinds: Type, Comp, Effect, $\mathsf{Row}_{\mathcal{L}}$, Presence, Handler, which respectively classify value types, computation types, effect types, row types, presence types, and handler types. Row kinds are annotated with a set of labels $\mathcal{L}$. The kind of a complete row is $Row_{\emptyset}$. More generally, the kind $Row_{\mathcal{L}}$ denotes a partial row that cannot mention the labels in $\mathcal{L}$. We write $\ell : A$ as syntactic sugar for $\ell : \mathsf{Pre}(A)$.

Type variables We let α, ρ and θ range over type variables. By convention we use α for value type variables or for type variables of unspecified kind, ρ for type variables of row kind, and θ for type variables of presence kind.

Type and kind environments Type environments (Γ) map term variables to their types and kind environments (Δ) map type variables to their kinds.

3.2 Terms

The terms are given in Figure 2. We let x, y, z, r, p range over term variables. By convention, we use r to denote resumption names. The syntax partitions terms into values, computations and handlers. Value terms comprise variables (x), lambda abstraction ($\lambda x^{A}.\, M$), type abstraction ($\Lambda \alpha^{K}.\, M$), and the introduction forms for records and variants. Records are introduced using the empty record $\langle\rangle$ and record extension $\langle \ell = V; W \rangle$, whilst variants are introduced using injection $(\ell\ V)^{R}$, which injects a field with label ℓ and value V into a row whose type is R. The annotation supports bottom-up type reconstruction.

All elimination forms are computation terms. Abstraction and type abstraction are eliminated using application ($V\ W$) and type application ($V\ T$) respectively. The record eliminator ($\mathbf{let}\ \langle \ell = x; y \rangle = V\ \mathbf{in}\ N$) splits a record V into x, the value associated with ℓ, and y, the rest of the record. Non-empty variants are eliminated using the case construct ($\mathbf{case}\ V\ \{\ell\ x \mapsto M; y \mapsto N\}$), which evaluates the computation M if the tag of V matches ℓ. Otherwise it falls through to y and evaluates N. The elimination form for empty variants is ($\mathbf{absurd}^{C}\ V$). A trivial computation ($\mathbf{return}\ V$) returns value V. The expression ($\mathbf{let}\ x \leftarrow M\ \mathbf{in}\ N$) evaluates M and binds the result value to x in N.

The construct $(\mathbf{do}\ \ell\ V)^{E}$ invokes an operation ℓ with value argument V. The handle construct ($\mathbf{handle}\ M\ \mathbf{with}\ H$) runs a computation M with handler definition H. A handler definition H consists of a return clause $\{\mathbf{return}\ x \mapsto M\}$ and a possibly empty set of operation clauses $\{\ell\ p\ r \mapsto N_{\ell}\}_{\ell \in \mathcal{L}}$. The return clause defines how to handle the final return value of the handled computation, which is bound to x in M. The operation clause for ℓ binds the operation parameter to p and the resumption r in N_{ℓ}.

We define three projections on handlers: H^{ret} yields the singleton set containing the return clause of H and H^{ℓ} yields the set of either zero or one operation clauses in H that handle the operation ℓ and H^{ops} yields the set of all operation clauses in H. We write $dom(H)$

for the set of operations handled by H. As our calculus is Church-style, we annotate various term forms with type or kind information; (term abstraction, type abstraction, injection, operations, and empty cases); we sometimes omit these annotations.

Syntactic sugar We make use of standard syntactic sugar for pattern matching, n-ary record extension, n-ary case elimination, and n-ary tuples. For instance:

$$\begin{aligned}
\lambda\langle x, y\rangle.M &\equiv \lambda z.\mathbf{let}\ \langle x, y\rangle = z\ \mathbf{in}\ M \\
\langle V_1, \ldots, V_n\rangle &\equiv \langle 1 = V_1, \ldots, n = V_n\rangle \\
\mathbf{case}\ V\ \{\ell_1\ x \mapsto N_1; \ldots; \ell_n\ x \mapsto N_n; z \mapsto N\} &\equiv \mathbf{case}\ V\ \{\ell_1\ x \mapsto N_1; z \mapsto \mathbf{case}\ z\ \{\ell_2\ x \mapsto N_1; z \mapsto \ldots \\
&\qquad \mathbf{case}\ z\ \{\ell_n\ x \mapsto N_1; z \mapsto N\}\ldots\}
\end{aligned}$$

3.3 Kinding and typing

The kinding judgement $\Delta \vdash T : K$ states that type T has kind K in kind environment Δ. The value typing judgement $\Delta; \Gamma \vdash V : A$ states that value term V has type A under kind environment Δ and type environment Γ. The computation typing judgement $\Delta; \Gamma \vdash M : C$ states that term M has computation type C under kind environment Δ and type environment Γ. The handler typing judgement $\Delta; \Gamma \vdash H : C \Rightarrow D$ states that handler H has type $C \Rightarrow D$ under kind environment Δ and type environment Γ. In the typing judgements, we implicitly assume that Γ, A, C, and D, are well-kinded with respect to Δ. We define the function $FTV(\Gamma)$ to be the set of free type variables in Γ. The full kinding and typing rules are given in Appendix A. The interesting rules are T-Do, T-Handle, and T-Handler.

T-Do
$$\frac{\Delta; \Gamma \vdash V : A \qquad E = \{\ell : A \to B; R\}}{\Delta; \Gamma \vdash (\mathbf{do}\ \ell\ V)^E : B!E}$$

T-Handle
$$\frac{\Delta; \Gamma \vdash M : C \qquad \Delta; \Gamma \vdash H : C \Rightarrow D}{\Delta; \Gamma \vdash \mathbf{handle}\ M\ \mathbf{with}\ H : D}$$

T-Handler
$$\frac{\begin{array}{c} C = A!\{(\ell_i : A_i \to B_i)_i; R\} \qquad D = B!\{(\ell_i : P_i)_i; R\} \qquad H = \{\mathbf{return}\ x \mapsto M\} \uplus \{\ell_i\ p\ r \mapsto N_{\ell_i}\}_i \\ \Delta; \Gamma, x : A \vdash M : D \qquad [\Delta; \Gamma, p : A_i, r : B_i \to D \vdash N_{\ell_i} : D]_i \end{array}}{\Delta; \Gamma \vdash H : C \Rightarrow D}$$

The T-Handler rule is where most of the work happens. The effect rows on the computation type C and the output computation type D must share the same suffix R. This means that the effect row of D must explicitly mention each of the operations ℓ_i to say whether an ℓ_i is present with a given type signature, absent, or polymorphic in its presence. The row R describes the operations that are forwarded. It may include a row-variable, in which case an arbitrary number of effects may be forwarded by the handler.

3.4 Operational semantics

We give a small-step operational semantics for $\lambda^{\rho}_{\text{eff}}$. Figure 3 gives the reduction rules. The reduction relation $\rightsquigarrow$ is defined on computation terms. We use evaluation contexts ($\mathcal{E}$) to focus on the active expression. The interesting rules are the handler rules. We write $BL(\mathcal{E})$ for the set of operation labels bound by $\mathcal{E}$.

$$BL([\,]) = \emptyset \qquad BL(\mathbf{let}\ x \leftarrow \mathcal{E}\ \mathbf{in}\ N) = BL(\mathcal{E}) \qquad BL(\mathbf{handle}\ \mathcal{E}\ \mathbf{with}\ H) = BL(\mathcal{E}) \cup dom(H)$$

The rule S-Handle-Ret invokes the return clause of a handler. The rule S-Handle-op handles an operation by invoking the appropriate operation clause. The constraint $\ell \notin BL(\mathcal{E})$ ensures that no inner handler inside the evaluation context is able to handle the operation: thus a handler is able to reach past any other inner handlers that do not handle ℓ.

We write R^+ for the transitive closure of relation R.

$$
\begin{array}{lrll}
\text{S-App} & (\lambda x^A.\, M)\, V & \rightsquigarrow & M[V/x] \\
\text{S-TyApp} & (\Lambda \alpha^K.\, M) A & \rightsquigarrow & M[A/\alpha] \\
\text{S-Split} & \textbf{let } \langle \ell = x; y \rangle = \langle \ell = V; W \rangle \textbf{ in } N & \rightsquigarrow & N[V/x, W/y] \\
\text{S-Case}_1 & \textbf{case } (\ell\ V)^R \{\ell\ x \mapsto M; y \mapsto N\} & \rightsquigarrow & M[V/x] \\
\text{S-Case}_2 & \textbf{case } (\ell\ V)^R \{\ell'\ x \mapsto M; y \mapsto N\} & \rightsquigarrow & N[(\ell\ V)^R/y], \quad \text{if } \ell \neq \ell' \\
\text{S-Let} & \textbf{let } x \leftarrow \textbf{return } V \textbf{ in } N & \rightsquigarrow & N[V/x] \\
\text{S-Handle-Ret} & \textbf{handle } (\textbf{return } V) \textbf{ with } H & \rightsquigarrow & N[V/x], \quad \text{where } H^{\text{ret}} = \{\textbf{return } x \mapsto N\} \\
\text{S-Handle-Op} & \textbf{handle } \mathcal{E}[\textbf{do } \ell\ V] \textbf{ with } H & \rightsquigarrow & N[V/p, \lambda y.\, \textbf{handle } \mathcal{E}[\textbf{return } y] \textbf{ with } H/r], \\
 & & & \quad \text{where } \ell \notin BL(\mathcal{E}) \text{ and } H^\ell = \{\ell\ p\ r \mapsto N\} \\
\text{S-Lift} & \mathcal{E}[M] & \rightsquigarrow & \mathcal{E}[N], \quad \text{if } M \rightsquigarrow N
\end{array}
$$

Evaluation contexts $\mathcal{E} ::= [\,] \mid \textbf{let } x \leftarrow \mathcal{E} \textbf{ in } N \mid \textbf{handle } \mathcal{E} \textbf{ with } H$

Figure 3 Small-step operational semantics

▶ **Definition 1.** We say that computation term N is normal with respect to effect E, if N is either of the form $\textbf{return } V$, or $\mathcal{E}[\textbf{do } \ell\ W]$, where $\ell \in E$ and $\ell \notin BL(\mathcal{E})$.

▶ **Theorem 2** (Type Soundness). *If $\vdash M : A!E$, then there exists $\vdash N : A!E$, such that $M \rightsquigarrow^+ N \not\rightsquigarrow$, and N is normal with respect to effect E.*

4 CPS translations

We now turn to the main business of the paper: continuation passing style translations from a calculus with effects and handlers to a calculus with neither. In doing so, we achieve two aims. First, we give an alternative formal explanation of effect handlers' semantics independent of the standard free monad interpretation. Second, we offer an implementation technique that is more efficient than the free monad interpretation because it does not allocate intermediate computation trees. We present our CPS translation in stages. We start with a basic translation for fine-grain call-by-value without handlers in §4.1. We then formulate first-order translations that progressively move from representing the dynamic stack of handlers as functions to explicit stacks in §4.2. This prepares us for our final higher-order one-pass translation in §4.3 that uses static computation at translation time to avoid administrative reductions during runtime. We then consider shallow handlers in §4.4, and exceptions in §4.5.

The untyped target calculus for our CPS translations is given in Figure 4. As in our fine-grain call-by-value source language, we make a syntactic distinction between values and computations. The reductions are standard β-reductions, also given in Figure 4. There are three differences from fine-grain call-by-value: *i)* we have no explicit **return** to lift values to computations, value terms are silently included in computation terms; *ii)* there is no **let** in the target calculus, because all sequencing will be expressed via continuation passing; and *iii)* we permit the function position of an application to be a computation (i.e., the application form is $M\ W$ rather than $V\ W$). This latter relaxation is used in our initial CPS translations, but will be ruled out in our final translation.

4.1 CPS translation for fine-grain call-by-value

We start by giving a CPS translation of the handler-free subset of $\lambda^\rho_{\text{eff}}$ in Figure 5. Fine-grain call-by-value admits a particularly simple CPS translation due to the separation of values and computations. All constructs from the source language are translated homomorphically into

Syntax

$$
\begin{array}{lll}
\text{Values} & V, W & ::= x \mid \lambda x.M \mid \langle\rangle \mid \langle \ell = V; W\rangle \mid \ell\ V \\
\text{Computations} & M, N & ::= V \mid M\ W \mid \mathbf{let}\ \langle \ell = x; y\rangle = V\ \mathbf{in}\ N \\
& & \quad \mid\ \mathbf{case}\ V\,\{\ell\ x \mapsto M; y \mapsto N\} \mid \mathbf{absurd}\ V \\
\text{Evaluation contexts} & \mathcal{E} & ::= [\,] \mid \mathcal{E}\ W
\end{array}
$$

Reductions

$$
\begin{array}{lr}
\text{U-App} & (\lambda x.\, M)\, V \rightsquigarrow M[V/x] \\
\text{U-Split} & \mathbf{let}\ \langle \ell = x; y\rangle = \langle \ell = V; W\rangle\ \mathbf{in}\ N \rightsquigarrow N[V/x, W/y] \\
\text{U-Case}_1 & \mathbf{case}\ (\ell\ V)\{\ell\ x \mapsto M; y \mapsto N\} \rightsquigarrow M[V/x] \\
\text{U-Case}_2 & \mathbf{case}\ (\ell\ V)\{\ell'\ x \mapsto M; y \mapsto N\} \rightsquigarrow N[\ell\ V/y], \quad \text{if } \ell \neq \ell' \\
\text{U-Lift} & \mathcal{E}[M] \rightsquigarrow \mathcal{E}[N], \quad \text{if } M \rightsquigarrow N
\end{array}
$$

Figure 4 Untyped target calculus

Values

$$
\begin{array}{rl}
[\![x]\!] &= x \\
[\![\lambda x.M]\!] &= \lambda x.[\![M]\!] \\
[\![\Lambda\alpha.M]\!] &= \lambda k.[\![M]\!]\ k \\
[\![\langle\rangle]\!] &= \langle\rangle \\
[\![\langle \ell = V; W\rangle]\!] &= \langle \ell = [\![V]\!]; [\![W]\!]\rangle \\
[\![\ell\ V]\!] &= \ell\ [\![V]\!]
\end{array}
$$

Computations

$$
\begin{array}{rl}
[\![V\ W]\!] &= [\![V]\!]\ [\![W]\!] \\
[\![V\ A]\!] &= [\![V]\!] \\
[\![\mathbf{let}\ \langle \ell = x; y\rangle = V\ \mathbf{in}\ N]\!] &= \mathbf{let}\ \langle \ell = x; y\rangle = [\![V]\!]\ \mathbf{in}\ [\![N]\!] \\
[\![\mathbf{case}\ V\ \{\ell\ x \mapsto M; y \mapsto N\}]\!] &= \mathbf{case}\ [\![V]\!]\ \{\ell\ x \mapsto [\![M]\!]; y \mapsto [\![N]\!]\} \\
[\![\mathbf{absurd}\ V]\!] &= \mathbf{absurd}\ [\![V]\!] \\
[\![\mathbf{return}\ V]\!] &= \lambda k.k\ [\![V]\!] \\
[\![\mathbf{let}\ x \leftarrow M\ \mathbf{in}\ N]\!] &= \lambda k.[\![M]\!](\lambda x.[\![N]\!]k)
\end{array}
$$

Figure 5 First-order CPS translation of fine-grain call-by-value

the target language, except for **return**, **let**, and type abstraction (the translation performs type erasure). Lifting a value V to a computation **return** V is interpreted by passing the value to the current continuation. Sequencing two computations with **let** is translated in the usual continuation passing way. In addition, we explicitly η-expand the translation of a type abstraction in order to ensure that value terms in the source calculus translate to value terms in the target.

4.2 First-order CPS translations of handlers

As is usual for CPS, the translation of a computation term by the basic CPS translation takes a single continuation parameter that represents the context. With effects and handlers in the source language, we must now keep track of two kinds of context in which each computation executes: a *pure context* that tracks the state of pure computation in the scope of the current handler, and an *effect context* that describes how to handle operations in the scope of the current handler. Correspondingly, we have both *pure continuations* (k) and *effect continuations* (h). As handlers can be nested, each computation executes in the context of a *stack* of pairs of pure and effect continuations (as in the abstract machine of [11]).

On invocation of a handler, the pure continuation is initialised to a representation of the return clause and the effect continuation to a representation of the operation clauses. As pure computation proceeds, the pure continuation may grow. If an operation is encountered then the effect continuation is invoked. The current continuation pair (k, h) is packaged up as a *resumption* and passed to the current handler along with the operation and its argument. The effect continuation then either handles the operation, invoking the resump-

tion as appropriate, or forwards the operation to an outer handler. In the latter case, the resumption is modified to reinstate the dynamic stack of continuations when invoked.

The translations introduced in this subsection differ in how they represent stacks of pure and effect continuations, and how they represent resumptions. The first translation represents the stack of continuations using currying, and resumptions as functions (§4.2.1). Currying obstructs proper tail-recursion, so we move to an explicit representation of the stack (§4.2.2). Then, in order to avoid administrative reductions in our final higher-order one-pass translation we use an explicit representation of resumptions (§4.2.3).

4.2.1 Curried translation

Our first translation builds upon the CPS translation of Figure 5. The extension to operations and handlers is modest since currying conveniently lets us get away with a shift in interpretation: rather than accepting a single continuation, translated computation terms now accept an arbitrary even number of curried arguments representing the stack of pure and effect continuations. Thus, the translation of core constructs remain exactly the same as in Figure 5, where we imagine there being some number of extra continuation arguments that have been η-reduced. The translation of operations, handlers, and top-level programs is as follows.

$$
\begin{aligned}
[\![\mathbf{do}\ \ell\ V]\!] &= \lambda k.\lambda h.h\ (\ell\ \langle [\![V]\!], \lambda x.k\ x\ h\rangle)\\
[\![\mathbf{handle}\ M\ \mathbf{with}\ H]\!] &= [\![M]\!]\ [\![H^{\mathrm{ret}}]\!]\ [\![H^{\mathrm{ops}}]\!],\ \text{where}\\
[\![\{\mathbf{return}\ x \mapsto N\}]\!] &= \lambda x.\lambda h.[\![N]\!]\\
[\![\{\ell\ p\ r \mapsto N_\ell\}_{\ell\in\mathcal{L}}]\!] &= \lambda z.\mathbf{case}\ z\ \{(\ell\ \langle p, r\rangle \mapsto [\![N_\ell]\!])_{\ell\in\mathcal{L}}; y \mapsto M_{\mathrm{forward}}\}\\
M_{\mathrm{forward}} &= \lambda k'.\lambda h'.\mathbf{vmap}\,(\lambda\langle p, r\rangle\ k.k\,\langle p, \lambda x.r\ x\ k'\ h'\rangle)\ y\ h'\\
\top[\![M]\!] &= [\![M]\!]\ (\lambda x.\lambda h.x)\ (\lambda z.\mathbf{absurd}\ z)
\end{aligned}
$$

We extend our target calculus in order to implement forwarding. The computation term $\mathbf{vmap}\ U\ V\ W$ maps the function U over the body of the variant V and passes the result to continuation W. Its reduction rule is:

$$\text{U-VMap}\quad \mathbf{vmap}\,U\,(\ell\ V)\ W \rightsquigarrow U\ V\,(\lambda x\,k.k\,(\ell\,x))\ W$$

In an untyped setting **vmap** is easily definable. In Appendix B we sketch how to adapt our row type system to type the CPS translation with **vmap**.

The translation of $\mathbf{do}\ \ell\ V$ accepts a pure continuation k and an effect continuation h, which acts as a dispatcher function for encoded operations. Each operation is encoded as a value tagged with the name ℓ, where the value consists of a pair consisting of the parameter of the operation, and a resumption, which ensures that any subsequent operations are handled by the same effect continuation h.

The translation of $\mathbf{handle}\ M\ \mathbf{with}\ H$ invokes the translation of M with new pure and effect continuation arguments for the return and operation clauses of H. The translation of a return clause is a term which garbage collects the current effect continuation h. The translation of a set of operation clauses is a function which dispatches on encoded operations, and in the default case forwards to an outer handler. In the forwarding case, the resumption is extended by the parent continuation pair in order to reinstate the handler stack, thereby ensuring subsequent invocations of the same operation are handled uniformly.

Conceptually, top-level programs are enclosed by a top-level handler with an empty collection of operation clauses and an identity return clause. The CPS translation materialises this handler as the identity pure continuation (the K combinator), and an effect continuation that is never intended to be called.

There are two practical problems with this initial translation. First, it is not properly tail-recursive due to the curried representation of the continuation stack. We will rectify this using an uncurried representation in the next subsection. Second, it yields administrative redexes. We will rectify this with a higher-order one-pass translation in §4.3.

To illustrate both issues, consider the following example:

$$\begin{aligned}\top[\![\mathbf{return}\ \langle\rangle]\!] &= (\lambda k.k\,\langle\rangle)\,(\lambda x.\lambda h.x)\,(\lambda z.\mathbf{absurd}\ z)\\ &\rightsquigarrow ((\lambda x.\lambda h.x)\,\langle\rangle)\,(\lambda z.\mathbf{absurd}\ z) \rightsquigarrow (\lambda h.\langle\rangle)\,(\lambda z.\mathbf{absurd}\ z) \rightsquigarrow \langle\rangle\end{aligned}$$

The first reduction is administrative: it has nothing to do with the dynamic semantics of the original term and there is no reason not to eliminate it statically. The second and third reductions simulate handling **return** $\langle\rangle$ at the top level. The second reduction partially applies $\lambda x.\lambda h.x$ to $\langle\rangle$, which must return a value so that the third reduction can be applied: evaluation is not tail-recursive. The lack of tail-recursion is also apparent in our relaxation of fine-grain call-by-value in Figure 4: the function position of an application can be a computation, and the calculus makes use of evaluation contexts.

Remark We originally derived the curried CPS translation for effect handlers by composing a translation from effect handlers to delimited continuations [10] with a CPS translation for delimited continuations [22].

4.2.2 Uncurried translation: continuations as explicit stacks

Following Materzok and Biernacki [22] we uncurry our CPS translation in order to obtain a properly tail-recursive translation. The translation of return, let binding, operations, handlers, and top level programs is as follows.

$$\begin{aligned}[\![\mathbf{return}\ V]\!] &= \lambda(k :: ks).k\ [\![V]\!]\ ks\\ [\![\mathbf{let}\ x \leftarrow M\ \mathbf{in}\ N]\!] &= \lambda(k :: ks).[\![M]\!]((\lambda x\,ks.[\![N]\!](k :: ks)) :: ks)\\ [\![\mathbf{do}\ \ell\ V]\!] &= \lambda(k :: h :: ks).h\ (\ell\ \langle [\![V]\!], \lambda x\,ks.k\ x\ (h :: ks)\rangle)\ ks\\ [\![\mathbf{handle}\ M\ \mathbf{with}\ H]\!] &= \lambda ks.[\![M]\!]([\![H^{\mathrm{ret}}]\!] :: [\![H^{\mathrm{ops}}]\!] :: ks),\ \text{where}\\ [\![\{\mathbf{return}\ x \mapsto N\}]\!] &= \lambda x\,ks.\mathbf{let}\ (h :: ks') = ks\ \mathbf{in}\ [\![N]\!]\ ks\\ [\![\{\ell\ p\ r \mapsto N_\ell\}_{\ell\in\mathcal{L}}]\!] &= \lambda z\,ks.\mathbf{case}\ z\ \{(\ell\ \langle p, r\rangle \mapsto [\![N_\ell]\!]\ ks)_{\ell\in\mathcal{L}};\ y \mapsto M_{\mathrm{forward}}\}\\ M_{\mathrm{forward}} &= \mathbf{let}\ (k' :: h' :: ks') = ks\ \mathbf{in}\\ &\qquad \mathbf{vmap}\,(\lambda\langle p, r\rangle\,(k :: ks).k\,\langle p, \lambda x\,ks.r\,x\,(k' :: h' :: ks)\rangle\,ks)\,y\,ks'\\ \top[\![M]\!] &= [\![M]\!]\ ((\lambda x\,ks.x) :: (\lambda z\,ks.\mathbf{absurd}\ z) :: [])\end{aligned}$$

The other cases are as in Figure 5. The stacks of continuations are now explicitly represented as lists, where pure continuations and effect continuations occupy alternating positions. We now require lists in our target, which we implement using right-nested pairs and unit:

$$[] \equiv \langle\rangle \qquad V :: W \equiv \langle V, W\rangle \qquad U :: V :: W \equiv \langle U, \langle V, W\rangle\rangle$$

Similarly, we extend pattern matching in the standard way to accommodate lists.

Since we now use a list representation for the stacks of continuations, we need to modify the translations of all the constructs that manipulate continuations. For **return** and **let**, we extract the top continuation k and manipulate it analogously to the original translation in Figure 5. For **do**, we extract the top pure continuation k and effect continuation h and invoke h in the same way as the curried translation, except that we explicitly maintain the stack ks of additional continuations. The translation of **handle**, however, pushes a continuation pair onto the stack instead of supplying them as arguments. Handling of operations is the same as before, except for explicit passing of the ks. Forwarding now pattern matches on

the stack to extract the next continuation pair, rather than accepting them as arguments. As we now use lists to represent stacks, we must modify the reduction rule for **vmap**.

U-VMap $\quad \mathbf{vmap}\ U\,(\ell\,V)\ W \rightsquigarrow U\ V\,(\lambda x\,(k :: ks).k\,(\ell\,x)\ W)$

Proper tail recursion coincides with a refinement of the target syntax. Now applications are either of the form $V\ W$ or of the form $U\ V\ W$. We could also add a rule for applying a two argument lambda abstraction to two arguments at once and eliminate the U-Lift rule, but we defer spelling out the details until §4.3.

4.2.3 Resumptions as explicit reversed stacks

In our two CPS translations so far, resumptions have been represented as functions and forwarding has been implemented by function composition. In order to avoid administrative reductions due to function composition, we move to an explicit representation of resumptions as *reversed* stacks of pure and effect continuations. We convert these reversed stacks to actual functions on demand using a special **fun** binding with the following reduction rule.

U-Fun $\quad \mathbf{let}\ r = \mathbf{fun}\,(V_n :: \ldots :: V_1 :: []) \ \mathbf{in}\ N \rightsquigarrow N[(\lambda x\,ks.\,V_1\,x\,(V_2 :: \ldots :: V_n :: ks))/r]$

This reduction rule reverses the stack, pulls out the top continuation V_1, and appends the remainder onto the current stack ks. The stack representing a resumption and the remaining stack ks are reminiscent of the zipper data structure for representing cursors in lists [12]. Resumptions represent pointers into the stack of handlers. We use exactly the same representation in our abstract machine for effect handlers [11].

The translations of **do**, handling, and forwarding need to be modified to handle the change in representation of resumptions. The translation of **do** builds a resumption stack, handling uses the **fun** construct to convert the resumption stack into a function, and M_{forward} extends the resumption stack with the current continuation pair.

$$
\begin{aligned}
[\![\mathbf{do}\ \ell\ V]\!] &= \lambda k :: h :: ks.h\,(\ell\ \langle [\![V]\!], h :: k :: []\rangle)\ ks \\
[\![\{(\ell\ p\ r \mapsto N_\ell)_{\ell\in\mathcal{L}}\}]\!] &= \lambda z\,ks.\mathbf{case}\ z\ \{(\ell\ \langle p, s\rangle \mapsto \mathbf{let}\ r = \mathbf{fun}\ s\ \mathbf{in}\ [\![N_\ell]\!]\ ks)_{\ell\in\mathcal{L}};\ y \mapsto M_{\text{forward}}\} \\
M_{\text{forward}} &= \mathbf{let}\ (k' :: h' :: ks') = ks\ \mathbf{in} \\
&\quad\ \mathbf{vmap}\ (\lambda\langle p, r\rangle\ (k :: ks).k\,\langle p, h' :: k' :: r\rangle\ ks)\ y\ ks'
\end{aligned}
$$

4.3 A higher-order explicit stack translation

We now adapt our uncurried CPS translation to a higher-order one-pass CPS translation [6] that partially evaluates administrative redexes at translation time. Following Danvy and Nielsen [7], we adopt a two-level lambda calculus notation to distinguish between *static* lambda abstraction and application in the meta language and *dynamic* lambda abstraction and application in the target language. The idea is that redexes marked as static are reduced as part of the translation (at compile time), whereas those marked as dynamic are reduced at runtime. The CPS translation is given in Figure 6.

An overline denotes a static syntax constructor and an underline denotes a dynamic syntax constructor. In order to facilitate this notation we write application explicitly as an infix "at" symbol (@). We assume the meta language is pure and hence respects the usual β and η equivalences. We extend the overline and underline notation to distinguish between static and dynamic let bindings.

The reify operator $\downarrow$ maps static lists to dynamic ones and the reflect constructor $\uparrow$ allows dynamic lists to be treated as static. We use list pattern matching in the meta language.

$$
\begin{aligned}
(\overline{\lambda}(\kappa \mathbin{\overline{::}} \mathcal{P}).\mathcal{M})\ \overline{@}\ (V \mathbin{\overline{::}} VS) &= \overline{\mathbf{let}}\ \kappa = V\ \overline{\mathbf{in}}\ (\overline{\lambda}\mathcal{P}.\mathcal{M})\ \overline{@}\ VS \\
(\overline{\lambda}(\kappa \mathbin{\overline{::}} \mathcal{P}).\mathcal{M})\ \overline{@}\uparrow V &= \underline{\mathbf{let}}\ (k \mathbin{\underline{::}} ks) = V\ \underline{\mathbf{in}}\ \overline{\mathbf{let}}\ \kappa = k\ \overline{\mathbf{in}}\ (\overline{\lambda}\mathcal{P}.\mathcal{M})\ \overline{@}\uparrow ks
\end{aligned}
$$

Static patterns and static lists

$$\begin{array}{lll} \text{Static patterns} & \mathcal{P} & ::= \kappa :: \mathcal{P} \mid \kappa s \\ \text{Static lists} & VS & ::= V :: VS \mid \uparrow V \end{array}$$

Reify

$$\begin{aligned} \downarrow(V :: VS) &= V :: \downarrow VS \\ \downarrow\uparrow V &= V \end{aligned}$$

Values

$$\begin{aligned} [\![x]\!] &= x & [\![\Lambda\alpha.M]\!] &= \lambda z\, ks.[\![M]\!] \mathbin{@} \uparrow ks & [\![\langle \ell = V; W\rangle]\!] &= \langle \ell = [\![V]\!]; [\![W]\!]\rangle \\ [\![\lambda x.M]\!] &= \lambda x\, ks.[\![M]\!] \mathbin{@} \uparrow ks & [\![\langle\rangle]\!] &= \langle\rangle & [\![\ell\ V]\!] &= \ell\ [\![V]\!] \end{aligned}$$

Computations

$$\begin{aligned} [\![V\ W]\!] &= \lambda\kappa s.[\![V]\!] \mathbin{@} [\![W]\!] \mathbin{@} \downarrow\kappa s \\ [\![V\ T]\!] &= \lambda\kappa s.[\![V]\!] \mathbin{@} \langle\rangle \mathbin{@} \downarrow\kappa s \\ [\![\mathbf{let}\ \langle \ell = x; y\rangle = V\ \mathbf{in}\ N]\!] &= \lambda\kappa s.\mathbf{let}\ \langle \ell = x; y\rangle = [\![V]\!]\ \mathbf{in}\ [\![N]\!] \mathbin{@} \kappa s \\ [\![\mathbf{case}\ V\ \{\ell\ x \mapsto M; y \mapsto N\}]\!] &= \lambda\kappa s.\mathbf{case}\ [\![V]\!]\ \{\ell\ x \mapsto [\![M]\!] \mathbin{@} \kappa s; y \mapsto [\![N]\!] \mathbin{@} \kappa s\} \\ [\![\mathbf{absurd}\ V]\!] &= \lambda\kappa s.\mathbf{absurd}\ [\![V]\!] \\ [\![\mathbf{return}\ V]\!] &= \lambda\kappa :: \kappa s.\kappa \mathbin{@} [\![V]\!] \mathbin{@} \downarrow\kappa s \\ [\![\mathbf{let}\ x \leftarrow M\ \mathbf{in}\ N]\!] &= \lambda\kappa :: \kappa s.[\![M]\!] \mathbin{@} ((\lambda x\, ks.[\![N]\!] \mathbin{@} (\kappa :: \uparrow ks)) :: \kappa s) \\ [\![\mathbf{do}\ \ell\ V]\!] &= \lambda\kappa :: \eta :: \kappa s.\eta \mathbin{@} (\ell\ \langle [\![V]\!], \eta :: \kappa :: []\rangle) \mathbin{@} \downarrow\kappa s \\ [\![\mathbf{handle}\ M\ \mathbf{with}\ H]\!] &= \lambda\kappa s.[\![M]\!] \mathbin{@} ([\![H^{\mathrm{ret}}]\!] :: [\![H^{\mathrm{ops}}]\!] :: \kappa s),\ \text{where} \end{aligned}$$

$$\begin{aligned} [\![\{\mathbf{return}\ x \mapsto N\}]\!] &= \lambda x\, ks.\mathbf{let}\ (h :: ks') = ks\ \mathbf{in}\ [\![N]\!] \mathbin{@} \uparrow ks' \\ [\![\{(\ell\ p\ r \mapsto N_\ell)_{\ell\in\mathcal{L}}\}]\!] &= \lambda z\, ks.\mathbf{case}\ z\ \{(\ell\ \langle p, s\rangle \mapsto \mathbf{let}\ r = \mathbf{fun}\ s\ \mathbf{in}\ [\![N_\ell]\!] \mathbin{@} \uparrow ks)_{\ell\in\mathcal{L}};\ y \mapsto M_{\mathrm{forward}}\} \\ M_{\mathrm{forward}} &= \mathbf{let}\ (k' :: h' :: ks') = ks\ \mathbf{in} \\ &\quad\ \mathbf{vmap}\ (\lambda\langle p, s\rangle\ (k :: ks).k\ \langle p, h' :: k' :: s\rangle\ ks)\ y\ ks' \end{aligned}$$

Top level program

$$\top[\![M]\!] = [\![M]\!] \mathbin{@} ((\lambda x\, ks.x) :: (\lambda z\, ks.\mathbf{absurd}\ z) :: \uparrow[])$$

Figure 6 Higher-order uncurried CPS translation of $\lambda^{\rho}_{\mathrm{eff}}$

Here we let $\mathcal{M}$ range over meta language expressions.

Now the target calculus is refined so that all lambda abstractions and applications take two arguments, the U-Lift rule is removed, and the U-App rule is replaced by the following reduction rule:

U-AppTwo $\quad (\lambda x\, ks.\, M) \mathbin{@} V \mathbin{@} W \rightsquigarrow M[V/x, W/ks]$

We add an extra dummy argument to the translation of type lambda abstractions and applications in order to ensure that all dynamic functions take exactly two arguments. The single argument lambdas and applications from the first-order uncurried translation are still present, but now they are all static.

In order to reason about the behaviour of the S-Handle-op rule, which is defined in terms of an evaluation context, we extend the CPS translation to evaluation contexts:

$$\begin{aligned} [\![[\ ]]\!] &= \lambda\kappa s.\ \kappa s \\ [\![\mathbf{let}\ x \leftarrow \mathcal{E}\ \mathbf{in}\ N]\!] &= \lambda\kappa :: \kappa s.\ [\![\mathcal{E}]\!] \mathbin{@} ((\lambda x\, ks.[\![N]\!] \mathbin{@} (k :: \uparrow ks)) :: \kappa s) \\ [\![\mathbf{handle}\ \mathcal{E}\ \mathbf{with}\ H]\!] &= \lambda\kappa s.\ [\![\mathcal{E}]\!] \mathbin{@} ([\![H^{\mathrm{ret}}]\!] :: [\![H^{\mathrm{ops}}]\!] :: \kappa s) \end{aligned}$$

The following lemma is the characteristic property of the CPS translation on evaluation contexts. This allows us to focus on the computation contained within an evaluation context.

▶ **Lemma 3** (Decomposition).

$$[\![\mathcal{E}[M]]\!] \mathbin{@} (V :: VS) = [\![M]\!] \mathbin{@} ([\![\mathcal{E}]\!] \mathbin{@} (V :: VS))$$

Though we have eliminated the static administrative redexes, we are still left with one form of administrative redex that cannot be eliminated statically because it only appears

at run-time. These arise from pattern matching against a reified stack of continuations and are given by the U-SplitList rule.

U-SplitList $\quad \mathbf{let}\ (k :: ks) = V :: W\ \mathbf{in}\ M \rightsquigarrow M[V/k, W/ks]$

This is isomorphic to the U-Split rule, but we now treat lists and U-SplitList as distinct from pairs, unit, and U-Split in the higher-order translation so that we can properly account for administrative reduction. We write $\rightsquigarrow_{\mathrm{a}}$ for the compatible closure of U-SplitList.

By definition, $\downarrow\uparrow V = V$, but we also need to reason about the inverse composition. The proof is by induction on the structure of M.

▶ **Lemma 4** (Reflect after reify). $[\![M]\!] \mathbin{@} (V_1 :: \ldots V_n :: \uparrow\downarrow VS) \rightsquigarrow_a^* [\![M]\!] \mathbin{@} (V_1 :: \ldots V_n :: VS)$

We next observe that the CPS translation simulates forwarding.

▶ **Lemma 5** (Forwarding). *If $\ell \notin dom(H_1)$ then:*

$[\![H_1^{\mathrm{ops}}]\!] \mathbin{@} \ell \langle U, V\rangle \mathbin{@} (V_2 :: [\![H_2^{\mathrm{ops}}]\!] :: W) \rightsquigarrow^+ [\![H_2^{\mathrm{ops}}]\!] \mathbin{@} \ell \langle U, [\![H_1^{\mathrm{ops}}]\!] :: V_1 :: V\rangle \mathbin{@} W$

Now we show that the translation simulates the S-Handle-op rule.

▶ **Lemma 6** (Handling). *If $\ell \notin BL(\mathcal{E})$ and $H^\ell = \{\ell\ p\ r \mapsto N_\ell\}$ then:*

$[\![\mathbf{do}\ \ell\ V]\!] \mathbin{@} ([\![\mathcal{E}]\!] \mathbin{@} ([\![H^{\mathrm{ret}}]\!] :: [\![H^{\mathrm{ops}}]\!] :: VS)) \rightsquigarrow^+ \rightsquigarrow_a^*$
$([\![N_\ell]\!] \mathbin{@} VS)[[\![V]\!]/p, (\lambda y\, ks.[\![\mathbf{return}\ y]\!] \mathbin{@} ([\![\mathcal{E}]\!] \mathbin{@} ([\![H^{\mathrm{ret}}]\!] :: [\![H^{\mathrm{ops}}]\!] :: \uparrow ks)))/r]$

This lemma follows from Lemmas 3, 4, and 5. We now turn to our main result which is a simulation result in style of Plotkin [24]. The theorem shows that the only extra behaviour exhibited by a translated term is the bureaucracy of deconstructing the continuation stack.

▶ **Theorem 7** (Simulation). *If $M \rightsquigarrow N$ then $\top[\![M]\!] \rightsquigarrow^+ \rightsquigarrow_a^* \top[\![N]\!]$.*

The proof is by case analysis on the reduction relation using Lemmas 3–6. The S-Handle-op case follows from Lemma 6.

In common with most CPS translations, full abstraction does not hold. However, as our semantics is deterministic it is straightforward to show a backward simulation result.

▶ **Corollary 8** (Backwards simulation). *If $\top[\![M]\!] \rightsquigarrow^+ \rightsquigarrow_a^* V$ then there exists W such that $M \rightsquigarrow^* W$ and $\top[\![W]\!] = V$.*

4.4 Shallow handlers

Shallow handlers [13] differ from deep handlers in that when handling an operation the former does not reinvoke the handler inside the resumption. The typing rules and operational semantics for shallow handlers are as follows.

T-Shallow-Handle
$$\frac{\Delta; \Gamma \vdash M : C \qquad \Delta; \Gamma \vdash H : C \Rightarrow^\dagger D}{\Delta; \Gamma \vdash \mathbf{handle}^\dagger\ M\ \mathbf{with}\ H : D}$$

T-Shallow-Handler
$$\frac{\begin{array}{c} C = A!\{(\ell_i : A_i \to B_i)_i; R\} \qquad D = B!\{(\ell_i : P_i)_i; R\} \\ H = \{\mathbf{return}\ x \mapsto M\} \uplus \{\ell_i\ y\ r \mapsto N_{\ell_i}\}_i \\ \Delta; \Gamma, x : A \vdash M : D \\ [\Delta; \Gamma, y : A_i, r : B_i \to C \vdash N_{\ell_i} : D]_i \end{array}}{\Delta; \Gamma \vdash H : C \Rightarrow^\dagger D}$$

S-Shallow-Ret $\quad \mathbf{handle}^\dagger\ (\mathbf{return}\ V)\ \mathbf{with}\ H \rightsquigarrow N[V/x]$, where $H^{\mathrm{ret}} = \{\mathbf{return}\ x \mapsto N\}$

S-Shallow-Op $\quad \mathbf{handle}^\dagger\ \mathcal{E}[\mathbf{do}\ \ell\ V]\ \mathbf{with}\ H \rightsquigarrow N[V/x, (\lambda y.\, \mathcal{E}[\mathbf{return}\ y])/r]$,
where $\ell \notin BL(\mathcal{E})$ and $H^\ell = \{\ell\ x\ r \mapsto N\}$

We write a dagger (†) superscript to distinguish shallow handlers from deep handlers. We adapt our higher-order CPS translation to accommodate both shallow and deep handlers.

$$\left.\begin{aligned}
[\![\mathbf{do}\ \ell\ V]\!] &= \overline{\lambda}\kappa\, \overline{::}\, \eta\, \overline{::}\, \kappa s.\eta\ \underline{@}\ (\ell\ \langle [\![V]\!], \kappa\ \underline{::}\ []\rangle)\ \underline{@}\ \downarrow\kappa s \\
[\![\mathbf{handle}\ M\ \mathbf{with}\ H]\!] &= \overline{\lambda}\kappa s.[\![M]\!]\ \overline{@}\ ([\![H^{\mathrm{ret}}]\!]\ \overline{::}\ [\![H^{\mathrm{ops}}]\!]\ \overline{::}\ \kappa s) \\
[\![\mathbf{handle}^{\dagger}\ M\ \mathbf{with}\ H]\!] &= \overline{\lambda}\kappa s.[\![M]\!]\ \overline{@}\ ([\![H^{\mathrm{ret}}]\!]\ \overline{::}\ [\![H^{\mathrm{ops}}]\!]^{\dagger}\ \overline{::}\ \kappa s)
\end{aligned}\right\} \text{where}$$

$$\begin{aligned}
[\![\{\mathbf{return}\ x \mapsto N\}]\!] &= \underline{\lambda} x\ ks.\underline{\mathbf{let}}\ (h\ \underline{::}\ ks') = ks\ \underline{\mathbf{in}}\ [\![N]\!]\ \overline{@}\uparrow ks' \\
[\![\{(\ell\ p\ r \mapsto N_\ell)_{\ell\in\mathcal{L}}\}]\!] &= \mathbf{rec}\, h\, z\, ks.\mathbf{case}\ z\ \{(\ell\ \langle p, s\rangle \mapsto \underline{\mathbf{let}}\ r = \mathbf{fun}\,(h\ \underline{::}\ s)\ \underline{\mathbf{in}}\ [\![N_\ell]\!]\ \overline{@}\uparrow ks)_{\ell\in\mathcal{L}} \\
&\qquad\qquad y \mapsto M_{\mathrm{forward}}\} \\
[\![\{(\ell\ p\ r \mapsto N_\ell)_{\ell\in\mathcal{L}}\}]\!]^{\dagger} &= \mathbf{rec}\, h\, z\, ks.\mathbf{case}\ z\ \{(\ell\ \langle p, s\rangle \mapsto \underline{\mathbf{let}}\ r = \mathbf{fun}\,(V_{\mathrm{id}}\ \underline{::}\ s)\ \underline{\mathbf{in}}\ [\![N_\ell]\!]\ \overline{@}\uparrow ks)_{\ell\in\mathcal{L}} \\
&\qquad\qquad y \mapsto M_{\mathrm{forward}}\} \\
M_{\mathrm{forward}} &= \underline{\mathbf{let}}\ (k'\ \underline{::}\ h'\ \underline{::}\ ks') = ks\ \underline{\mathbf{in}} \\
&\qquad \mathbf{vmap}\ (\underline{\lambda}\langle p, s\rangle\ (k\ \underline{::}\ ks).k\,\langle p, h'\ \underline{::}\ k'\ \underline{::}\ h\ \underline{::}\ s\rangle\ ks)\ y\ ks' \\
V_{\mathrm{id}} &= \mathbf{rec}\, h\, y\, ks.M_{\mathrm{forward}}
\end{aligned}$$

For deep handlers, the current effect continuation is now added inside the translation of the operation clauses rather than the translation of the operation. This necessitates making the translation of operation clauses recursive, which we do using a recursion operator.

U-Rec $\quad (\mathbf{rec}\, f\, x\, ks.M)\ V\ W \rightsquigarrow M[(\mathbf{rec}\, f\, x.M)/f, V/x, W/ks]$

In order to translate a shallow handler we insert an identity effect continuation in place of the current effect continuation.

4.5 Exceptions and their handlers as separate constructs

Our core calculus $\lambda^{\rho}_{\mathrm{eff}}$ (and also Links) does not have special support for exceptions and their handlers. Indeed, exceptions are a special case of effects where the operations return an uninhabited type similar to the `Fail` operation from §2. On the other hand, Multicore OCaml maintains effects separate from exceptions not only for backwards compatibility but also due to the fact that exceptions in Multicore OCaml are cheaper than effects. Multicore OCaml relies on runtime support for stack manipulation in order to implement effect handlers, where raising an exception need only unwind the stack and need not capture the continuation. Thus, there is benefit in separating exceptions from effects; computations which raise exceptions but do not perform other effects may be retained in direct-style in a selective CPS translation. We can model this by extending handlers with exception clauses.

Handlers $\quad H ::= \{\mathbf{return}\ x \mapsto M\} \mid \{\ell\ x\ r \mapsto M\} \uplus H \mid \{\ell\ x \mapsto M\} \uplus H$

S-Handle-Ex $\quad \mathbf{handle}\ \mathcal{E}[\mathbf{raise}\ \ell\ V]\ \mathbf{with}\ H \rightsquigarrow N[V/x]$, where $\ell \notin BL(\mathcal{E})$ and $H^{\ell} = \{\ell\ x \mapsto N\}$

Exception clauses lack a resumption. The CPS translation can now be adapted to maintain a stack of triples (pure continuation, exception continuation, effect continuation).

5 Implementation

In this section, we briefly outline our experiences of implementing the CPS translations described in §4.

A first prototype (available at `https://github.com/Armael/lam`) was implemented by the fourth author and Armaël Guéneau for Multicore OCaml, which makes a distinction between exception handlers and general effect handlers. Thus, this implementation is based on the approach of §4.5. It uses a curried CPS translation which is not properly tail-recursive.

The Links implementation (available at `https://github.com/links-lang/links`) relies on the higher-order CPS translation of §4.3. The Links client-side JavaScript backend has

long used a higher-order CPS translation, relying on a trampoline for supporting lightweight concurrency and responsive user-interfaces. As it is based on a trampoline, the stack is periodically discarded and it is essential that the CPS translation be properly tail-recursive. Initially we attempted to implement a higher-order curried translation. We then realised that it is unclear whether it is even possible to define a higher-order curried translation for effect handlers, so we began implementing a first-order curried translation. It quickly became apparent that this approach could not work given the need to be properly tail-recursive. At this point we changed tack and successfully implemented a properly tail-recursive higher-order uncurried translation along the lines of the one described in §4.3.

6 Conclusions and future work

We have carried out a comprehensive study of CPS translations for effect handlers. We have presented the first full CPS translations for effect handlers: our translations go all the way to lambda calculus without relying on a special low-level handling construct as Leijen [17] does. We began with a standard first-order call-by-value CPS translation, which we extended to support effect handlers. We then refined the first-order translation by uncurrying it in order to yield a properly tail-recursive translation, and by adapting it to a higher-order one-pass translation that statically eliminates administrative redexes. We proved that the higher-order uncurried CPS translation simulates reduction in the source language. In addition, we have also shown how to adapt the translations to support shallow handlers.

The server backend for Links [11] is based on an extension of a CEK machine to support handlers. There are clear connections between their abstract machine and our higher-order CPS translation. In future work we intend to make these connections precise.

While our translations apply to unary handlers, we would like to investigate how to adapt them to *multi handlers* which handle multiple computations at simultaneously [20].

Many useful effect handlers do not use resumptions more than once. The Multicore OCaml compiler takes advantage of supporting only affine use of resumptions, by default, to obtain remarkably strong performance. However, Multicore OCaml makes use of its own custom stack implementation, whereas for certain backends (notably JavaScript) that luxury is not available. We would like to explore linear and affine variants of our CPS translations, mediated by a suitable substructural type system, to see if we can obtain similar benefits. Another direction we intend to explore in this setting is the use of JavaScript's existing generator abstraction for implementing linear and affine handlers. We also intend to perform a quantitative study of implementation strategies for effect handlers.

Acknowledgements. We thank the anonymous reviewers for their insightful comments. Daniel Hillerström was supported by EPSRC grant EP/L01503X/1 (CDT in Pervasive Parallelism). Sam Lindley was supported by EPSRC grant EP/K034413/1 (A Basis for Concurrency and Distribution). KC Sivaramakrishnan was supported by a Research Fellowship from the Royal Commission for the Exhibition of 1851.

References

1 Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.
2 Andrej Bauer and Matija Pretnar. Programming with algebraic effects and handlers. *J. Log. Algebr. Meth. Program.*, 84(1):108–123, 2015.
3 Bernard Berthomieu and Camille le Moniès de Sagazan. A calculus of tagged types, with applications to process languages. In *Workshop on Types for Program Analysis*, 1995.

4 Edwin Brady. Programming and reasoning with algebraic effects and dependent types. In *ICFP*, pages 133–144. ACM, 2013.
5 Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: Web programming without tiers. In *FMCO*, volume 4709 of *LNCS*, pages 266–296. Springer, 2006.
6 Olivier Danvy and Andrzej Filinski. Abstracting control. In *LISP and Functional Programming*, pages 151–160, 1990.
7 Olivier Danvy and Lasse R. Nielsen. A first-order one-pass CPS transformation. *Theor. Comput. Sci.*, 308(1-3):239–257, 2003.
8 Stephen Dolan, Leo White, KC Sivaramakrishnan, Jeremy Yallop, and Anil Madhavapeddy. Effective concurrency through algebraic effects. OCaml Workshop, 2015.
9 Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *PLDI*, pages 237–247. ACM, 1993.
10 Yannick Forster, Ohad Kammar, Sam Lindley, and Matija Pretnar. On the expressive power of user-defined effects: Effect handlers, monadic reflection, delimited control. *Proc. ACM Program. Lang.*, 1(ICFP), September 2017.
11 Daniel Hillerström and Sam Lindley. Liberating effects with rows and handlers. In *TyDe@ICFP*, pages 15–27. ACM, 2016.
12 Gérard P. Huet. The zipper. *J. Funct. Program.*, 7(5):549–554, 1997.
13 Ohad Kammar, Sam Lindley, and Nicolas Oury. Handlers in action. In *ICFP*, pages 145–158. ACM, 2013.
14 Andrew Kennedy. Compiling with continuations, continued. In *ICFP*, pages 177–190. ACM, 2007.
15 Oleg Kiselyov and Hiromi Ishii. Freer monads, more extensible effects. In *Haskell*, pages 94–105. ACM, 2015.
16 Oleg Kiselyov and KC Sivaramakrishnan. Eff directly in OCaml. ML Workshop, 2016.
17 Daan Leijen. Type directed compilation of row-typed algebraic effects. In *POPL*, pages 486–499. ACM, 2017.
18 Paul Blain Levy, John Power, and Hayo Thielecke. Modelling environments in call-by-value programming languages. *Inf. Comput.*, 185(2):182–210, 2003.
19 Sam Lindley and James Cheney. Row-based effect types for database integration. In *TLDI*, pages 91–102. ACM, 2012.
20 Sam Lindley, Conor McBride, and Craig McLaughlin. Do be do be do. In *POPL*, pages 500–514. ACM, 2017.
21 Marek Materzok and Dariusz Biernacki. Subtyping delimited continuations. In *ICFP*, pages 81–93. ACM, 2011.
22 Marek Materzok and Dariusz Biernacki. A dynamic interpretation of the CPS hierarchy. In *APLAS*, volume 7705 of *LNCS*, pages 296–311. Springer, 2012.
23 Eugenio Moggi. Notions of computation and monads. *Inf. Comput.*, 93(1):55–92, 1991.
24 Gordon D. Plotkin. Call-by-name, call-by-value and the lambda-calculus. *Theor. Comput. Sci.*, 1(2):125–159, 1975.
25 Gordon D. Plotkin and John Power. Adequacy for algebraic effects. In *FoSSaCS*, volume 2030 of *LNCS*, pages 1–24. Springer, 2001.
26 Gordon D. Plotkin and Matija Pretnar. Handling algebraic effects. *Logical Methods in Computer Science*, 9(4), 2013.
27 Matija Pretnar. An introduction to algebraic effects and handlers. *Electr. Notes Theor. Comput. Sci.*, 319:19–35, 2015. Invited tutorial paper.
28 Didier Remy. Syntactic theories and the algebra of record terms. Technical Report RR-1869, INRIA, 1993.
29 Philip Wadler. The essence of functional programming. In *POPL*, pages 1–14. ACM, 1992.

$$\textsc{TyVar}\ \frac{}{\Delta, \alpha : K \vdash \alpha : K} \qquad \textsc{Comp}\ \frac{\Delta \vdash A : \mathsf{Type} \quad \Delta \vdash E : \mathsf{Effect}}{\Delta \vdash A!E : \mathsf{Comp}} \qquad \textsc{Fun}\ \frac{\Delta \vdash A : \mathsf{Type} \quad \Delta \vdash C : \mathsf{Comp}}{\Delta \vdash A \to C : \mathsf{Type}} \qquad \textsc{Forall}\ \frac{\Delta, \alpha : K \vdash C : \mathsf{Comp}}{\Delta \vdash \forall \alpha^K.\, C : \mathsf{Type}}$$

$$\textsc{Record}\ \frac{\Delta \vdash R : \mathsf{Row}_\emptyset}{\Delta \vdash \langle R \rangle : \mathsf{Type}} \qquad \textsc{Variant}\ \frac{\Delta \vdash R : \mathsf{Row}_\emptyset}{\Delta \vdash [R] : \mathsf{Type}} \qquad \textsc{Effect}\ \frac{\Delta \vdash R : \mathsf{Row}_\emptyset}{\Delta \vdash \{R\} : \mathsf{Effect}}$$

$$\textsc{Present}\ \frac{\Delta \vdash A : \mathsf{Type}}{\Delta \vdash \mathsf{Pre}(A) : \mathsf{Presence}} \qquad \textsc{Absent}\ \frac{}{\Delta \vdash \mathsf{Abs} : \mathsf{Presence}} \qquad \textsc{EmptyRow}\ \frac{}{\Delta \vdash \cdot : \mathsf{Row}_{\mathcal{L}}} \qquad \textsc{ExtendRow}\ \frac{\Delta \vdash P : \mathsf{Presence} \quad \Delta \vdash R : \mathsf{Row}_{\mathcal{L} \uplus \{\ell\}}}{\Delta \vdash \ell : P; R : \mathsf{Row}_{\mathcal{L}}}$$

$$\textsc{Handler}\ \frac{\Delta \vdash C : \mathsf{Comp} \qquad \Delta \vdash D : \mathsf{Comp}}{\Delta \vdash C \Rightarrow D : \mathsf{Handler}}$$

Figure 7 Kinding rules for $\lambda^{\rho}_{\text{eff}}$

Values

$$\text{T-Var}\ \frac{x : A \in \Gamma}{\Delta; \Gamma \vdash x : A} \qquad \text{T-Lam}\ \frac{\Delta; \Gamma, x : A \vdash M : C}{\Delta; \Gamma \vdash \lambda x^A.\, M : A \to C} \qquad \text{T-PolyLam}\ \frac{\Delta, \alpha : K; \Gamma \vdash M : C \qquad \alpha \notin FTV(\Gamma)}{\Delta; \Gamma \vdash \Lambda \alpha^K.\, M : \forall \alpha^K.\, C}$$

$$\text{T-Unit}\ \frac{}{\Delta; \Gamma \vdash \langle\rangle : \langle\rangle} \qquad \text{T-Extend}\ \frac{\Delta; \Gamma \vdash V : A \qquad \Delta; \Gamma \vdash W : \langle \ell : \mathsf{Abs}; R \rangle}{\Delta; \Gamma \vdash \langle \ell = V;\, W \rangle : \langle \ell : \mathsf{Pre}(A); R \rangle} \qquad \text{T-Inject}\ \frac{\Delta; \Gamma \vdash V : A}{\Delta; \Gamma \vdash (\ell\ V)^R : [\ell : \mathsf{Pre}(A); R]}$$

Computations

$$\text{T-App}\ \frac{\Delta; \Gamma \vdash V : A \to C \qquad \Delta; \Gamma \vdash W : A}{\Delta; \Gamma \vdash V\ W : C} \qquad \text{T-PolyApp}\ \frac{\Delta; \Gamma \vdash V : \forall \alpha^K.\, C \qquad \Delta \vdash T : K}{\Delta; \Gamma \vdash V\ T : C[T/\alpha]} \qquad \text{T-Split}\ \frac{\Delta; \Gamma \vdash V : \langle \ell : \mathsf{Pre}(A); R \rangle \qquad \Delta; \Gamma, x : A, y : \langle \ell : \mathsf{Abs}; R \rangle \vdash N : C}{\Delta; \Gamma \vdash \mathbf{let}\ \langle \ell = x; y \rangle = V\ \mathbf{in}\ N : C}$$

$$\text{T-Case}\ \frac{\Delta; \Gamma \vdash V : [\ell : \mathsf{Pre}(A); R] \qquad \Delta; \Gamma, x : A \vdash M : C \qquad \Delta; \Gamma, y : [\ell : \mathsf{Abs}; R] \vdash N : C}{\Delta; \Gamma \vdash \mathbf{case}\ V\{\ell\ x \mapsto M; y \mapsto N\} : C} \qquad \text{T-Absurd}\ \frac{\Delta; \Gamma \vdash V : []}{\Delta; \Gamma \vdash \mathbf{absurd}^C\ V : C}$$

$$\text{T-Return}\ \frac{\Delta; \Gamma \vdash V : A}{\Delta; \Gamma \vdash \mathbf{return}\ V : A!E} \qquad \text{T-Let}\ \frac{\Delta; \Gamma \vdash M : A!E \qquad \Delta; \Gamma, x : A \vdash N : B!E}{\Delta; \Gamma \vdash \mathbf{let}\ x \leftarrow M\ \mathbf{in}\ N : B!E}$$

$$\text{T-Do}\ \frac{\Delta; \Gamma \vdash V : A \qquad E = \{\ell : A \to B; R\}}{\Delta; \Gamma \vdash (\mathbf{do}\ \ell\ V)^E : B!E} \qquad \text{T-Handle}\ \frac{\Delta; \Gamma \vdash M : C \qquad \Delta; \Gamma \vdash H : C \Rightarrow D}{\Delta; \Gamma \vdash \mathbf{handle}\ M\ \mathbf{with}\ H : D}$$

Handlers

$$\text{T-Handler}\ \frac{\begin{array}{c} C = A!\{(\ell_i : A_i \to B_i)_i; R\} \qquad D = B!\{(\ell_i : P_i)_i; R\} \qquad H = \{\mathbf{return}\ x \mapsto M\} \uplus \{\ell_i\ y\ r \mapsto N_{\ell_i}\}_i \\ [\Delta; \Gamma, y : A_i, r : B_i \to D \vdash N_{\ell_i} : D]_i \qquad \Delta; \Gamma, x : A \vdash M : D \end{array}}{\Delta; \Gamma \vdash H : C \Rightarrow D}$$

Figure 8 Typing rules for $\lambda^{\rho}_{\text{eff}}$

A Kinding and typing rules for $\lambda^{\rho}_{\text{eff}}$

The kinding rules for $\lambda^{\rho}_{\text{eff}}$ are given in Figure 7 and the typing rules are given in Figure 8.

B Typed CPS translations

Given a suitably polymorphic target calculus, we can define a typed CPS translation for $\lambda^{\rho}_{\text{eff}}$. Of course, we can use the polymorphism of the target calculus to model polymorphism (including row polymorphism) in source terms. More importantly, polymorphism may be used to abstract over the return type of effectful computations.

Operations may be encoded with polymorphic variants and handlers with case expressions. Alas, our existing row type system is not quite expressive enough to encode generic forwarding in this setting, so let us begin by considering the restriction of $\lambda^{\rho}_{\text{eff}}$ without forwarding, where we assume all effect rows are closed and all handlers are complete (i.e. include operation clauses for all operations in their types).

B.1 Handlers without forwarding

Figure 9 gives the CPS translation of $\lambda^{\rho}_{\text{eff}}$ without forwarding. The image of the translation lies within the pure fragment of $\lambda^{\rho}_{\text{eff}}$. The translations on value types and values are omitted as they are entirely homomorphic. If we erase all types then we obtain the translation of §4.2.1 with two differences. First, the former translation η-expands the body of a type lambda in order to ensure that it is a value (this is a superficial difference). There is no need to do that here as type lambdas are values. Second, the former translation performs forwarding, which we address in §B.2.

As the translation on terms is in essence one we have already seen, the main interest is in the translation on types. The body of the translation of a computation type $\mathcal{C}_{A,E}$ is exactly that of the continuation monad instantiated with return type $(\llbracket E \rrbracket\gamma \to \gamma) \to \gamma$. The argument to this function is the type of a handler continuation whose eventual return type is γ. By abstracting over the type of γ we can allow the handler stack to grow dynamically as necessary. This polymorphism accomplishes a similar purpose to Materzok and Biernacki's subtyping system for delimited continuations [21], which allows arbitrarily nested delimited continuations to be typed. We can witness the instantiation of the return type in the translation of a handler where the translation of the computation being handled $\llbracket M \rrbracket$ is applied to $\mathcal{C}_{B,E'}$. Polymorphism allows us to dynamically construct an arbitrarily deep stack of continuation monads, each carrying its own handler continuation.

Effect types type operations, which are encoded as variants pairing up a value with a continuation. The translation on effect types is parameterised by return type C.

▶ **Theorem 9** (Type preservation). *If* $\Delta; \Gamma \vdash M : A!E$ *then* $\llbracket\Delta\rrbracket; \llbracket\Gamma\rrbracket \vdash \llbracket M \rrbracket : \llbracket A!B \rrbracket$.

B.2 Forwarding and shapes

In order to encode forwarding we need to be able to parametrically specify what a default case does. Given a default variable y, we know that y is of the form $\ell\,\langle V, W\rangle$ where $V : A$ and $W : B \to C$ for some unknown types A and B and a fixed return type C. From y we need to produce a new value $\ell\,\langle V, W'\rangle$ where $W' : B \to C'$ and C' is a new return type. We can do so if we can define a typed version of the **vmap** operation.

The extension we propose to our row type system is to allow a row type to be given a *shape*, also known as a *type scheme*, which constrains the form of the ordinary types it contains. For instance, the shape of a row for a variant representing an operation at return type C is $\alpha^{\mathsf{Type}}\,\beta^{\mathsf{Type}}.\langle\alpha, \beta \to C\rangle$ and the shape of an unconstrained row, that is the shape of all rows in plain $\lambda^{\rho}_{\text{eff}}$, is $\alpha^{\mathsf{Type}}.\alpha$.

Computation types

$$\mathcal{C}_{A,E} = (\llbracket A \rrbracket \to (\llbracket E \rrbracket \gamma \to \gamma) \to \gamma) \to (\llbracket E \rrbracket \gamma \to \gamma) \to \gamma$$
$$\llbracket A ! E \rrbracket = \forall \gamma^{\mathsf{Type}}.\mathcal{C}_{A,E}$$

Effect types

$$\llbracket \{\ell : \llbracket A_\ell \rrbracket \to \llbracket B_\ell \rrbracket\}_{\ell \in \mathcal{L}} \rrbracket C = [\ell : \langle \llbracket A_\ell \rrbracket, \llbracket B_\ell \rrbracket \to C \rangle]_{\ell \in \mathcal{L}}$$

Computations (the other cases are homomorphic)

$$\begin{aligned}
\llbracket (\mathbf{return}\ V)^{A!E} \rrbracket &= \Lambda\gamma^{\mathsf{Type}}.\lambda k^{\llbracket A \rrbracket \to (\llbracket E \rrbracket \gamma \to \gamma) \to \gamma}.k\ \llbracket V \rrbracket \\
\llbracket \mathbf{let}\ x \leftarrow M^{A!E}\ \mathbf{in}\ N^{B!E} \rrbracket &= \Lambda\gamma^{\mathsf{Type}}.\lambda k^{\llbracket B \rrbracket \to (\llbracket E \rrbracket \gamma \to \gamma) \to \gamma}.\llbracket M \rrbracket (\lambda x^{\llbracket B \rrbracket}.\llbracket N \rrbracket k) \\
\llbracket (\mathbf{do}\ \ell\ V)^{E} \rrbracket &= \Lambda\gamma^{\mathsf{Type}}.\lambda k^{\llbracket B \rrbracket \to (\llbracket E \rrbracket \gamma \to \gamma) \to \gamma}.\lambda h^{\llbracket E \rrbracket \gamma \to \gamma}.h\ (\ell\ \langle \llbracket V \rrbracket, \lambda x^{\llbracket B \rrbracket}.k\ x\ h \rangle)^{\llbracket E \rrbracket \gamma} \\
&\qquad \text{where } \ell : A \to B \in E \\
\llbracket \mathbf{handle}\ M\ \mathbf{with}\ H^{A!E \Rightarrow B!E'} \rrbracket &= \Lambda\gamma^{\mathsf{Type}}.\llbracket M \rrbracket\ \mathcal{C}_{B,E'}\ \llbracket H^{\mathrm{ret}} \rrbracket^{A!E \Rightarrow B!E'}\ \llbracket H^{\mathrm{ops}} \rrbracket^{A!E \Rightarrow B!E'}, \text{ where} \\
\llbracket \{\mathbf{return}\ x \mapsto N_{\mathrm{ret}}\} \rrbracket^{A!E \Rightarrow B!E'} &= \lambda x^{\llbracket A \rrbracket}.\lambda h^{\llbracket E \rrbracket \mathcal{C}_{B,E'} \to \mathcal{C}_{B,E'}}.\llbracket N_{\mathrm{ret}} \rrbracket \\
\llbracket \{\ell\ p\ r \mapsto N_\ell\}_{\ell \in \mathcal{L}} \rrbracket^{A!E \Rightarrow B!E'} &= \lambda z^{\llbracket E \rrbracket \mathcal{C}_{B,E'}}.\mathbf{case}\ z\ \{(\ell\ \langle p, r \rangle \mapsto \llbracket N_\ell \rrbracket \gamma)_{\ell \in \mathcal{L}}\}
\end{aligned}$$

Figure 9 Typed first-order curried CPS translation of $\lambda^{\rho}_{\mathrm{eff}}$ without forwarding

With shapes we can give **vmap** the following typing rule

$$\textsc{SH-T-VMap}\quad \frac{\Delta \vdash R : \mathsf{Row}((\alpha_i^{K_i})_i.A, \emptyset) \qquad \Delta; \Gamma \vdash U : \forall (\alpha_i^{K_i})_i.A \to (B \to C) \to C \qquad \Delta; \Gamma \vdash V : [R] \qquad \Delta; \Gamma \vdash W : [((\alpha_i^{K_i})_i.A \Rightarrow B)R] \to C}{\Delta; \Gamma \vdash \mathbf{vmap}\ U\ V\ W : C}$$

where row kinds now take an additional shape parameter and the shape of a row type R of kind $\mathsf{Row}((\alpha_i^{K_i})_i.A, \mathcal{L})$ may be transformed using the special type operator $((\alpha_i^{K_i})_i.A \Rightarrow B)$ to a row of kind $\mathsf{Row}((\alpha_i^{K_i})_i.B, \mathcal{L})$. The CPS translation on the operation clauses now becomes

$$\llbracket \{\ell\ p\ r \mapsto N_\ell\}_{\ell \in \mathcal{L}} \rrbracket^{A!E \Rightarrow B!E'} = \lambda z^{\llbracket E \rrbracket \mathcal{C}_{B,E'}}.\mathbf{case}\ z\ \{(\ell\ \langle p, r \rangle \mapsto \llbracket N_\ell \rrbracket \gamma)_{\ell \in \mathcal{L}};\ y \mapsto M_{\mathrm{forward}}\}$$

where

$$\begin{aligned}
M_{\mathrm{forward}} &= \lambda k'^{\llbracket B \rrbracket \to C' \to \gamma}.\lambda h'^{C'}. \\
&\qquad \mathbf{vmap}\ (\Lambda\alpha^{\mathsf{Type}}\ \beta^{\mathsf{Type}}.\lambda\langle p, r \rangle^{\langle \alpha, \beta \to C' \rangle}\ k^{\langle \alpha, \beta \to C' \rangle \to C'}.k\ \langle p, \lambda x^{\beta}.r\ x\ k'\ h' \rangle)\ y\ h' \\
C' &= \llbracket E' \rrbracket \gamma \to \gamma
\end{aligned}$$

Performing the typed CPS translation (with forwarding) followed by type erasure yields the same result as performing the untyped translation of §4.2.1.

Our shapes are similar to the type schemes in Berthomieu and Sagazan's tagged types [3]. They can also be encoded using something similar to Remy's generalised record algebras [28]. We defer a full investigation of our extended row type system and typed variants of the uncurried CPS translations to future work.