Shallow Effect Handlers

Daniel Hillerström and Sam Lindley(✉)

The University of Edinburgh, United Kingdom
{Daniel.Hillerstrom,Sam.Lindley}@ed.ac.uk

Abstract. Plotkin and Pretnar's effect handlers offer a versatile abstraction for modular programming with user-defined effects. Traditional *deep handlers* are defined by folds over computation trees. In this paper we study *shallow handlers*, defined instead by case splits over computation trees. We show that deep and shallow handlers can simulate one another up to specific notions of administrative reduction. We present the first formal accounts of an abstract machine for shallow handlers and a Continuation Passing Style (CPS) translation for shallow handlers taking special care to avoid memory leaks. We provide implementations in the Links web programming language and empirically verify that neither implementation introduces unwarranted memory leaks.

Keywords: effect handlers · abstract machines · continuation passing

1 Introduction

Expressive control abstractions are pervasive in mainstream programming languages, be that async/await as pioneered by C#, generators and iterators as commonly found in JavaScript and Python, or coroutines in C++20. Such abstractions may be simulated directly with higher-order functions, but at the expense of writing the entire source program in Continuation Passing Style (CPS). To retain *direct-style*, some languages build in several different control abstractions, e.g., JavaScript has both async/await and generators/iterators, but hard-wiring multiple abstractions increases the complexity of the compiler and run-time.

An alternative is to provide a single control abstraction, and derive others as libraries. Plotkin and Pretnar's effect handlers provide a modular abstraction that subsumes all of the above control abstractions. Moreover, they have a strong mathematical foundation [19,20] and have found applications across a diverse spectrum of disciplines such as concurrent programming [4], probabilistic programming [8], meta programming [23], and more [11].

With effect handlers computations are viewed as trees. Effect handlers come in two flavours *deep* and *shallow*. Deep handlers are defined by folds (specifically *catamorphisms* [17]) over computation trees, whereas shallow handlers are defined as case-splits. Catamorphisms are attractive because they are semantically well-behaved and provide appropriate structure for efficient implementations using optimisations such as fusion [22]. However, they are not always convenient for implementing other structural recursion schemes such as mutual recursion.

Most existing accounts of effect handlers use deep handlers. In this paper we develop the theory of shallow effect handlers.

As shallow handlers impose no particular structural recursion scheme, they can be more convenient. For instance, using shallow handlers it is easy to model Unix pipes as two mutually recursive functions (specifically *mutumorphisms* [7]) that alternate production and consumption of data. With shallow handlers we define a classic demand-driven Unix pipeline operator as follows

$$\begin{aligned}
\mathsf{pipe} &: \langle \langle\rangle \to \alpha!\{\mathsf{Yield} : \beta \to \langle\rangle\}, \langle\rangle \to \alpha!\{\mathsf{Await} : \beta\}\rangle \to \alpha!\emptyset \\
\mathsf{copipe} &: \langle \beta \to \alpha!\{\mathsf{Await} : \beta\}, \langle\rangle \to \alpha!\{\mathsf{Yield} : \beta \to \langle\rangle\}\rangle \to \alpha!\emptyset
\end{aligned}$$

$$\begin{aligned}
\mathsf{pipe}\,\langle p, c\rangle = \mathbf{handle}^{\dagger}\ c\,\langle\rangle\ \mathbf{with}\ & \qquad \mathsf{copipe}\,\langle c, p\rangle = \mathbf{handle}^{\dagger}\ p\,\langle\rangle\ \mathbf{with} \\
\mathbf{return}\ x &\mapsto x & \mathbf{return}\ x &\mapsto x \\
\mathsf{Await}\ r &\mapsto \mathsf{copipe}\,\langle r, p\rangle & \mathsf{Yield}\ p\ r &\mapsto \mathsf{pipe}\,\langle r, \lambda\langle\rangle.c\,p\rangle
\end{aligned}$$

A pipe takes two thunked computations, a producer p and a consumer c. A computation type $A!E$ is a value type A and an effect E, which enumerates the operations that the computation may perform.

The pipe function specifies how to *handle* the operations of its arguments and in doing so performs no operations of its own, thus its effect is pure $\emptyset$. Each of the thunks returns a value of type α. The producer can perform the Yield operation, which yields a value of type β and the consumer can perform the Await operation, which correspondingly awaits a value of type β. The shallow handler runs the consumer. If the consumer returns a value, then the return clause is executed and simply returns that value as is. If the consumer performs the Await operation, then the handler is supplied with a special resumption argument r, which is the continuation of the consumer computation reified as a first-class function. The copipe is now invoked with r and the producer as arguments.

The copipe function is similar. The arguments are swapped and the consumer now expects a value. The shallow handler runs the producer. If it performs the Yield operation, then pipe is invoked with the resumption of the producer along with a thunk that applies the resumption of the consumer to the yielded value.

As a simple example consider the composition of a producer that yields a stream of ones, and a consumer that awaits a single value.

$$\begin{aligned}
& \mathsf{pipe}\,\langle \mathbf{rec}\ \mathit{ones}\ \langle\rangle.\mathbf{do}\ \mathsf{Yield}\ 1; \mathit{ones}\ \langle\rangle, \lambda\langle\rangle.\mathbf{do}\ \mathsf{Await}\rangle \\
\rightsquigarrow^{+}\ & \mathsf{copipe}\,\langle \lambda x.x, \mathbf{rec}\ \mathit{ones}\ \langle\rangle.\mathbf{do}\ \mathsf{Yield}\ 1; \mathit{ones}\ \langle\rangle\rangle \\
\rightsquigarrow^{+}\ & \mathsf{pipe}\,\langle \lambda\langle\rangle.\mathbf{rec}\ \mathit{ones}\ \langle\rangle.\mathbf{do}\ \mathsf{Yield}\ 1; \mathit{ones}\ \langle\rangle, \lambda\langle\rangle.1\rangle \rightsquigarrow^{+} 1
\end{aligned}$$

(The computation $\mathbf{do}\,\ell\,p$ performs operation ℓ with parameter p.)

The difference between shallow handlers and deep handlers is that in the latter the original handler is implicitly wrapped around the body of the resumption, meaning that the next effectful operation invocation is necessarily handled by the same handler. Shallow handlers allow the freedom to choose how to handle the next effectful operation; deep handlers do not. Pipes provide the quintessential example for contrasting shallow and deep handlers. To implement pipes with deep handlers, we cannot simply use term level recursion, instead we effectively have to *defunctionalise* [21] the shallow version of pipes using recursive types. Following Kammar et al. [11] we define two mutually recursive types

for producers and consumers, respectively.

$$\begin{aligned}\mathsf{Producer}\,\alpha\,\beta &= \langle\rangle \to (\mathsf{Consumer}\,\alpha\,\beta \to \alpha!\emptyset)!\emptyset\\ \mathsf{Consumer}\,\alpha\,\beta &= \beta \to (\mathsf{Producer}\,\alpha\,\beta \to \alpha!\emptyset)!\emptyset\end{aligned}$$

The underlying idea is *state-passing*: the Producer type is an alias for a suspended computation which returns a computation parameterised by a Consumer computation. Correspondingly, Consumer is an alias for a function that consumes an element of type β and returns a computation parameterised by a Producer computation. The ultimate return value has type α. Using these recursive types, we can now give types for deep pipe operators and their implementations.

$$\begin{aligned}\mathsf{pipe}' &: (\langle\rangle \to \alpha!\{\mathsf{Await} : \beta\}) \to \mathsf{Producer}\,\alpha\,\beta \to \mathsf{Consumer}\,\alpha\,\beta \to \alpha!\emptyset\\ \mathsf{copipe}' &: (\langle\rangle \to \alpha!\{\mathsf{Yield} : \beta \to \langle\rangle\}) \to \mathsf{Consumer}\,\alpha\,\beta \to \alpha!\emptyset\end{aligned}$$

$$\begin{aligned}\mathsf{pipe}'\ c = \mathbf{handle}\ c\,\langle\rangle\ \mathbf{with}\\ \quad \mathbf{return}\ x \mapsto \lambda y.x\\ \quad \mathsf{Await}\ r \mapsto \lambda p.p\,\langle\rangle\,r\end{aligned}\qquad\begin{aligned}\mathsf{copipe}'\ p = \mathbf{handle}\ p\,\langle\rangle\ \mathbf{with}\\ \quad \mathbf{return}\ x \mapsto \lambda y.x\\ \quad \mathsf{Yield}\ p\,r \mapsto \lambda c.c\,p\,r\end{aligned}$$

$$\mathsf{runPipe}\langle p; c\rangle = \mathsf{pipe}'\ c\,(\lambda\langle\rangle.\mathsf{copipe}'\ p)$$

Application of the pipe operator is no longer direct as extra plumbing is required to connect the now decoupled handlers. The observable behaviour of runPipe is the same as the shallow pipe. Indeed, the above example yields the same result.

$$\mathsf{runPipe}\,\langle \mathbf{rec}\ \mathit{ones}\,\langle\rangle.\mathbf{do}\ \mathsf{Yield}\,1; \mathit{ones}\,\langle\rangle, \lambda\langle\rangle.\mathbf{do}\ \mathsf{Await}\rangle \rightsquigarrow^{+} 1$$

In this paper we make five main contributions, each shedding their own light on the computational differences between deep and shallow handlers:

- A proof that shallow handlers with general recursion can simulate deep handlers up to congruence and that, at the cost of performance, deep handlers can simulate shallow handlers up to administrative reductions (§3).
- The first formal account of an abstract machine for shallow handlers (§4).
- The first formal account of a CPS translation for shallow handlers (§5).
- An implementation of both the abstract machine and the CPS translation as backends for the Links web programming language [2].
- Empirical evaluations of our implementations (§6).

§2 introduces our core calculus of deep and shallow effect handlers. §7 discusses related work and §8 concludes.

2 Handler Calculus

In this section, we present $\lambda^{\dagger}$, a Church-style row-polymorphic call-by-value calculus for effect handlers. To support comparison within a single language we include both deep and shallow handlers. The calculus is an extension of Hillerström and Lindley's calculus of extensible deep handlers $\lambda^{\rho}_{\text{eff}}$ [9] with shallow handlers and recursive functions. Following Hillerström and Lindley, $\lambda^{\dagger}$ provides a row polymorphic effect type system and is based on fine-grain call-by-value [15], which names each intermediate computation as in A-normal form [6], but unlike A-normal form is closed under β-reduction.

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

Fig. 1: Types, Kinds, and Environments

2.1 Types

The syntax of types, kinds, and environments is given in Fig. 1.

Value Types. Function type $A \to C$ maps values of type A to computations of type C. Polymorphic type $\forall \alpha^K.C$ is parameterised by a type variable α of kind K. Record type $\langle R \rangle$ represents records with fields constrained by row R. Dually, variant type $[R]$ represents tagged sums constrained by row R.

Computation Types and Effect 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.

Handler Types. The handler type $C \Rightarrow^{\delta} D$ represent handlers that transform computations of type C into computations of type D (where δ empty denotes a deep handler and $\delta = \dagger$ a shallow handler).

Row Types. Effect, record, and variant types are given by row types. A *row type* (or just *row*) describes a collection of distinct labels, each annotated by a presence type. A presence type indicates whether a label is *present* with 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 type can be instantiated with additional labels. We identify rows up to reordering of labels. For instance, we consider 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. The unit type is the empty closed record, that is, $\langle \cdot \rangle$. Dually, the empty type is the empty, closed variant $[\cdot]$. Often we omit the $\cdot$ for closed rows.

Kinds. We have six kinds: Type, Comp, Effect, $\mathsf{Handler}$, $\mathsf{Row}_{\mathcal{L}}$, $\mathsf{Presence}$, 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 $\mathit{Row}_{\emptyset}$. More generally, $\mathit{Row}_{\mathcal{L}}$ denotes a partial row that may not mention labels in $\mathcal{L}$. We write $\ell : A$ as sugar for $\ell : \mathsf{Pre}(A)$.

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

Fig. 2: Term Syntax

Type Variables. We let α, ρ and θ range over type variables. By convention we write 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.

2.2 Terms

The terms are given in Fig. 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 ($\Lambda\alpha^{K}.M$), the introduction forms for records and variants, and recursive functions ($\mathbf{rec}\, g^{A\to C}\, x.M$). 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.

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 to x in N.

Operation invocation ($\mathbf{do}\ \ell\ V)^{E}$ performs operation ℓ with value argument V. Handling ($\mathbf{handle}^{\delta}\, M\ \mathbf{with}\ H$) runs a computation M using deep (δ empty) or shallow ($\delta = \dagger$) handler 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. Various term forms are annotated with type or kind information; we sometimes omit such annotations. We write $Id(M)$ for $\mathbf{handle}\, M\, \mathbf{with}\, \{\mathbf{return}\, x \mapsto \mathbf{return}\, x\}$.

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

2.3 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 $\Delta; \Gamma \vdash H : C \Rightarrow^{\delta} D$ states that handler H has type $C \Rightarrow^{\delta} D$ 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 $FTV(\Gamma)$ 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 those for performing and handling operations.

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

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

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

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

The T-Handler and T-Handler† rules are 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. The difference in typing deep and shallow handlers is that the resumption of the former has return type D, whereas the resumption of the latter has return type C.

2.4 Operational Semantics

Figure 3 gives a small-step operational semantics for $\lambda^{\dagger}$. The reduction relation $\rightsquigarrow$ is defined on computation terms. The interesting rules are the handler rules.

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

$$\text{Evaluation contexts} \quad \mathcal{E} ::= [\,] \mid \mathbf{let}\ x \leftarrow \mathcal{E}\ \mathbf{in}\ N \mid \mathbf{handle}^{\delta}\ \mathcal{E}\ \mathbf{with}\ H$$

Fig. 3: Small-Step Operational Semantics

We write $BL(\mathcal{E})$ for the set of operation labels bound by $\mathcal{E}$.

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

The S-Ret rule invokes the return clause of a handler. The S-Op$^{\delta}$ rules handle an operation by invoking the appropriate operation clause. The constraint $\ell \notin BL(\mathcal{E})$ asserts that no handler in the evaluation context handles the operation: a handler reaches past any other inner handlers that do not handle ℓ. The difference between S-Op and S-Op† is that the former rewraps the handler about the body of the resumption. We write R^{+} for transitive closure of relation R.

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

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

3 Deep as Shallow and Shallow as Deep

In this section we show that shallow handlers and general recursion can simulate deep handlers up to congruence, and that deep handlers can simulate shallow handlers up to administrative reduction. Both translations are folklore, but we believe the precise simulation results are novel.

3.1 Deep as Shallow

The implementation of deep handlers using shallow handlers (and recursive functions) is by a rather direct local translation. Each handler is wrapped in a recursive function and each resumption has its body wrapped in a call to this recursive function. Formally, the translation $\mathcal{S}[\![-]\!]$ is defined as the homomorphic extension of the following equations to all terms.

$$\begin{aligned}
\mathcal{S}[\![\mathbf{handle}\ M\ \mathbf{with}\ H]\!] &= (\mathbf{rec}\ h\ f.\mathbf{handle}^{\dagger}\ f\ \langle\rangle\ \mathbf{with}\ \mathcal{S}[\![H]\!]h)\,(\lambda\langle\rangle.\mathcal{S}[\![M]\!]) \\
\mathcal{S}[\![H]\!]h &= \mathcal{S}[\![H^{\mathrm{ret}}]\!]h \uplus \mathcal{S}[\![H^{\mathrm{ops}}]\!]h \\
\mathcal{S}[\![\{\mathbf{return}\ x \mapsto N\}]\!]h &= \{\mathbf{return}\ x \mapsto \mathcal{S}[\![N]\!]\} \\
\mathcal{S}[\![\{\ell\ p\ r \mapsto N_\ell\}_{\ell\in\mathcal{L}}]\!]h &= \{\ell\ p\ r \mapsto \mathbf{let}\ r \leftarrow \mathbf{return}\ \lambda x.h\,(\lambda\langle\rangle.r\,x)\ \mathbf{in}\ \mathcal{S}[\![N_\ell]\!]\}_{\ell\in\mathcal{L}}
\end{aligned}$$

Theorem 3. *If* $\Delta;\Gamma \vdash M : C$ *then* $\Delta;\Gamma \vdash \mathcal{S}[\![M]\!] : C$.

In order to obtain a simulation result, we allow reduction in the simulated term to be performed under lambda abstractions (and indeed anywhere in a term), which is necessary because of the redefinition of the resumption to wrap the handler around its body. Nevertheless, the simulation proof makes minimal use of this power, merely using it to rename a single variable. We write R_{cong} for the compatible closure of relation R, that is the smallest relation including R and closed under term constructors for $\lambda^{\dagger}$.

Theorem 4 (Simulation up to Congruence). *If* $M \rightsquigarrow N$ *then* $\mathcal{S}[\![M]\!] \rightsquigarrow^{+}_{\mathrm{cong}} \mathcal{S}[\![N]\!]$.

Proof. By induction on $\rightsquigarrow$ using a substitution lemma. The interesting case is S-Deep-Op, which is where we apply a single β-reduction, renaming a variable, under the lambda abstraction representing the resumption.

3.2 Shallow as Deep

Implementing shallow handlers in terms of deep handlers is slightly more involved than the other way round. It amounts to the encoding of a case split by a fold and involves a translation on handler types as well as handler terms. Formally, the translation $\mathcal{D}[\![-]\!]$ is defined as the homomorphic extension of the following equations to all types, terms, and type environments.

$$\mathcal{D}[\![C \Rightarrow D]\!] = \mathcal{D}[\![C]\!] \Rightarrow \langle\langle\rangle \to \mathcal{D}[\![C]\!], \langle\rangle \to \mathcal{D}[\![D]\!]\rangle$$

$$\begin{aligned}
\mathcal{D}[\![\mathbf{handle}^{\dagger}\ M\ \mathbf{with}\ H]\!] &= \mathbf{let}\ z \leftarrow \mathbf{handle}\ \mathcal{D}[\![M]\!]\ \mathbf{with}\ \mathcal{D}[\![H]\!]\ \mathbf{in} \\
&\quad\ \mathbf{let}\ \langle f, g\rangle = z\ \mathbf{in}\ g\,\langle\rangle \\
\mathcal{D}[\![H]\!] &= \mathcal{D}[\![H^{\mathrm{ret}}]\!] \uplus \mathcal{D}[\![H^{\mathrm{ops}}]\!] \\
\mathcal{D}[\![\{\mathbf{return}\ x \mapsto N\}]\!] &= \{\mathbf{return}\ x \mapsto \mathbf{return}\ \langle\lambda\langle\rangle.\mathbf{return}\ x, \lambda\langle\rangle.\mathcal{D}[\![N]\!]\rangle\} \\
\mathcal{D}[\![\{\ell\ p\ r \mapsto N\}_{\ell\in\mathcal{L}}]\!] &= \{\ell\ p\ r \mapsto \\
&\qquad \mathbf{let}\ r = \lambda x.\mathbf{let}\ z \leftarrow r\,x\ \mathbf{in}\ \mathbf{let}\ \langle f, g\rangle = z\ \mathbf{in}\ f\,\langle\rangle\ \mathbf{in} \\
&\qquad \mathbf{return}\ \langle\lambda\langle\rangle.\mathbf{let}\ x \leftarrow \mathbf{do}\ \ell\ p\ \mathbf{in}\ r\,x, \lambda\langle\rangle.\mathcal{D}[\![N]\!]\rangle\}_{\ell\in\mathcal{L}}
\end{aligned}$$

Each shallow handler is encoded as a deep handler that returns a pair of thunks. The first forwards all operations, acting as the identity on computations. The second interprets a single operation before reverting to forwarding.

Theorem 5. *If* $\Delta; \Gamma \vdash M : C$ *then* $\mathcal{D}[\![\Delta]\!]; \mathcal{D}[\![\Gamma]\!] \vdash \mathcal{D}[\![M]\!] : \mathcal{D}[\![C]\!]$.

As with the implementation of deep handlers as shallow handlers, the implementation is again given by a local translation. However, this time the administrative overhead is more significant. Reduction up to congruence is insufficient and we require a more semantic notion of administrative reduction.

Definition 6 (Administrative Evaluation Contexts). *An evaluation context* $\mathcal{E}$ *is administrative,* $admin(\mathcal{E})$*, iff*

1. *For all values* V*, we have:* $\mathcal{E}[\mathbf{return}\ V] \rightsquigarrow^* \mathbf{return}\ V$
2. *For all evaluation contexts* $\mathcal{E}'$*, operations* $\ell \in BL(\mathcal{E}) \backslash BL(\mathcal{E}')$*, values* V*:*

$$\mathcal{E}[\mathcal{E}'[\mathbf{do}\ \ell\ V]] \rightsquigarrow^* \mathbf{let}\ x \leftarrow \mathbf{do}\ \ell\ V\ \mathbf{in}\ \mathcal{E}[\mathcal{E}'[\mathbf{return}\ x]]$$

The intuition is that an administrative evaluation context behaves like the empty evaluation context up to some amount of administrative reduction, which can only proceed once the term in the context becomes sufficiently evaluated. Values annihilate the evaluation context and handled operations are forwarded.

Definition 7 (Approximation up to Administrative Reduction). *Define* $\gtrsim$ *as the compatible closure of the following inference rules.*

$$\frac{}{M \gtrsim M} \qquad \frac{M \rightsquigarrow M' \qquad M' \gtrsim N}{M \gtrsim N} \qquad \frac{admin(\mathcal{E}) \qquad M \gtrsim N}{\mathcal{E}[M] \gtrsim N}$$

We say that M *approximates* N *up to administrative reduction if* $M \gtrsim N$*.*

Approximation up to administrative reduction captures the property that administrative reduction may occur anywhere within a term. The following lemma states that the forwarding component of the translation is administrative.

Lemma 8. *For all shallow handlers* H*, the following context is administrative:*

$$\mathbf{let}\ z \leftarrow \mathbf{handle}\ [\ ]\ \mathbf{with}\ \mathcal{D}[\![H]\!]\ \mathbf{in\ let}\ \langle f; \_ \rangle = z\ \mathbf{in}\ f\ \langle\rangle$$

Theorem 9 (Simulation up to Administrative Reduction). *If* $M' \gtrsim \mathcal{D}[\![M]\!]$ *and* $M \rightsquigarrow N$ *then there exists* N' *such that* $N' \gtrsim \mathcal{D}[\![N]\!]$ *and* $M' \rightsquigarrow^+ N'$*.*

Proof. By induction on $\rightsquigarrow$ using a substitution lemma and Lemma 8. The interesting case is S-Op†, which uses Lemma 8 to approximate the body of the resumption up to administrative reduction.

4 Abstract Machine

In this section we develop an abstract machine that supports deep and shallow handlers *simultaneously*. We build upon prior work [9] in which we developed an abstract machine for deep handlers by generalising the continuation structure of a CEK machine (Control, Environment, Kontinuation) [5]. In our prior work we sketched an adaptation for shallow handlers. It turns out that this adaptation has a subtle flaw. We fix the flaw here with a full development of shallow handlers along with a proof of correctness.

$$
\begin{array}{lrl}
\text{Configurations} & \mathcal{C} & ::= \langle M \mid \gamma \mid \kappa \circ \kappa' \rangle \\
\text{Value environments} & \gamma & ::= \emptyset \mid \gamma[x \mapsto v] \\
\text{Values} & v, w & ::= (\gamma, \lambda x^A.M) \mid (\gamma, \Lambda \alpha^K.M) \\
 & & \mid \ \langle\rangle \mid \langle \ell = v; w \rangle \mid (\ell\, v)^R \mid \kappa^A \mid (\kappa, \sigma)^A
\end{array}
$$

$$
\begin{array}{llll}
\text{Continuations} & \kappa ::= [] \mid \theta :: \kappa & \text{Continuation frames} & \theta ::= (\sigma, \chi) \\
 & & \text{Handler closures} & \chi ::= (\gamma, H)^\delta \\
\text{Pure continuations} & \sigma ::= [] \mid \phi :: \sigma & \text{Pure continuation frames} & \phi ::= (\gamma, x, N)
\end{array}
$$

Fig. 4: Abstract Machine Syntax

The informal account. A machine continuation is a list of handler frames. A handler frame is a pair of a *handler closure* (handler definition) and a *pure continuation* (a sequence of let bindings). Handling an operation amounts to searching through the continuation for a matching handler. The resumption is constructed during the search by reifying each handler frame. The resumption is assembled in one of two ways depending on whether the matching handler is deep or shallow. For a deep handler, the current handler closure is included, and a deep resumption is a reified continuation. An invocation of a deep resumption amounts to concatenating it with the current machine continuation. For a shallow handler, the current handler closure must be discarded leaving behind a dangling pure continuation, and a shallow resumption is a pair of this pure continuation and the remaining reified continuation. (By contrast, the prior flawed adaptation prematurely precomposed the pure continuation with the outer handler in the current resumption.) An invocation of a shallow resumption again amounts to concatenating it with the current machine continuation, but taking care to concatenate the dangling pure continuation with that of the next frame.

The formal account. The abstract machine syntax is given in Fig. 4. A configuration $\mathcal{C} = \langle M \mid \gamma \mid \kappa \circ \kappa' \rangle$ of our abstract machine is a quadruple of a computation term (M), an environment (γ) mapping free variables to values, and two continuations (κ) and (κ'). The latter continuation is always the identity, except when forwarding an operation, in which case it is used to keep track of the extent to which the operation has been forwarded. We write $\langle M \mid \gamma \mid \kappa \rangle$ as syntactic sugar for $\langle M \mid \gamma \mid \kappa \circ [] \rangle$ where $[]$ is the identity continuation.

Values consist of function closures, type function closures, records, variants, and captured continuations. A continuation κ is a stack of frames $[\theta_1, \ldots, \theta_n]$. We annotate captured continuations with input types in order to make the results of §4.1 easier to state. Each frame $\theta = (\sigma, \chi)$ represents pure continuation σ, corresponding to a sequence of let bindings, inside handler closure χ. A pure continuation is a stack of pure frames. A pure frame (γ, x, N) closes a let-binding **let** $x = [\,]$ **in** N over environment γ. A handler closure (γ, H) closes a handler definition H over environment γ. We write $[]$ for an empty stack, $x :: s$ for the result of pushing x on top of stack s, and $s +\!\!+ s'$ for the concatenation of stack s on top of s'. We use pattern matching to deconstruct stacks.

Transition function

$$
\begin{array}{lrll}
\textsc{M-Init} & M & \longrightarrow \langle M \mid \emptyset \mid [([], (\emptyset, \{\mathbf{return}\ x \mapsto x\}))] \rangle & \\
\textsc{M-AppClosure} & \langle V\ W \mid \gamma \mid \kappa \rangle & \longrightarrow \langle M \mid \gamma'[x \mapsto [\![W]\!]\gamma] \mid \kappa \rangle, & \text{if } [\![V]\!]\gamma = (\gamma', \lambda x^A.M) \\
\textsc{M-AppRec} & \langle V\ W \mid \gamma \mid \kappa \rangle & \longrightarrow \langle M \mid \gamma'[g \mapsto (\gamma', \mathbf{rec}\, g^{A \to C}\ x.M), x \mapsto [\![W]\!]\gamma] \mid \kappa \rangle, & \text{if } [\![V]\!]\gamma = (\gamma', \mathbf{rec}\, g^{A \to C}\ x.M) \\
\textsc{M-AppCont} & \langle V\ W \mid \gamma \mid \kappa \rangle & \longrightarrow \langle \mathbf{return}\ W \mid \gamma \mid \kappa' +\!\!+ \kappa \rangle, & \text{if } [\![V]\!]\gamma = (\kappa')^A \\
\textsc{M-AppCont}^\dagger & \langle V\ W \mid \gamma \mid (\sigma, \chi) :: \kappa \rangle & \longrightarrow \langle \mathbf{return}\ W \mid \gamma \mid \kappa' +\!\!+ ((\sigma' +\!\!+ \sigma, \chi) :: \kappa) \rangle, & \text{if } [\![V]\!]\gamma = (\kappa', \sigma')^A \\
\textsc{M-AppType} & \langle V\ A \mid \gamma \mid \kappa \rangle & \longrightarrow \langle M[A/\alpha] \mid \gamma' \mid \kappa \rangle, & \text{if } [\![V]\!]\gamma = (\gamma', \Lambda\alpha^K.\, M) \\
\textsc{M-Split} & \langle \mathbf{let}\ \langle \ell = x; y \rangle = V\ \mathbf{in}\ N \mid \gamma \mid \kappa \rangle & \longrightarrow \langle N \mid \gamma[x \mapsto v, y \mapsto w] \mid \kappa \rangle, & \text{if } [\![V]\!]\gamma = \langle \ell = v; w \rangle \\
\textsc{M-Case} & \langle \mathbf{case}\ V\ \{\ell\ x \mapsto M; y \mapsto N\} \mid \gamma \mid \kappa \rangle & \longrightarrow \begin{cases} \langle M \mid \gamma[x \mapsto v] \mid \kappa \rangle, \\ \langle N \mid \gamma[y \mapsto \ell'\ v] \mid \kappa \rangle, \end{cases} & \begin{array}{l} \text{if } [\![V]\!]\gamma = \ell\ v \\ \text{if } [\![V]\!]\gamma = \ell'\ v \text{ and } \ell \neq \ell' \end{array} \\
\textsc{M-Let} & \langle \mathbf{let}\ x \leftarrow M\ \mathbf{in}\ N \mid \gamma \mid (\sigma, \chi) :: \kappa \rangle & \longrightarrow \langle M \mid \gamma \mid ((\gamma, x, N) :: \sigma, \chi) :: \kappa \rangle & \\
\textsc{M-Handle} & \langle \mathbf{handle}^\delta\ M\ \mathbf{with}\ H \mid \gamma \mid \kappa \rangle & \longrightarrow \langle M \mid \gamma \mid ([], (\gamma, H)^\delta) :: \kappa \rangle & \\
\textsc{M-RetCont} & \langle \mathbf{return}\ V \mid \gamma \mid ((\gamma', x, N) :: \sigma, \chi) :: \kappa \rangle & \longrightarrow \langle N \mid \gamma'[x \mapsto [\![V]\!]\gamma] \mid (\sigma, \chi) :: \kappa \rangle & \\
\textsc{M-RetHandler} & \langle \mathbf{return}\ V \mid \gamma \mid ([], (\gamma', H)) :: \kappa \rangle & \longrightarrow \langle M \mid \gamma'[x \mapsto [\![V]\!]\gamma] \mid \kappa \rangle, & \text{if } H^{\mathrm{ret}} = \{\mathbf{return}\ x \mapsto M\} \\
\textsc{M-RetTop} & \langle \mathbf{return}\ V \mid \gamma \mid [] \rangle & \longrightarrow [\![V]\!]\gamma & \\
\textsc{M-Do} & \langle (\mathbf{do}\ \ell\ V)^E \mid \gamma \mid ((\sigma, (\gamma', H)) :: \kappa) \circ \kappa' \rangle & \longrightarrow \langle M \mid \gamma'[x \mapsto [\![V]\!]\gamma, r \mapsto (\kappa' +\!\!+ [(\sigma, (\gamma', H))])^B] \mid \kappa \rangle, & \\
 & & & \text{if } \ell : A \to B \in E \text{ and } H^\ell = \{\ell\ x\ r \mapsto M\} \\
\textsc{M-Do}^\dagger & \langle (\mathbf{do}\ \ell\ V)^E \mid \gamma \mid ((\sigma, (\gamma', H)^\dagger) :: \kappa) \circ \kappa' \rangle & \longrightarrow \langle M \mid \gamma'[x \mapsto [\![V]\!]\gamma, r \mapsto (\kappa', \sigma)^B] \mid \kappa \rangle, & \\
 & & & \text{if } \ell : A \to B \in E \text{ and } H^\ell = \{\ell\ x\ r \mapsto M\} \\
\textsc{M-Forward} & \langle (\mathbf{do}\ \ell\ V)^E \mid \gamma \mid ((\sigma, (\gamma', H)^\delta) :: \kappa) \circ \kappa' \rangle & \longrightarrow \langle (\mathbf{do}\ \ell\ V)^E \mid \gamma \mid \kappa \circ (\kappa' +\!\!+ [(\sigma, (\gamma', H)^\delta)]) \rangle, & \text{if } H^\ell = \emptyset
\end{array}
$$

Value interpretation

$$
\begin{array}{llll}
[\![x]\!]\gamma = \gamma(x) & [\![\lambda x^A.M]\!]\gamma = (\gamma, \lambda x^A.M) & [\![\Lambda\alpha^K.M]\!]\gamma = (\gamma, \Lambda\alpha^K.M) & [\![\mathbf{rec}\, g^{A \to C}\ x.M]\!]\gamma = (\gamma, \mathbf{rec}\, g^{A \to C}\ x.M) \\
[\![\langle\rangle]\!]\gamma = \langle\rangle & [\![\langle \ell = V; W \rangle]\!]\gamma = \langle \ell = [\![V]\!]\gamma; [\![W]\!]\gamma \rangle & [\![(\ell\ V)^R]\!]\gamma = (\ell\ [\![V]\!]\gamma)^R &
\end{array}
$$

Fig. 5: Abstract Machine Semantics

The abstract machine semantics defining the transition function $\longrightarrow$ is given in Fig. 5. It depends on an interpretation function $[\![-]\!]$ for values. The machine is initialised (M-Init) by placing a term in a configuration alongside the empty environment and identity continuation. The rules (M-AppClosure), (M-AppRec), (M-AppCont), (M-AppCont†), (M-AppType), (M-Split), and (M-Case) enact the elimination of values. The rules (M-Let) and (M-Handle) extend the current continuation with let bindings and handlers respectively. The rule (M-RetCont) binds a returned value if there is a pure continuation in the current continuation frame; (M-RetHandler) invokes the return clause of a handler if the pure continuation is empty; and (M-RetTop) returns a final value if the continuation is empty. The rule (M-Do) applies the current handler to an operation if the label matches one of the operation clauses. The captured continuation is assigned the forwarding continuation with the current frame appended to the end of it. The rule (M-Do†) is much like (M-Do), except it constructs a shallow resumption, discarding the current handler but keeping the current pure continuation. The rule (M-Forward) appends the current continuation frame onto the end of the forwarding continuation.

4.1 Correctness

The (M-Init) rule provides a canonical way to map a computation term onto a configuration. Fig. 6 defines an inverse mapping $(\!|-|\!)$ from configurations to computation terms via a collection of mutually recursive functions defined on configurations, continuations, computation terms, handler definitions, value terms, and values. We write $dom(\gamma)$ for the domain of γ and $\gamma\backslash\{x_1, \ldots, x_n\}$ for the restriction of environment γ to $dom(\gamma)\backslash\{x_1, \ldots, x_n\}$.

The $(\!|-|\!)$ function enables us to classify the abstract machine reduction rules according to how they relate to the operational semantics. The rules (M-Init) and (M-RetTop) are concerned only with initial input and final output, neither a feature of the operational semantics. The rules (M-AppCont$^\delta$), (M-Let), (M-Handle), and (M-Forward) are administrative in that $(\!|-|\!)$ is invariant under them. This leaves β-rules (M-AppClosure), (M-AppRec), (M-AppType), (M-Split), (M-Case), (M-RetCont), (M-RetHandler), (M-Do†), and (M-Do†), each of which corresponds directly to performing a reduction in the operational semantics. We write $\longrightarrow_a$ for administrative steps, $\longrightarrow_\beta$ for β-steps, and $\Longrightarrow$ for a sequence of steps of the form $\longrightarrow_a^* \longrightarrow_\beta$.

Each reduction in the operational semantics is simulated by a sequence of administrative steps followed by a single β-step in the abstract machine. The *Id* handler (§2.2) implements the top-level identity continuation.

Theorem 10 (Simulation). *If $M \rightsquigarrow N$, then for any $\mathcal{C}$ such that $(\!|\mathcal{C}|\!) = Id(M)$ there exists $\mathcal{C}'$ such that $\mathcal{C} \Longrightarrow \mathcal{C}'$ and $(\!|\mathcal{C}'|\!) = Id(N)$.*

Proof. By induction on the derivation of $M \rightsquigarrow N$.

Corollary 11. *If $\vdash M : A!E$ and $M \rightsquigarrow^+ N \not\rightsquigarrow$, then $M \longrightarrow^+ \mathcal{C}$ with $(\!|\mathcal{C}|\!) = N$.*

Configurations

$$(\!|\langle M \mid \gamma \mid \kappa \circ \kappa' \rangle|\!) = (\!|\kappa' +\!\!+ \kappa|\!)((\!|M|\!)\gamma) = (\!|\kappa'|\!)((\!|\kappa|\!)(\!|M|\!)\gamma)$$

Pure continuations

$$(\!|[]|\!)M = M \qquad (\!|((\gamma, x, N) :: \sigma)|\!)M = (\!|\sigma|\!)(\mathbf{let}\ x \leftarrow M\ \mathbf{in}\ (\!|N|\!)(\gamma \backslash \{x\}))$$

Continuations

$$(\!|[]|\!)M = M \qquad (\!|(\sigma, \chi) :: \kappa|\!)M = (\!|\kappa|\!)((\!|\chi|\!)((\!|\sigma|\!)(M)))$$

Handler closures

$$(\!|(\gamma, H)|\!)^{\delta} M = \mathbf{handle}^{\delta}\ M\ \mathbf{with}\ (\!|H|\!)\gamma$$

Computation terms

$$\begin{aligned}
(\!|V\ W|\!)\gamma &= (\!|V|\!)\gamma\ (\!|W|\!)\gamma \\
(\!|V\ A|\!)\gamma &= (\!|V|\!)\gamma\ A \\
(\!|\mathbf{let}\ \langle \ell = x; y \rangle = V\ \mathbf{in}\ N|\!)\gamma &= \mathbf{let}\ \langle \ell = x; y \rangle = (\!|V|\!)\gamma\ \mathbf{in}\ (\!|N|\!)(\gamma \backslash \{x, y\}) \\
(\!|\mathbf{case}\ V\ \{\ell\ x \mapsto M; y \mapsto N\}|\!)\gamma &= \mathbf{case}\ (\!|V|\!)\gamma\ \{\ell\ x \mapsto (\!|M|\!)(\gamma \backslash \{x\}); y \mapsto (\!|N|\!)(\gamma \backslash \{y\})\} \\
(\!|\mathbf{return}\ V|\!)\gamma &= \mathbf{return}\ (\!|V|\!)\gamma \\
(\!|\mathbf{let}\ x \leftarrow M\ \mathbf{in}\ N|\!)\gamma &= \mathbf{let}\ x \leftarrow (\!|M|\!)\gamma\ \mathbf{in}\ (\!|N|\!)(\gamma \backslash \{x\}) \\
(\!|\mathbf{do}\ \ell\ V|\!)\gamma &= \mathbf{do}\ \ell\ (\!|V|\!)\gamma \\
(\!|\mathbf{handle}^{\delta}\ M\ \mathbf{with}\ H|\!)\gamma &= \mathbf{handle}^{\delta}\ (\!|M|\!)\gamma\ \mathbf{with}\ (\!|H|\!)\gamma
\end{aligned}$$

Handler definitions

$$\begin{aligned}
(\!|\{\mathbf{return}\ x \mapsto M\}|\!)\gamma &= \{\mathbf{return}\ x \mapsto (\!|M|\!)(\gamma \backslash \{x\})\} \\
(\!|\{\ell\ x\ k \mapsto M\} \uplus H|\!)\gamma &= \{\ell\ x\ k \mapsto (\!|M|\!)(\gamma \backslash \{x, k\}\} \uplus (\!|H|\!)\gamma
\end{aligned}$$

Value terms and values

$$\begin{aligned}
(\!|x|\!)\gamma &= (\!|v|\!), \quad \text{if } \gamma(x) = v & (\!|\kappa^A|\!) &= \lambda x^A.(\!|\kappa|\!)(\mathbf{return}\ x) \\
(\!|x|\!)\gamma &= x, \quad \text{if } x \notin dom(\gamma) & (\!|(\kappa, \sigma)^A|\!) &= \lambda x^A.(\!|\sigma|\!)((\!|\kappa|\!)(\mathbf{return}\ x)) \\
(\!|\lambda x^A.M|\!)\gamma &= \lambda x^A.(\!|M|\!)(\gamma \backslash \{x\}) & (\!|(\gamma, \lambda x^A.M)|\!) &= \lambda x^A.(\!|M|\!)(\gamma \backslash \{x\}) \\
(\!|\Lambda \alpha^K.M|\!)\gamma &= \Lambda \alpha^K.(\!|M|\!)\gamma & (\!|(\gamma, \Lambda \alpha^K.M)|\!) &= \Lambda \alpha^K.(\!|M|\!)\gamma \\
(\!|\langle\rangle|\!)\gamma &= \langle\rangle & (\!|\langle\rangle|\!) &= \langle\rangle \\
(\!|\langle \ell = V; W \rangle|\!)\gamma &= \langle \ell = (\!|V|\!)\gamma; (\!|W|\!)\gamma \rangle & (\!|\langle \ell = v; w \rangle|\!) &= \langle \ell = (\!|v|\!); (\!|w|\!) \rangle \\
(\!|(\ell\ V)^R|\!)\gamma &= (\ell\ (\!|V|\!)\gamma)^R & (\!|(\ell\ v)^R|\!) &= (\ell\ (\!|v|\!))^R
\end{aligned}$$

$$(\!|\mathbf{rec}\ g^{A \to C}\ x.M|\!)\gamma = \mathbf{rec}\ g^{A \to C}\ x.(\!|M|\!)(\gamma \backslash \{g, x\}) = (\!|(\gamma, \mathbf{rec}\ g^{A \to C}\ x.M)|\!)$$

Fig. 6: Mapping from Abstract Machine Configurations to Terms

5 Higher-Order CPS Translation

In this section we formalise a CPS translation for deep and shallow handlers. We adapt the higher-order translation of Hillerström et al. [10]. They formalise a translation for deep handlers and then briefly outline an extension for shallow handlers. Alas, there is a bug in their extension. Their deep handler translation takes advantage of the rewrapping of the body of a resumption with the current handler to combine the current return clause with the current pure continuation. Their shallow handler translation attempts to do the same, but the combination

Syntax

$$\begin{array}{lll} & \text{Values} & V, W ::= x \mid \underline{\lambda} x\, k.M \mid \mathbf{rec}\, g\, x\, k.M \mid \ell \mid \underline{\langle} V, W \underline{\rangle} \mid \mathbf{res}^{\delta} \\ & \text{Computations} & M, N ::= V \mid U \mathbin{\underline{@}} V \mathbin{\underline{@}} W \mid \mathbf{let}\ \underline{\langle} x, y \underline{\rangle} = V\ \mathbf{in}\ N \\ & & \quad \mid \mathbf{case}\ V\, \{\ell \mapsto M; x \mapsto N\} \mid \mathbf{app}\ V\ W \end{array}$$

Syntactic sugar

$$\begin{array}{lll} \mathbf{let}\ x = V\ \mathbf{in}\ N \equiv N[V/x] & \langle\rangle \equiv \ell_{\langle\rangle} & [] \equiv \ell_{[]} \\ \ell\ V \equiv \underline{\langle} \ell, V \underline{\rangle} & \langle \ell = V; W \rangle \equiv \ell\ \underline{\langle} V, W \underline{\rangle} & V :: W \equiv \ell_{::}\ \underline{\langle} V, W \underline{\rangle} \end{array}$$

$$\begin{array}{ll} \mathbf{case}\ V\ \{\ell\, x \mapsto M; y \mapsto N\} \equiv & \mathbf{let}\ \langle \ell = x; y \rangle = V\ \mathbf{in}\ N \equiv \\ \quad \mathbf{let}\ y = V\ \mathbf{in}\ \mathbf{let}\ \underline{\langle} z, x \underline{\rangle} = y\ \mathbf{in} & \quad \mathbf{let}\ \underline{\langle} z, z' \underline{\rangle} = V\ \mathbf{in}\ \mathbf{let}\ \underline{\langle} x, y \underline{\rangle} = z'\ \mathbf{in} \\ \quad \mathbf{case}\ z\ \{\ell \mapsto M; z \mapsto N\} & \quad \mathbf{case}\ z\ \{\ell \mapsto N; z \mapsto \ell_{\perp}\} \end{array}$$

Reductions

$$\begin{array}{lll} \text{U-App} & (\underline{\lambda} x\, k.M) \mathbin{\underline{@}} V \mathbin{\underline{@}} W \rightsquigarrow M[V/x, W/k] & \\ \text{U-Rec} & (\mathbf{rec}\, g\, x\, k.M) \mathbin{\underline{@}} V \mathbin{\underline{@}} W \rightsquigarrow M[\mathbf{rec}\, g\, x\, k.M/g, V/x, W/k] & \\ \text{U-Split} & \mathbf{let}\ \underline{\langle} x, y \underline{\rangle} = \underline{\langle} V, W \underline{\rangle}\ \mathbf{in}\ N \rightsquigarrow N[V/x, W/y] & \\ \text{U-Case}_1 & \mathbf{case}\ \ell\, \{\ell \mapsto M; x \mapsto N\} \rightsquigarrow M & \\ \text{U-Case}_2 & \mathbf{case}\ \ell\, \{\ell' \mapsto M; x \mapsto N\} \rightsquigarrow N[\ell/x], & \text{if } \ell \neq \ell' \\ \text{U-KAppNil} & \mathbf{app}\ (\underline{\langle}[], \underline{\langle} v, e \underline{\rangle} \mathbin{\underline{::}} k)\ V \rightsquigarrow v \mathbin{\underline{@}} V \mathbin{\underline{@}} k & \\ \text{U-KAppCons} & \mathbf{app}\ (\underline{\langle} f \mathbin{\underline{::}} s, h \underline{\rangle} \mathbin{\underline{::}} k)\ V \rightsquigarrow f \mathbin{\underline{@}} V \mathbin{\underline{@}} (\underline{\langle} s, h \underline{\rangle} \mathbin{\underline{::}} k) & \\ \text{U-Res} & \mathbf{res}\, (q_n \mathbin{\underline{::}} \cdots \mathbin{\underline{::}} q_1 \mathbin{\underline{::}} []) \rightsquigarrow \lambda x\, k.\mathbf{app}\ (q_1 \mathbin{\underline{::}} \cdots \mathbin{\underline{::}} q_n \mathbin{\underline{::}} k)\ x & \\ \text{U-Res}^{\dagger} & \mathbf{res}^{\dagger}\, (\underline{\langle} f_1 \mathbin{\underline{::}} \cdots \mathbin{\underline{::}} f_m, h \underline{\rangle} \mathbin{\underline{::}} q_n \mathbin{\underline{::}} \cdots \mathbin{\underline{::}} q_1 \mathbin{\underline{::}} []) \rightsquigarrow & \\ & \quad \lambda x\, k.\mathbf{let}\ (\underline{\langle} s', h' \underline{\rangle} \mathbin{\underline{::}} k') = k\ \mathbf{in} & \\ & \quad\quad \mathbf{app}\ (q_1 \mathbin{\underline{::}} \cdots \mathbin{\underline{::}} q_n \mathbin{\underline{::}} \underline{\langle} f_1 \mathbin{\underline{::}} \cdots \mathbin{\underline{::}} f_m \mathbin{\underline{::}} s', h' \underline{\rangle} \mathbin{\underline{::}} k')\ x & \end{array}$$

Fig. 7: Untyped Target Calculus

is now unsound as the return clause must be discarded by the resumption. We fix the bug by explicitly separating out the return continuation. Moreover, our translation is carefully designed to avoid memory leaks. The key insight is that to support the typical tail-recursive pattern of shallow handlers without generating useless identity continuations it is essential that we detect and eliminate them. We do so by representing pure continuations as lists of pure frames whereby the identity continuation is just an empty list, much like the abstract machine of §4.

Following Hillerström et al. [10], we present a higher-order uncurried CPS translation into an untyped lambda calculus. In the style of Danvy and Nielsen [3], 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: overline denotes a static syntax constructor; underline denotes a dynamic syntax constructor. To facilitate this notation we write application as an infix "at" symbol (@). We assume the meta language is pure and hence respects the usual β and η equivalences.

5.1 Target Calculus

The target calculus is given in Fig. 7. As in $\lambda^\dagger$ there is a syntactic distinction between values (V) and computations (M). Values (V) comprise: lambda abstractions ($\underline{\lambda} x\ k.M$) and recursive functions ($\mathbf{rec}\ g\ x\ k.M$), each of which take an additional continuation parameter; first-class labels (ℓ); pairs $\underline{\langle} V, W \underline{\rangle}$; and two special convenience constructors for building deep ($\mathbf{res}\ V$) and shallow ($\mathbf{res}^\dagger\ V$) resumptions, which we will explain shortly. Computations (M) comprise: values (V); applications ($U\ \underline{@}\ V\ \underline{@}\ W$); pair elimination ($\mathbf{let}\ \underline{\langle} x, y \underline{\rangle} = V\ In\ N$); label elimination ($\mathbf{case}\ V\ \{\ell \mapsto M; x \mapsto N\}$); and a special convenience constructor for continuation application ($\mathbf{app}\ V\ W$).

Lambda abstraction, pairs, application, and pair elimination are underlined to distinguish them from equivalent constructs in the meta language. We define syntactic sugar for variant values, record values, list values, let binding, variant eliminators, and record eliminators. We assume standard n-ary generalisations and use pattern matching syntax for deconstructing variants, records, and lists.

The reductions for functions, pairs, and first-class labels are standard. To explain the reduction rules for continuations, we first explain the encoding of continuations. Much like the abstract machine, a continuation (k) is given by a list of continuation frames. A continuation frame ($\underline{\langle} s, h \underline{\rangle}$) consists of a pair of a pure continuation (s) and a handler (h). A pure continuation is a list of pure continuation frames (f). A handler is a pair of a return continuation (v) and an effect continuation (e) which dispatches on the operations provided by a handler. There are two continuation reduction rules, both of which inspect the first frame of the continuation. If the pure continuation of this frame is empty then the return clause is invoked (U-KAppNil). If the pure continuation of this frame is non-empty then the first pure continuation frame is invoked (U-KAppCons). A crucial difference between our representation of continuations and that of Hillerström et al. [10] is that they use a flat list of frames whereas we use a nested structure in which each pure continuation is a list of pure frames.

To explain the reduction rules for continuations, we first explain the encoding of resumptions. Reified resumptions are constructed frame-by-frame as reversed continuations — they grow a frame at a time as operations are forwarded through the handler stack. Hillerström et al. [10] adopt such an intensional representation in order to obtain a relatively tight simulation result. We take further advantage of this representation to discard the handler when constructing a shallow handler's resumption. The resumption reduction rules turn reified resumptions into actual resumptions. The deep rule (U-Res) simply appends the reified resumption onto the continuation. The shallow rule (U-Res†) appends the tail of the reified resumption onto the continuation after discarding the topmost handler from the resumption and appending the topmost pure continuation from the resumption onto the topmost pure continuation of the continuation.

The continuation application and resumption constructs along with their reduction rules are macro-expressible in terms of the standard constructs. We choose to build them in order to keep the presentation relatively concise.

Values

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

$$[\![\mathbf{rec}\ g\ x.M]\!] = \mathbf{rec}\ g\ x\ k.[\![M]\!]\ \overline{@}\ k$$

Computations

$$\begin{array}{rl}
[\![V\ W]\!] &= \overline{\lambda}\kappa.[\![V]\!]\ \underline{@}\ [\![W]\!]\ \underline{@}\downarrow\kappa \\
[\![V\ T]\!] &= \overline{\lambda}\kappa.[\![V]\!]\ \underline{@}\ \langle\rangle\ \underline{@}\downarrow\kappa \\
[\![\mathbf{let}\ \langle \ell = x; y\rangle = V\ \mathbf{in}\ N]\!] &= \overline{\lambda}\kappa.\mathbf{let}\ \langle \ell = x; y\rangle = [\![V]\!]\ \mathbf{in}\ [\![N]\!]\ \overline{@}\ \kappa \\
[\![\mathbf{case}\ V\ \{\ell\ x \mapsto M; y \mapsto N\}]\!] &= \overline{\lambda}\kappa.\mathbf{case}\ [\![V]\!]\ \{\ell\ x \mapsto [\![M]\!]\ \overline{@}\ \kappa; y \mapsto [\![N]\!]\ \overline{@}\ \kappa\} \\
[\![\mathbf{absurd}\ V]\!] &= \overline{\lambda}\kappa.\mathbf{absurd}\ [\![V]\!] \\
[\![\mathbf{return}\ V]\!] &= \overline{\lambda}\kappa.\mathbf{app}\downarrow\kappa\ [\![V]\!] \\
[\![\mathbf{let}\ x \leftarrow M\ \mathbf{in}\ N]\!] &= \overline{\lambda}\overline{\langle} s, \chi\overline{\rangle} :: \kappa.[\![M]\!]\ \overline{@}\ (\overline{\langle}(\underline{\lambda} x\ k.[\![N]\!]\ \overline{@}\ k) :: s, \chi\overline{\rangle} :: \kappa) \\
[\![\mathbf{do}\ \ell\ V]\!] &= \overline{\lambda}\overline{\langle} s, \overline{\langle} v, e\overline{\rangle}\rangle :: \kappa.e\ \underline{@}\ (\ell\ \underline{\langle}[\![V]\!], \underline{\langle} s, \underline{\langle} v, e\underline{\rangle}\rangle :: []\underline{\rangle})\ \underline{@}\downarrow\kappa \\
[\![\mathbf{handle}^{\delta}\ M\ \mathbf{with}\ H]\!] &= \overline{\lambda}\kappa.[\![M]\!]\ \overline{@}\ \overline{\langle}[], [\![H]\!]^{\delta}\overline{\rangle} :: \kappa \\
[\![H]\!]^{\delta} &= \overline{\langle}[\![H^{\mathrm{ret}}]\!], [\![H^{\mathrm{ops}}]\!]^{\delta}\overline{\rangle} \\
[\![\{\mathbf{return}\ x \mapsto N\}]\!] &= \underline{\lambda} x\ k.[\![N]\!]\ \overline{@}\ k \\
[\![\{(\ell\ p\ r \mapsto N_\ell)_{\ell\in\mathcal{L}}\}]\!]^{\delta} &= \underline{\lambda} x\ k.\mathbf{let}\ \underline{\langle} z, \underline{\langle} p, rk\underline{\rangle}\rangle = x\ \mathbf{in} \\
&\qquad \mathbf{case}\ z\ \{(\ell \mapsto \mathbf{let}\ r = \mathbf{res}^{\delta}\ rk\ \mathbf{in}\ [\![N_\ell]\!]\ \overline{@}\ k)_{\ell\in\mathcal{L}} \\
&\qquad\quad y \mapsto M_{\mathrm{forward}}((y, p, rk), k)\} \\
M_{\mathrm{forward}}((y, p, rk), k) &= \mathbf{let}\ \underline{\langle} s', \underline{\langle} v', e'\underline{\rangle}\rangle :: k' = k\ \mathbf{in} \\
&\quad e'\ \underline{@}\ (y\ \underline{\langle} p, \underline{\langle} s', \overline{\langle} v', e'\underline{\rangle}\rangle :: rk\underline{\rangle})\ \underline{@}\ k'
\end{array}$$

Top-level program

$$\top[\![M]\!] = [\![M]\!]\ \overline{@}\ (\overline{\langle}[], \overline{\langle}\underline{\lambda} x\ k.x, \underline{\lambda} z\ k.\mathbf{absurd}\ z\overline{\rangle}\rangle :: [])$$

Fig. 8: Higher-Order Uncurried CPS Translation of $\lambda^{\dagger}$

5.2 Static Terms

Redexes marked as $\overline{\text{static}}$ are reduced as part of the translation (at compile time), whereas those marked as $\underline{\text{dynamic}}$ are reduced at runtime.

We make use of static lambda abstractions, pairs, and lists. We let κ range over static continuations and χ range over static handlers. We let $\mathcal{V}, \mathcal{W}$ range over meta language values, $\mathcal{M}$ range over meta language expressions, and $\mathcal{P}, \mathcal{Q}$ over meta language patterns. We use list and record pattern matching in the meta language.

$$\begin{array}{l}
(\overline{\lambda}\overline{\langle}\mathcal{P}, \mathcal{Q}\overline{\rangle}.\mathcal{M})\ \overline{@}\ \overline{\langle}\mathcal{V}, \mathcal{W}\overline{\rangle} \;=\; (\overline{\lambda}\mathcal{P}.\overline{\lambda}\mathcal{Q}.\mathcal{M})\ \overline{@}\ \mathcal{V}\ \overline{@}\ \mathcal{W} \;=\; (\overline{\lambda}(\mathcal{P} :: \mathcal{Q}).\mathcal{M})\ \overline{@}\ (\mathcal{V} :: \mathcal{W}) \\
(\overline{\lambda}\overline{\langle}\mathcal{P}, \mathcal{Q}\overline{\rangle}.\mathcal{M})\ \overline{@}\ V \;=\; \mathbf{let}\ \underline{\langle} f, s\underline{\rangle} = V\ \mathbf{in}\ (\overline{\lambda}\mathcal{P}.\overline{\lambda}\mathcal{Q}.\mathcal{M})\ \overline{@}\ f\ \overline{@}\ s \;=\; (\overline{\lambda}(\mathcal{P} :: \mathcal{Q}).\mathcal{M})\ \overline{@}\ V
\end{array}$$

A meta language value V can be reified as a target language value $\downarrow V$.

$$\downarrow V = V \qquad \downarrow(\mathcal{V} :: \mathcal{W}) = \downarrow\mathcal{V} :: \downarrow\mathcal{W} \qquad \downarrow\overline{\langle}\mathcal{V}, \mathcal{W}\overline{\rangle} = \underline{\langle}\downarrow\mathcal{V}, \downarrow\mathcal{W}\underline{\rangle}$$

5.3 The Translation

The CPS translation is given in Fig. 8. Its behaviour on constructs for introducing and eliminating values is standard. Where necessary static continuations in

the meta language are reified as dynamic continuations in the target language. The translation of **return** V applies the continuation to $[\![V]\!]$. The translation of **let** $x \leftarrow M$ **in** N adds a frame to the pure continuation on the topmost frame of the continuation. The translation of **do** $\ell\ V$ dispatches the operation to the effect continuation at the head of the continuation. The resumption is initialised with the topmost frame of the continuation. The translations of deep and shallow handling each add a new frame to the continuation. The translation of the operation clauses of a handler dispatches on the operation. If a match is found then the reified resumption is turned into a function and made available in the body of the operation clause. If there is no match, then the operation is forwarded by unwinding the continuation, transferring the topmost frame to the head of the reified resumption before invoking the next effect continuation. The only difference between the translations of a deep handler and a shallow handler is that the reified resumption of the latter is specially marked in order to ensure that the handler is disposed of in the body of a matching operation clause.

Example. The following example illustrates how the higher-order CPS translation avoids generating administrative redexes by performing static reductions.

$$\begin{aligned}
\top[\![\textbf{handle}\ (\textbf{do}\ \mathsf{Await}\ \underline{\langle\rangle})\ \textbf{with}\ H]\!] &= [\![\textbf{handle}\ (\textbf{do}\ \mathsf{Await}\ \underline{\langle\rangle})\ \textbf{with}\ H]\!]\ \overline{@}\ \mathcal{K}_\top \\
&= [\![\textbf{do}\ \mathsf{Await}\ \underline{\langle\rangle}]\!]\ \overline{@}\ \overline{\langle}[], [\![H]\!]\overline{\rangle} :\!\overline{:}\ \mathcal{K}_\top \\
&= [\![\textbf{do}\ \mathsf{Await}\ \underline{\langle\rangle}]\!]\ \overline{@}\ \overline{\langle}[], \overline{\langle}[\![H^{\mathrm{ret}}]\!], [\![H^{\mathrm{ops}}]\!]\overline{\rangle}\overline{\rangle} :\!\overline{:}\ \mathcal{K}_\top \\
&= [\![H^{\mathrm{ops}}]\!]\ \underline{@}\ \mathsf{Await}\ \underline{\langle\langle\rangle}, \underline{\langle}[], [\![H]\!]\underline{\rangle} :\!\underline{:} []\underline{\rangle}\ \underline{@}\downarrow\mathcal{K}_\top
\end{aligned}$$

where $\mathcal{K}_\top = (\overline{\langle}[], \overline{\langle}\underline{\lambda} x\ k.x, \underline{\lambda} z\ k.\textbf{absurd}\ z\overline{\rangle}\overline{\rangle} :\!\overline{:}\ [])$. The resulting term passes Await directly to the dispatcher that implements the operation clauses of H.

5.4 Correctness

The translation naturally lifts to evaluation contexts.

$$\begin{aligned}
[\![\ ]\!] &= \overline{\lambda}\kappa.\kappa \\
[\![\textbf{let}\ x \leftarrow \mathcal{E}\ \textbf{in}\ N]\!] &= \overline{\lambda}\overline{\langle} s, \chi\overline{\rangle} :\!\overline{:}\ \kappa.[\![\mathcal{E}]\!]\ \overline{@}\ (\overline{\langle}(\underline{\lambda} x\ k.[\![N]\!]\ \overline{@}\ k) :\!\underline{:}\ s, \chi\overline{\rangle} :\!\overline{:}\ \kappa) \\
[\![\textbf{handle}^{\delta}\ \mathcal{E}\ \textbf{with}\ H]\!] &= \overline{\lambda}\kappa.[\![\mathcal{E}]\!]\ \overline{@}\ (\overline{\langle}[], [\![H]\!]^{\delta}\overline{\rangle} :\!\overline{:}\ \kappa)
\end{aligned}$$

Lemma 12 (Decomposition). $[\![\mathcal{E}[M]]\!]\ \overline{@}\ (\mathcal{V} :\!\overline{:}\ \mathcal{W}) = [\![M]\!]\ \overline{@}\ ([\![\mathcal{E}]\!]\ \overline{@}\ (\mathcal{V} :\!\overline{:}\ \mathcal{W}))$

Though it eliminates static administrative redexes, the translation still yields administrative redexes that cannot be eliminated statically, as they only appear at run-time, which arise from deconstructing a reified stack of continuations. We write $\rightsquigarrow_{\mathrm{a}}$ for the compatible closure of U-Split, U-Case$_1$ and U-Case$_2$.

The following lemma is central to our simulation theorem. It characterises the sense in which the translation respects the handling of operations.

Lemma 13 (Handling). *If* $\ell \notin BL(\mathcal{E})$ *and* $H^{\ell} = \{\ell\ p\ r \mapsto N_{\ell}\}$ *then:*

1. $[\![\textbf{do}\ \ell\ V]\!]\ \overline{@}\ ([\![\mathcal{E}]\!]\ \overline{@}\ (\overline{\langle}[], [\![H]\!]\overline{\rangle} :\!\overline{:}\ \mathcal{W})) \rightsquigarrow^{+} \rightsquigarrow_{a}^{*}$
   $([\![N_{\ell}]\!]\ \overline{@}\ \mathcal{W})[[\![V]\!]/p, \underline{\lambda} y\ k.[\![\textbf{return}\ y]\!]\ \overline{@}\ ([\![\mathcal{E}]\!]\ \overline{@}\ (\overline{\langle}[], [\![H]\!]\overline{\rangle} :\!\overline{:}\ k))/r]$

2. $[\![\mathbf{do}\ \ell\ V]\!]\ \overline{@}\ ([\![\mathcal{E}]\!]\ \overline{@}\ (\overline{\langle}[], [\![H]\!]^{\dagger}\overline{\rangle} :: \mathcal{W})) \rightsquigarrow^{+} \rightsquigarrow_a^*$
$([\![N_\ell]\!]\ \overline{@}\ \mathcal{W})[[\![V]\!]/p, \underline{\lambda} y\, k.\mathbf{let}\ (\underline{\langle} s, \underline{\langle} v, e\underline{\rangle}\rangle :: k) = k\ \mathbf{in}$
$[\![\mathbf{return}\ y]\!]\ \overline{@}\ ([\![\mathcal{E}]\!]\ \overline{@}\ (\overline{\langle} s, \overline{\langle} v, e\overline{\rangle}\rangle :: k))/r]$

We now give a simulation result in the style of Plotkin [18]. The theorem shows that the only extra behaviour exhibited by a translated term is the necessary bureaucracy of dynamically deconstructing the continuation stack.

Theorem 14 (Simulation). *If $M \rightsquigarrow N$ then for all static values $\mathcal{V}$ and $\mathcal{W}$, we have $[\![M]\!]\ \overline{@}\ (\mathcal{V} :: \mathcal{W}) \rightsquigarrow^{+} \rightsquigarrow_a^* [\![N]\!]\ \overline{@}\ (\mathcal{V} :: \mathcal{W})$.*

Proof. By induction on the reduction relation ($\rightsquigarrow$) using Lemma 13.

As a corollary, we obtain that the translation simulates full reduction to a value.

Corollary 15. *$M \rightsquigarrow^* V$ iff $\top[\![M]\!] \rightsquigarrow^* \rightsquigarrow_a^* \top[\![V]\!]$.*

6 Empirical Evaluation

We conducted a basic empirical evaluation using an experimental branch of the Links web programming language [2] extended with support for shallow handlers and JavaScript backends based on the CEK machine (§4) and CPS translation (§5). The full details are given in Appendix B. Here we give a brief high-level summary. Our benchmarks are adapted from Kammar et al. [11], comprising: pipes, a count down loop, and n-Queens. Broadly, our results align with those of Kammar et al. Specifically, the shallow implementation of pipes outperforms the deep implementation. The shallow-as-deep translation fails to complete most benchmarks as it runs out of memory. The memory usage pattern exhibited by deep, shallow, and shallow-as-deep implementations are all stable.

Deep handlers perform slightly better than shallow handlers except on the pipes benchmark (CEK and CPS) and the countdown benchmark on the CEK machine. The former is hardly surprising given the inherent indirection in the deep implementation of pipes, which causes unnecessary closure allocations to happen when sending values from one end of the pipe to the other. We conjecture that the relatively poor performance of deep handlers on the CEK version of the countdown benchmark is also due to unnecessary closure allocation in the interpretation of state. Kammar et al. avoid this problem by adopting *parameterised handlers*, which thread a parameter through each handler.

7 Related Work

Shallow Handlers. Most existing accounts of effect handlers use deep handlers. Notable exceptions include Haskell libraries based on free monads [11,12,13], and the Frank programming language [16]. Kiselyov and Ishii [12] optimise their implementation by allowing efficient implementations of catenable lists to be used to support manipulation of continuations. We conjecture that both our abstract machine and our CPS translation could benefit from a similar representation.

Abstract Machines for Handlers. Lindley et al. [16] implement Frank using an abstract machine similar to the one described in this paper. Their abstract machine is not formalised and differs in several ways. In particular, continuations are represented by a single flattened stack, rather than a nested stack like ours, and Frank supports multihandlers, which handle several computations at once. Biernacki et al. [1] present an abstract machine for deep effect handlers similar to that of Hillerström and Lindley [9] but factored slightly differently.

CPS for Handlers. Leijen [14] implements a selective CPS translation for deep handlers, but does not go all the way to plain lambda calculus, relying on a special built in handling construct.

8 Conclusion and Future Work

We have presented the first comprehensive formal analysis of shallow effect handlers. We introduced the handler calculus $\lambda^\dagger$ as a uniform calculus of deep and shallow handlers. We specified formal translations back and forth between deep and shallow handlers within $\lambda^\dagger$, an abstract machine for $\lambda^\dagger$, and a higher-order CPS translation for $\lambda^\dagger$. In each case we proved a precise simulation result, drawing variously on different notions of administrative reduction. We have implemented the abstract machine and CPS translation as backends for Links and evaluated the performance of deep and shallow handlers and their encodings, measuring both execution time and memory consumption. Though deep and shallow handlers can always encode one another, the results suggest that the shallow-as-deep encoding is not viable in practice due to administrative overhead, whereas the deep-as-shallow encoding may be viable. In future we intend to perform a more comprehensive performance evaluation for a wider range of effect handler implementations.

Another outstanding question is to what extent shallow handlers are really needed at all. We have shown that we can encode them generically using deep handlers, but the resulting cruft hinders performance in practice. Extensions to deep handlers not explored in this paper, such as *parameterised handlers* [11,20] or a deep version of the *multihandlers* of Lindley et al. [16], offer the potential for expressing certain shallow handlers without the cruft. Parameterised handlers thread a parameter through each handler, avoiding unnecessary closure allocation. Deep multihandlers directly capture mutumorphisms over computations, allowing a direct implementation of pipes. In future we plan to study the precise relationship between shallow handlers, parameterised handlers, deep multihandlers, and perhaps handlers based on other structural recursion schemes.

Acknowledgements. We would like to thank John Longley for insightful discussions about the inter-encodings of deep and shallow handlers. Daniel Hillerström was supported by EPSRC grant EP/L01503X/1 (EPSRC Centre for Doctoral Training in Pervasive Parallelism). Sam Lindley was supported by EPSRC grant EP/K034413/1 (From Data Types to Session Types—A Basis for Concurrency and Distribution).

References

1. Biernacki, D., Piróg, M., Polesiuk, P., Sieczkowski, F.: Handle with care: relational interpretation of algebraic effects and handlers. PACMPL **2**(POPL), 8:1–8:30 (2018)
2. Cooper, E., Lindley, S., Wadler, P., Yallop, J.: Links: Web programming without tiers. In: FMCO. LNCS, vol. 4709, pp. 266–296. Springer (2006)
3. Danvy, O., Nielsen, L.R.: A first-order one-pass CPS transformation. Theor. Comput. Sci. **308**(1-3), 239–257 (2003)
4. Dolan, S., White, L., Sivaramakrishnan, K., Yallop, J., Madhavapeddy, A.: Effective concurrency through algebraic effects. OCaml Workshop (2015)
5. Felleisen, M., Friedman, D.P.: Control operators, the SECD-machine, and the λ-calculus. In: Formal Description of Programming Concepts III. pp. 193–217 (1987)
6. Flanagan, C., Sabry, A., Duba, B.F., Felleisen, M.: The essence of compiling with continuations. In: PLDI. pp. 237–247. ACM (1993)
7. Fokkinga, M.M.: Tupling and mutumorphisms. The Squiggolist **1**(4), 81–82 (1990)
8. Goodman, N.: Uber AI Labs open sources Pyro, a deep probabilistic programming language (Nov 2017), https://eng.uber.com/pyro/
9. Hillerström, D., Lindley, S.: Liberating effects with rows and handlers. In: TyDe@ICFP. pp. 15–27. ACM (2016)
10. Hillerström, D., Lindley, S., Atkey, R., Sivaramakrishnan, K.C.: Continuation passing style for effect handlers. In: FSCD. LIPIcs, vol. 84, pp. 18:1–18:19 (2017)
11. Kammar, O., Lindley, S., Oury, N.: Handlers in action. In: ICFP. pp. 145–158. ACM (2013)
12. Kiselyov, O., Ishii, H.: Freer monads, more extensible effects. In: Haskell. pp. 94–105. ACM (2015)
13. Kiselyov, O., Sabry, A., Swords, C.: Extensible effects: an alternative to monad transformers. In: Haskell. pp. 59–70. ACM (2013)
14. Leijen, D.: Type directed compilation of row-typed algebraic effects. In: POPL. pp. 486–499. ACM (2017)
15. Levy, P.B., Power, J., Thielecke, H.: Modelling environments in call-by-value programming languages. Inf. Comput. **185**(2), 182–210 (2003)
16. Lindley, S., McBride, C., McLaughlin, C.: Do be do be do. In: POPL. pp. 500–514. ACM (2017)
17. Meijer, E., Fokkinga, M.M., Paterson, R.: Functional programming with bananas, lenses, envelopes and barbed wire. In: FPCA. Lecture Notes in Computer Science, vol. 523, pp. 124–144. Springer (1991)
18. Plotkin, G.D.: Call-by-name, call-by-value and the lambda-calculus. Theor. Comput. Sci. **1**(2), 125–159 (1975)
19. Plotkin, G.D., Power, J.: Adequacy for algebraic effects. In: FoSSaCS. LNCS, vol. 2030, pp. 1–24. Springer (2001)
20. Plotkin, G.D., Pretnar, M.: Handling algebraic effects. Logical Methods in Computer Science **9**(4) (2013)
21. Reynolds, J.C.: Definitional interpreters for higher-order programming languages. Higher-Order and Symbolic Computation **11**(4), 363–397 (1998)
22. Wu, N., Schrijvers, T.: Fusion for free - efficient algebraic effect handlers. In: MPC. Lecture Notes in Computer Science, vol. 9129, pp. 302–322. Springer (2015)
23. Yallop, J.: Staged generic programming. PACMPL **1**(ICFP), 29:1–29:29 (2017)

A Kinding and typing rules for $\lambda^\dagger$

The kinding rules for $\lambda^\dagger$ are given in Fig. 9 and the typing rules are given in Fig. 10.

$$\textsc{TyVar}\ \frac{}{\Delta, \alpha : K \vdash \alpha : K} \qquad \textsc{Forall}\ \frac{\Delta, \alpha : K \vdash C : \mathsf{Comp}}{\Delta \vdash \forall \alpha^K.\, C : \mathsf{Type}}$$

$$\textsc{Comp}\ \frac{\Delta \vdash A : \mathsf{Type} \quad \Delta \vdash E : \mathsf{Effect}}{\Delta \vdash A!E : \mathsf{Comp}} \qquad \textsc{Fun}\ \frac{\Delta \vdash A : \mathsf{Type} \quad \Delta \vdash C : \mathsf{Comp}}{\Delta \vdash A \to C : \mathsf{Type}}$$

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

$$\textsc{Present}\ \frac{\Delta \vdash A : \mathsf{Type}}{\Delta \vdash \mathsf{Pre}(A) : \mathsf{Presence}} \qquad \textsc{Absent}\ \frac{}{\Delta \vdash \mathsf{Abs} : \mathsf{Presence}}$$

$$\textsc{EmptyRow}\ \frac{}{\Delta \vdash \cdot : \mathsf{Row}_\mathcal{L}} \qquad \textsc{ExtendRow}\ \frac{\Delta \vdash P : \mathsf{Presence} \quad \Delta \vdash R : \mathsf{Row}_{\mathcal{L} \uplus \{\ell\}}}{\Delta \vdash \ell : P; R : \mathsf{Row}_\mathcal{L}}$$

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

Fig. 9: Kinding rules for $\lambda^\dagger$

Values

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

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

$$\textsc{T-Inject}\ \frac{\Delta; \Gamma \vdash V : A}{\Delta; \Gamma \vdash (\ell\ V)^{R} : [\ell : \mathsf{Pre}(A); R]}$$

Computations

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

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

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

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

Handlers

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

Fig. 10: Typing rules for $\lambda^{\dagger}$

B Empirical Results

We evaluated the performance of native deep handlers, native shallow handlers, and the two inter-encodings (§3) on the CEK machine (§4) and CPS translation (§5) using an experimental branch of the Links web programming language [2]. We measured the execution time of the implementations on a set of benchmarks, and empirically verified that both the CEK machine and CPS translation do not cause memory leaks for native deep and shallow programs. The deep encoding of shallow handlers leaks memory, however, which is the expected behaviour as each handling of an operation causes a new handler to be installed, whilst the original handler remains live propagating subsequent operation invocations.

B.1 Experimental Setup

The experiments were conducted using a PC with quad-core Intel Xeon CPU E5-1620 v2 running at 3.70 GHz and 32 GB of RAM, using an experimental branch of Links 0.7.3 on Ubuntu 16.04 LTS. The experimental branch contains an implementation of the CEK machine (§4) in JavaScript along with a translator which reflects Links source programs in JavaScript. In addition the branch contains an implementation of the CPS translation (§5) which also targets JavaScript.

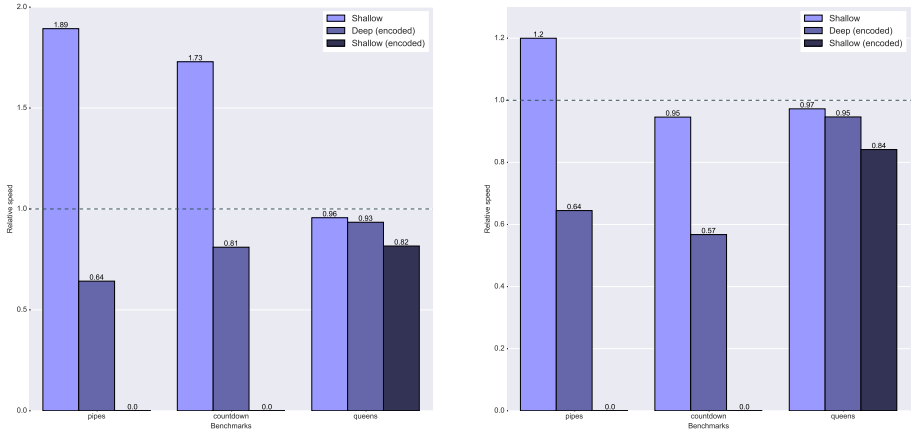
We have adapted the Haskell *countdown*, *pipes*, and *n-Queens* benchmarks from Kammar et al. [11] to Links. We have implemented four variations of the benchmarks, one for each kind of handler implementation, i.e. native deep, native shallow, and using the respective encodings of shallow and deep handlers (§3). Each benchmark was sampled fifteen times on the V8 engine (version 6.8), which is the JavaScript engine powering the Chrome web browser (version 68). We have performed two sets of experiments. The first set of experiments measures the median relative execution time between the different variations of each benchmark. The second set of experiments measures the memory consumption of each handler implementation strategy for the pipes benchmark.

All performance testing on V8 was done using the `--stack-size 13102` and `--use-strict` compiler flags, which fixes the call stack size to 128 MB and forces strict mode interpretation of JavaScript code, respectively. To compensate for the lack of tail call elimination in V8, the CPS translation was instrumented to emit trampolined code.

B.2 Time Performance Results

The results for the median relative execution time for the benchmarks on the CEK machine are shown in Fig. 11a, whilst the results for the CPS benchmarks are depicted in Fig. 11b. For each benchmark the result is shown relative to the respective native deep handler implementation.

Pipes. The pipes benchmark constructs a pipeline consisting of 2^{10} nested subpipes, and pushes 1000 integers through it. The native shallow version performs consistently better than the native deep handler version. On the CEK machine



(a) CEK (b) CPS

Fig. 11: Relative median performance

the shallow version is 1.89× better, whilst for the CPS code it is 1.2× better. These results conform with the results of Kammar et al. [11]. The performance difference is due to the deep encoding of pipes as it does allocate more closures. Thus, the shallow encoding of the native deep handler program suffers from a "double encoding", i.e. firstly it is penalised by the deep encoding of pipes and then secondly by the shallow encoding of deep handlers. The deep encoding of shallow pipes runs out of memory before producing a result.

Countdown. The countdown program iteratively decrements an ambient state parameter from 10^6 to 0. The state is implemented using two operations $\mathsf{Get} : \mathsf{Int}$ and $\mathsf{Put} : \mathsf{Int} \to \langle\rangle$ for reading and writing the state parameter, respectively. Thus the program performs a total of $2 * 10^6$ operations in a tight loop.

The deep state handler gives the standard functional state-passing interpretation of Get and Put, which means that the handler initially returns a closure that takes as input the initial state. Therefore each invocation of the deep resumption returns a new closure that takes the updated state as its parameter which causes many closure allocations during evaluation. The shallow formulation of state-passing need not return a closure every time, rather, it takes the (initial) state as input in addition to the stateful computation. The shallow state handler is 1.73× faster than the deep handler on the CEK machine, whilst it is marginally slower in the CPS setting.

Queens. The Queens program is the classic n-Queens problem which we have implemented using a single nondeterministic nary operation $\mathsf{Choose} : [a] \to a$. The handler for Choose returns the first correct solution to the n-Queens problem. We tested the implementation with $n = 12$. The baseline deep handler program is marginally faster than the shallow and deep encoded variants. This

Variation	Memory usage (MB)
Baseline (V8)	42.10
Shallow	1032.948
Deep	1355.50
Deep (encoded)	1775.908
Shallow (encoded)	more than 2196.63

(a) CEK

Variation	Memory usage (MB)
Baseline (V8)	23.50
Shallow	60.48
Deep	61.92
Deep (encoded)	63.04
Shallow (encoded)	more than 2191.10

(b) CPS

Table 1: Peak Physical Memory Usage

is the only benchmark where the deep encoding of shallow handlers did not run out of memory before producing a (correct) result.

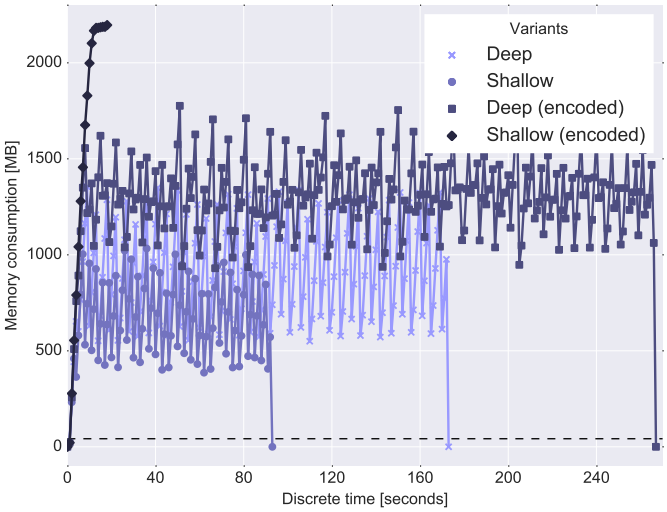
B.3 Memory Performance Results

We measure the memory consumed by the four different variations of the pipes benchmarks using the CEK machine and CPS translation. The default JavaScript maximum heap size on the test machine is roughly 2.2 GB.

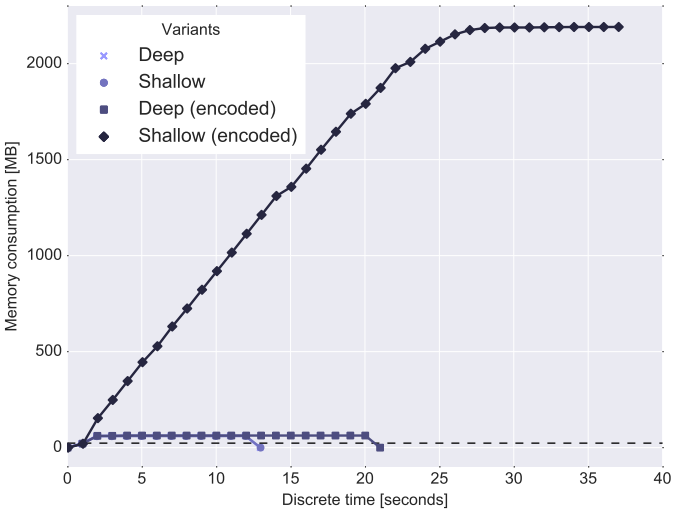
CEK. The baseline memory consumption for the CEK machine on V8 was obtained by observing the memory usage by running a simple tail-recursive function that would call itself infinitely. The baseline memory consumption and the peak memory usage for each of four pipes programs are given in Table 1a. The memory usage pattern of the four pipes programs is depicted in Fig. 12a

The only program to run out of memory is variant using the deep encoding of shallow handlers. For this particular program the JavaScript engine simply halts execution and fails with an out of memory error. The other three programs finish their execution. Moreover, they exhibit a similar oscillating memory pattern. The rapid increase in memory consumption is due to CEK machine allocating many objects during each machine-loop iteration. Dually, the rapid decrease is due to the objects being garbage collected.

CPS. The baseline memory consumption for V8 was obtained by observing the memory usage of a simple infinite loop with an empty loop-body written in native JavaScript. Table 1b summarises the peak memory usage and Fig. 12b depicts the memory usage for each variation of the pipes benchmark. The memory footprint of the native deep and shallow handlers and encoded deep handlers is roughly the same. The deep encoding of shallow handlers runs out of memory. In contrast to the CEK machine, memory usage for the CPS translated programs oscillate much less, in fact, it remains nearly constant for the programs that do not leak memory. We believe the reason for this behaviour is that CPS run-time allocates only small objects which are typically the size of a single cons-cell containing a function pointer, and quickly becomes available for garbage collection.



(a) CEK



(b) CPS

Fig. 12: Physical Memory Consumption