Under consideration for publication in J. Functional Programming

Effect Handlers via Generalised Continuations

DANIEL HILLERSTRÖM
Laboratory for Foundations of Computer Science, The University of Edinburgh, UK
(*e-mail:* daniel.hillerstrom@ed.ac.uk)

SAM LINDLEY
Laboratory for Foundations of Computer Science, The University of Edinburgh, UK
Department of Computing, Imperial College London, UK
(*e-mail:* sam.lindley@ed.ac.uk)

ROBERT ATKEY
Mathematically Structured Programming Group, University of Strathclyde, UK
(*e-mail:* robert.atkey@strath.ac.uk)

Abstract

Plotkin and Pretnar's effect handlers offer a versatile abstraction for modular programming with user-defined effects. This paper focuses on foundations for implementing effect handlers, for the three different kinds of effect handlers that have been proposed in the literature: deep, shallow, and parameterised.

Traditional deep handlers are defined by folds over computation trees, and are the original construct proposed by Plotkin and Pretnar. Shallow handlers are defined by case splits (rather than folds) over computation trees. Parameterised handlers are deep handlers extended with a state value that is threaded through the folds over computation trees. We formulate the extensions both directly and via encodings in terms of deep handlers, and illustrate how the direct implementations avoid the generation of unnecessary closures.

We give two distinct foundational implementations of all the kinds of handlers we consider: a continuation passing style (CPS) transformation and a CEK-style abstract machine. In both cases, the key ingredient is a generalisation of the notion of continuation to accommodate stacks of effect handlers. We obtain our CPS translation through a series of refinements as follows. 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 by uncurrying it to yield a properly tail-recursive translation, and then moving towards more and more intensional representations of continuations in order to support different kinds of effect handlers. Finally, we make the translation higher-order in order to contract administrative redexes at translation time. Our abstract machine design then uses the same generalised continuation representation as the CPS translation. We have implemented both the abstract machine and the CPS transformation (plus extensions) as backends for the Links web programming language.

1 Introduction

Effect handlers provide a modular and structured interface for programming with delimited control. They subsume contemporary control idioms such as async/await and generators and iterators directly. The aforementioned control idioms provide a restricted form of de-

limited control (James & Sabry, 2011). In fact general delimited control operators such as shift/reset turn out to be instances of effect handlers (Forster *et al.*, 2017, 2019; Piróg *et al.*, 2019). In contrast to many control abstractions, effect handlers have a strong mathematical foundation (Plotkin & Power, 2001; Plotkin & Pretnar, 2013), yet their practical relevance is inescapable as they have been applied across a wide spectrum of diverse programming disciplines including asynchronous programming (Leijen, 2017b; Dolan *et al.*, 2017), concurrent programming (Dolan *et al.*, 2015), probabilistic programming (Bingham *et al.*, 2018), meta programming (Yallop, 2017), and modular program construction (Kammar *et al.*, 2013).

Effect handlers come in two flavours *deep* and *shallow*. Deep handlers are defined by folds (specifically *catamorphisms* (Meijer *et al.*, 1991)) 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 (Wu & Schrijvers, 2015). 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.

Effect handlers enjoy rather simple static and dynamic semantics, providing the basis for many feasible implementation strategies. For instance, Kammar *et al.* (2013) implement effect handlers as libraries by making variously use of free monads, continuation monads, and delimited continuations. By contrast, Multicore OCaml (Dolan *et al.*, 2015) uses a form of segmented stacks (Bruggeman *et al.*, 1996) to provide an efficient native implementation. Explicit stack manipulation is appealing when one has complete control over the design of the backend. Similarly, delimited continuations are appealing when the backend has support for delimited continuations (Kammar *et al.*, 2013; Kiselyov & Sivaramakrishnan, 2016). In this paper we study two foundational implementation strategies.

1. We translate effect handlers from a rich source lambda calculus into a plain lambda calculus. Specifically, we study *continuation passing style* (CPS) transformations for effect handlers. The benefit of CPS is that we require no primitives in the target language to support effect handlers, meaning CPS is a good fit for targeting backends where one has little or no control of runtime. CPS is also an established intermediate representation used by compilers (Kranz *et al.*, 1986; Appel, 1992; Kennedy, 2007), which makes 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 key to implementing a CPS transformation for effect handlers is to generalise the notion of continuation to model a stack of effect handlers combined with their associated local continuations.
2. We study an *abstract machine* with simultaneous support for deep and shallow handlers which is based on the CEK machine of Felleisen & Friedman (1986). As with the CPS transform, we generalise the usual notion of continuation to a stack of effect handlers combined with their local continuations in order to construct a machine that supports effect handlers.

Both implementation strategies form the basis for our implementation of effect handlers in Links (Hillerström, 2015). Links is a single-source, tierless, strict ML-like functional web programming language with Hindley-Milner type inference and a type-and-effect

system based on row polymorphism (Cooper *et al.*, 2006). The compiler slices a given source program into three parts: a part that compiles to JavaScript using a CPS transform to run on the client (in a web browser), and a second part that runs on the server, which is implemented as an abstract machine, and the third part which comprises database queries that are compiled directly to SQL and executed on a database.

The main contributions of this paper are as follows:

- a tutorial on modular effectful programming with effect handlers (Section 2) that demonstrates the three kinds of effect handlers we study in this paper: deep, shallow, and parameterised;
- a fine-grain call-by-value calculus $\lambda^\dagger$ of deep and shallow handlers with an evaluation context based small step operational semantics, which captures the core aspects of our implementation of effect handlers in Links including a row-based effect type system (Section 3);
- encodings back and forth between deep and shallow handlers along with simulations up to suitable notions of administrative reduction (Section 4);
- a higher-order CPS transformation for $\lambda^\dagger$ along with a detailed proof that the CPS transformation implements the operational semantics (Section 5);
- an abstract machine for $\lambda^\dagger$ that implements the operational semantics, based on ideas developed in the CPS translation (Section 6); and
- an extension of $\lambda^\dagger$ with parameterised handlers, along with a proof that parameterised handlers can be implemented by a local translation into deep handlers (Section 7).

Section 8 describes our implementation of effect handlers in Links, Section 9 discusses related work, and Section 10 concludes.

Relation to Prior Publications This paper combines and streamlines the main results of three previously published papers (Hillerström & Lindley, 2016; Hillerström *et al.*, 2017; Hillerström & Lindley, 2018). We have improved the CPS translations described by Hillerström *et al.* (2017) and Hillerström & Lindley (2018) to remove all administrative reductions, and fixed a minor bug in the implementation of shallow handlers (Section 5). The abstract machine described by Hillerström & Lindley (2016) has been extended to also implement shallow handlers. This paper also provides additional examples of the use of effect handlers (Section 2), more detailed discussion of the design decisions in effect handler calculi, includes detailed proofs for the complex higher order CPS translation, and describes the necessary extensions to support parameterised handlers (Section 7).

2 Modular Effectful Programming with Effect Handlers

In this section, we give a high-level introduction to programming with effect handlers by example. We demonstrate the usefulness of effect handlers as a practical programming abstraction by iteratively developing a modular implementation of the mathematical game Nim (Bouton, 1901). Starting with a fixed abstract model of the game, we show how effect handlers let us obtain different instantiations with ease.

We consider a variation of Nim in which two players, Alice and Bob, take turns to remove between one or three sticks from a heap starting with n sticks. Whoever takes the last stick wins the game. We implement the game in $\lambda^\dagger$, our calculus with effect handlers, formally presented in Section 3, although we allow ourselves a fair amount of syntactic sugar to make the examples more readable (in particular we use direct-style rather than the fine-grain call-by-value discipline of $\lambda^\dagger$).

2.1 Abstract Operations and an Abstract Game Model

We model the Nim game as two mutually recursive functions aliceTurn and bobTurn. The two players are also represented as values of a (polymorphic) type variant with two constructors each denoting either Alice or Bob: $\mathsf{Player} \stackrel{\text{def}}{=} [\mathsf{Alice} \mid \mathsf{Bob}]$. We implement the aliceTurn function as follows:

$$
\begin{aligned}
&\mathsf{aliceTurn} : \mathsf{Int} \to \mathsf{Player}!\{\mathsf{Move} : \langle \mathsf{Player}, \mathsf{Int} \rangle \to \mathsf{Int}, \varepsilon\} \\
&\mathsf{aliceTurn}\ n \stackrel{\text{def}}{=} \mathbf{if}\ n \leq 0\ \mathbf{then}\ \mathsf{Bob} \\
&\qquad\qquad \mathbf{else\ let}\ m \leftarrow \mathbf{do}\ \mathsf{Move}\ \langle \mathsf{Alice}, n \rangle\ \mathbf{in} \\
&\qquad\qquad\quad \mathsf{bobTurn}\ (n - m)
\end{aligned}
$$

Besides some peculiarities in the tail of the type signature and the **do**-construct, the above program ought to look familiar to a functional programmer. The signature states that aliceTurn is a function that takes an integer as input and produces a value of type Player as output. As a side effect of computing the output value the function may perform an effectful operation Move, which is parameterised by a pair consisting of a Player and an integer, and whose result is an integer. The right hand side of the bang (!) is the *effect signature* of the function. In our calculus, effect signatures are represented as rows (Remy, 1993), so we will also refer to them as *effect rows*. The presence of Move indicates that the function is permitted to perform the Move operation with the specified types. The row is terminated with an *effect variable* ε, which can be instantiated with additional operations. As a result, the function aliceTurn may be invoked in a larger effect context that permits more effects than it requires.

In the definition of aliceTurn the parameter n is the current number of sticks on the heap. When it is her turn, Alice first checks whether there are any sticks left on the heap, if it is empty, then she declares Bob the winner. Otherwise, she performs her move. The **do**-construct is the introduction form for effectful operations. The label Move is an *abstract* operation symbol with no predefined semantics. The operation is invoked with a pair containing the label Alice and the current heap state n. The returned value of the operation is another integer (as evident from the effect signature). This integer is intended to denote the number of sticks that the Alice decides to take from the heap. After Alice takes the sticks from the heap the turn passes to Bob. The bobTurn function is the same, but swaps Alices and Bobs:

$$
\begin{aligned}
&\mathsf{bobTurn} : \mathsf{Int} \to \mathsf{Player}!\{\mathsf{Move} : \langle \mathsf{Player}, \mathsf{Int} \rangle \to \mathsf{Int}, \varepsilon\} \\
&\mathsf{bobTurn}\ n \stackrel{\text{def}}{=} \mathbf{if}\ n \leq 0\ \mathbf{then}\ \mathsf{Alice} \\
&\qquad\qquad \mathbf{else\ let}\ m \leftarrow \mathbf{do}\ \mathsf{Move}\ \langle \mathsf{Bob}, n \rangle\ \mathbf{in} \\
&\qquad\qquad\quad \mathsf{aliceTurn}\ (n - m)
\end{aligned}
$$

We define an auxiliary function game for configuring a new game with n sticks on the heap.

$$\mathsf{game} : \mathsf{Int} \to \langle\rangle \to \mathsf{Player}!\{\mathsf{Move} : \langle \mathsf{Player}, \mathsf{Int}\rangle \to \mathsf{Int}, \varepsilon\}$$
$$\mathsf{game}\ n \stackrel{\text{def}}{=} \lambda\langle\rangle.\mathsf{aliceTurn}\ n$$

By convention Alice starts every game. Given an initial number of sticks, the game function returns a suspended computation that starts the game when forced. Without the λ-abstraction to suspend computation, running aliceTurn n as is would cause the evaluation relation ($\rightsquigarrow$) on terms to eventually get stuck in some evaluation context $\mathscr{E}$:

$$\mathsf{aliceTurn}\ n \rightsquigarrow^{+} \mathscr{E}[\mathbf{do}\ \mathsf{Move}\ \langle \mathsf{Alice}, n\rangle], \quad n > 0.$$

Evaluation gets stuck in this configuration because we have not yet provided an instantiation of Move. It is an abstract operation. Thus, we say that game, aliceTurn, and bobTurn are *abstract* computations. In the following sections we will consider several possible interpretations of Move that enable us to instantiate the game with support for alternative strategies, monitoring of cheating players, and exploration of alternative plays of the game. By separating effect operations from their semantics, such different interpretations can be programmed in a modular way.

Syntactic sugar (effect variables) The reader might have observed that the first arrow in the signature for game lacks an effect row. The actual type of game is

$$\mathsf{Int} \to (\langle\rangle \to \mathsf{Player}!\{\mathsf{Move} : \langle \mathsf{Player}, \mathsf{Int}\rangle \to \mathsf{Int}, \varepsilon\})!\{\varepsilon'\},$$

where ε' is a distinct effect variable from ε. The application game n for some number n does not cause any effects, it is *pure*. The presence of the effect variable means that it is parametric in the actual effect context that it is used in. Had game been given the type

$$\mathsf{Int} \to (\langle\rangle \to \mathsf{Player}!\{\mathsf{Move} : \langle \mathsf{Player}, \mathsf{Int}\rangle \to \mathsf{Int}, \varepsilon\})!\emptyset,$$

then it could only be invoked in a pure context. By convention, we omit effect annotations when the effect row is a singleton row with an effect variable that is only mentioned once in the whole signature.

2.2 Deep Handlers and Assigning Strategies to Players

Abstract operations, like Move, have no predefined semantics. The programmer provides them with semantics by writing an *effect handler*. When an abstract operation is invoked, the current continuation is captured and passed to the effect handler. The captured continuation is then exposed to the programmer as a first-class function that can be invoked multiple times, discarded, or stored for later use. By choosing how the continuation of an abstract operation is resumed, the programmer has freedom to choose the particular semantics of the abstract operations. Henceforth we will use the term *resumption* to describe the captured continuation in order to differentiate it from the notion of (generalised) continuation we use in our CPS translations (Section 5) and abstract machine (Section 6).

In our setting we can use effect handlers to encode particular strategies for Alice and Bob. For example, consider the *perfect* strategy defined by $\mathsf{ps} \stackrel{\text{def}}{=} \lambda n.\mathsf{max}\ 1\ (\mathsf{mod}\ n\ 4)$,

where n is the number of sticks left in the heap. We can assign both players the perfect strategy via a handler as follows.

$$\begin{aligned}
&\mathsf{ps_vs_ps} : (\langle\rangle \to \alpha!\{\mathsf{Move} : \langle \mathsf{Player}, \mathsf{Int}\rangle \to \mathsf{Int}, \varepsilon\}) \to \alpha!\{\varepsilon\} \\
&\mathsf{ps_vs_ps}\ m \stackrel{\text{def}}{=} \textbf{handle}\ m\ \langle\rangle\ \textbf{with} \\
&\qquad\qquad \textbf{return}\ x \qquad\qquad\quad \mapsto x \\
&\qquad\qquad \mathsf{Move}\ \langle _, n\rangle\ \mathit{resume} \mapsto \mathit{resume}\ (\mathit{ps}\ n)
\end{aligned}$$

The function ps_vs_ps (an abbreviation for *perfect-strategy-vs-perfect-strategy*) embodies the handler. The signature of ps_vs_ps tells us, that the function takes as input a (suspended) computation that may perform the Move operation and ultimately return a value of type α. The type of the *value* returned by ps_vs_ps is the same as the return type of its input computation. However, the Move operation has been removed from its effect signature. This signifies that the Move operations invoked by the argument have been instantiated with a concrete interpretation; they have been *handled*.

Syntactic sugar (presence polymorphism) Formally the type signature of ps_vs_ps is $(\langle\rangle \to \alpha!\{\mathsf{Move} : \langle \mathsf{Player}, \mathsf{Int}\rangle \to \mathsf{Int}, \varepsilon\}) \to \alpha!\{\mathsf{Move} : \theta, \varepsilon\}$, where the return type explicitly mentions the Move operation. The convention we adopt is to allow such presence-polymorphic operations to be omitted from type signatures when they can be inferred from the rest of the signature[1]. Here we see that ε appears in an effect row containing the Move operation; omitting the Move operation in the other effect row in the signature denotes that Move is associated with a fresh presence variable.

The definition of ps_vs_ps uses the **handle** M **with** H construct to instantiate the Move operation. The **handle**-construct is the elimination form for effectful operations. It runs a computation M and interprets its effectful operations according to the handler definition H. Any handler definition consists of a **return** clause and a collection of operation clauses. The **return** clause defines how to handle the final return value of the input computation. In this example, we simply return the final value returned by the computation. The operation cases define how to interpret operations that may occur in the computation. The left hand side of an operation case matches on the particular label of an operation (in our instance Move) and the value that it carries. In addition, the left hand side also provides a name for the resumption. By convention, we call it *resume*. The right hand side of an operation case defines the dynamic semantics of the operation. Here, we interpret Move, regardless of the player, as playing the perfect strategy by invoking the resumption *resume* with the value determined by ps n. The application of *resume* transfers control back to the invocation site of Move and substitutes its argument for the whole operation invocation term.

Using this handler, we can compute the winner of a single game, for instance:

$$\mathsf{ps\_vs\_ps}\ (\mathsf{game}\ 7) \rightsquigarrow^{+} \mathsf{Alice} \quad \text{and} \quad \mathsf{ps\_vs\_ps}\ (\mathsf{game}\ 8) \rightsquigarrow^{+} \mathsf{Bob}.$$

The handler handles *all* invocations of Move. This can be seen by examining the type of the resumption *resume*: $\mathsf{Int} \to \alpha!\{\varepsilon\}$. The type is determined by the enclosing handler and the signature of the operation. The input type is the return type of the operation Move, and the

[1] This syntactic sugar is also available in our concrete implementation in Links.

return type is the body type of the handler. Similarly, the effect signature is the same as its enclosing handler. This reveals that *resume* handles any subsequent occurrences of Move. It handles those operations by implicitly re-wrapping the handler around the remainder of the computation (i.e. the computation following the operation invocation), until the **return** clause is invoked by the computation finishing.

The following sketched reduction sequence illustrates how the handler is re-wrapped by giving the interesting steps of the computation ps_vs_ps (game 7). We let $H_{\mathsf{ps\_vs\_ps}}$ denote the handler definition.

$$
\begin{array}{ll}
 & \mathsf{ps\_vs\_ps}\ (\mathsf{game}\ 7) \\
\rightsquigarrow^{+} & \quad (\text{definition of } \mathsf{ps\_vs\_ps}) \\
 & \mathbf{handle}\ (\mathsf{game}\ 7)\ \langle\rangle\ \mathbf{with}\ H_{\mathsf{ps\_vs\_ps}} \\
\rightsquigarrow^{+} & \quad (\text{definition of game 7 with } \mathscr{E} = \mathbf{let}\ m \leftarrow [\,]\ \mathbf{in}\ \mathsf{bobTurn}(7-m)) \\
 & \mathbf{handle}\ \mathscr{E}[\mathbf{do}\ \mathsf{Move}\ \langle \mathsf{Alice}, 7\rangle]\ \mathbf{with}\ H_{\mathsf{ps\_vs\_ps}} \\
\rightsquigarrow^{+} & \quad (\mathsf{Move}\ \text{clause definition}) \\
 & (\mathit{resume}\ (\mathsf{ps}\ 7))[\lambda x.\mathbf{handle}\ \mathscr{E}[x]\ \mathbf{with}\ H_{\mathsf{ps\_vs\_ps}}/\mathit{resume}]
\end{array}
$$

The handler forces evaluation of the suspended game computation. After some amount of standard reduction steps the redex is **do** Move $\langle \mathsf{Alice}, 7\rangle$. At this point control gets transferred to the handler, specifically the Move clause within the handler definition. The resumption *resume* is substituted for a lambda abstraction, whose body contains the same handler enclosing the remainder of the evaluation context $\mathscr{E}$. As a consequence, any subsequent invocation of Move gets handled in the same manner. This handling idiom is known as *deep* handlers. This behaviour is analogous to how *folds* (catamorphisms) in functional programming work. Evaluation continues by invoking the resumption with 3.

$$
\begin{array}{ll}
\rightsquigarrow^{+} & \quad (\text{resuming with } \mathsf{ps}\ 7 = 3) \\
 & \mathbf{handle}\ \mathscr{E}[3]\ \mathbf{with}\ H_{\mathsf{ps\_vs\_ps}} \\
\rightsquigarrow^{+} & \quad (\text{definition of } \mathsf{bobTurn}(4) \text{ with } \mathscr{E}' = \mathbf{let}\ m \leftarrow [\,]\ \mathbf{in}\ \mathsf{aliceTurn}(4-m)) \\
 & \mathbf{handle}\ \mathscr{E}'[\mathbf{do}\ \mathsf{Move}\ \langle \mathsf{Bob}, 4\rangle]\ \mathbf{with}\ H_{\mathsf{ps\_vs\_ps}} \\
\rightsquigarrow^{+} & \quad (\mathsf{Move}\ \text{clause definition}) \\
 & (\mathit{resume}\ (\mathsf{ps}\ 4))[\lambda x.\mathbf{handle}\ \mathscr{E}'[x]\ \mathbf{with}\ H_{\mathsf{ps\_vs\_ps}}/\mathit{resume}] \\
\rightsquigarrow^{+} & \quad (\text{resuming with } \mathsf{ps}\ 4 = 1) \\
 & \mathbf{handle}\ \mathscr{E}'[1]\ \mathbf{with}\ H_{\mathsf{ps\_vs\_ps}} \\
\rightsquigarrow^{+} & \quad (\text{definition of } \mathsf{aliceTurn}(3) \text{ with } \mathscr{E}'' = \mathbf{let}\ m \leftarrow [\,]\ \mathbf{in}\ \mathsf{bobTurn}(3-m)) \\
 & \mathbf{handle}\ \mathscr{E}''[\mathbf{do}\ \mathsf{Move}\ \langle \mathsf{Alice}, 3\rangle]\ \mathbf{with}\ H_{\mathsf{ps\_vs\_ps}} \\
\rightsquigarrow^{+} & \quad (\mathsf{Move}\ \text{clause definition}) \\
 & (\mathit{resume}\ (\mathsf{ps}\ 3))[\lambda x.\mathbf{handle}\ \mathscr{E}''[x]\ \mathbf{with}\ H_{\mathsf{ps\_vs\_ps}}/\mathit{resume}] \\
\rightsquigarrow^{+} & \quad (\text{resuming with } \mathsf{ps}\ 3 = 3) \\
 & \mathbf{handle}\ \mathscr{E}''[3]\ \mathbf{with}\ H_{\mathsf{ps\_vs\_ps}} \\
\rightsquigarrow^{+} & \quad (\mathsf{bobTurn}(0) = \mathsf{Alice}) \\
 & \mathbf{handle}\ \mathsf{Alice}\ \mathbf{with}\ H_{\mathsf{ps\_vs\_ps}} \\
\rightsquigarrow^{+} & \quad (\text{definition of the } \mathbf{return} \text{ clause}) \\
 & \mathsf{Alice}
\end{array}
$$

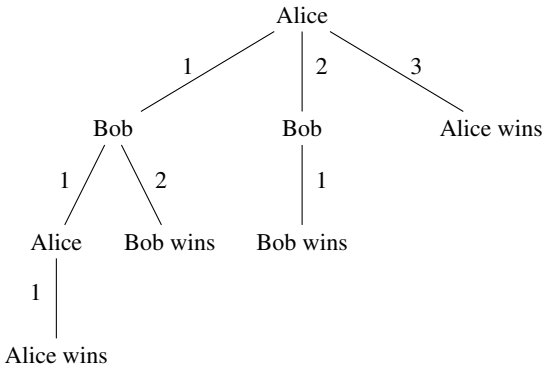

Fig. 1. Game Tree Generated by gameTree (game 3)

An alternative to deep handlers is *shallow* handlers, which do not wrap its handler around the remaining evaluation context $\mathscr{E}$. Instead it is up to the programmer to decide how to handle further operations. We present and give examples of shallow handlers in Section 2.6.

We can assign the players different strategies by pattern matching on the player identifiers. For example, we can assign Bob a *cheating strategy* such as taking all the remaining sticks on the heap, thereby winning in a single move. The following handler, ps_vs_cs, assigns the perfect strategy to Alice and the cheating strategy to Bob.

$$
\begin{aligned}
&\mathsf{ps\_vs\_cs} : (\langle\rangle \to \alpha!\{\mathsf{Move} : \langle \mathsf{Player}, \mathsf{Int}\rangle \to \mathsf{Int}, \varepsilon\}) \to \alpha!\{\varepsilon\} \\
&\mathsf{ps\_vs\_cs}\ m \stackrel{\text{def}}{=} \textbf{handle}\ m\ \langle\rangle\ \textbf{with} \\
&\qquad\qquad \textbf{return}\ x \qquad\qquad\qquad \mapsto x \\
&\qquad\qquad \mathsf{Move}\ \langle \mathsf{Alice}, n\rangle\ \mathit{resume} \mapsto \mathit{resume}\ (\mathsf{ps}\ n) \\
&\qquad\qquad \mathsf{Move}\ \langle \mathsf{Bob}, n\rangle\ \ \mathit{resume} \mapsto \mathit{resume}\ n
\end{aligned}
$$

Using this handler, Bob wins whenever $n \geq 4$. In Section 2.4 we will show how to handle cheaters in a modular fashion using another effect handler.

2.3 Multi-shot Resumptions and Computing Game Data

The handlers ps_vs_ps and ps_vs_cs compute the winner of a single game under fixed strategies for the players by invoking their respective resumption exactly once per operation invocation. However, by invoking the resumption multiple times, we can compute the outcomes of all possible legal strategies for a single game. As a concrete example we will demonstrate how to compute the game tree of a particular game. We define a type of game tree inductively as:

$$\mathsf{GameTree} \stackrel{\text{def}}{=} [\mathsf{Take} : \langle \mathsf{Player}, \mathsf{List}\ \langle \mathsf{Int}, \mathsf{GameTree}\rangle\rangle \mid \mathsf{Winner} : \mathsf{Player}].$$

Each path from the root of a game tree to one of its leaves induces a particular sequence of moves. The interior nodes are given by Take constructors, which carry with them information about whose turn it was, and all the possible moves (and outcomes) that the player can possibly commit to. The leaves of a game tree are of the form Winner p, where

p is the winning player resulting from that path through the tree. The following handler computes the game tree for any given game.

$$\begin{array}{l}\mathsf{gameTree} : (\langle\rangle \to \mathsf{Player}!\{\mathsf{Move} : \langle \mathsf{Player}, \mathsf{Int}\rangle \to \mathsf{Int}, \varepsilon\}) \to \mathsf{GameTree}!\{\varepsilon\} \\ \mathsf{gameTree}\ m \stackrel{\text{def}}{=} \textbf{handle}\ m\ \langle\rangle\ \textbf{with} \\ \qquad \begin{array}{ll} \textbf{return}\ x & \mapsto \mathsf{Winner}\ x \\ \mathsf{Move}\ \langle p, n\rangle\ \mathit{resume} & \mapsto \textbf{let}\ \mathit{moves} \leftarrow \mathsf{legalMoves}\ n\ \textbf{in} \\ & \quad \textbf{let}\ \mathit{subgames} \leftarrow \mathsf{map}\ \mathit{resume}\ \mathit{moves}\ \textbf{in} \\ & \quad \textbf{let}\ \mathit{subtrees} \leftarrow \mathsf{zip}\ \mathit{moves}\ \mathit{subgames}\ \textbf{in} \\ & \quad \mathsf{Take}\ \langle p, \mathit{subtrees}\rangle \end{array}\end{array}$$

The **return** clause wraps the winning player x with a Winner constructor. The operation case computes every subgame by invoking the resumption *resume* with every possible legal move. The result of every subgame is reified as a subtree in the game tree. In order to compute the subgames and subtree, we make use of standard list functions *map* and *zip* for transforming a list and point-wise joining two lists, respectively. The auxiliary function legalMoves ensures that the resumption *resume* is only applied to legal moves. For Nim, we define it using another standard list operator filter: $\mathsf{legalMoves} \stackrel{\text{def}}{=} \lambda n.\mathit{filter}\ (\leq n)\ [1,2,3]$. Figure 1 visualises the result computed by gameTree (game 7). Without modifying the underlying game model we have been able to compute data about particular games by simply reinterpreting the abstract operation Move.

2.4 Effect Forwarding and Cheat Detection

Thus far we have considered a single operation Move, but in general a computation will have more operations. We could define a monolithic handler that interprets every operation that may occur in a given computation. A more modular alternative is to define a collection of fine-grained, specialised handlers that each handle a particular operation, and then compose them together to fully interpret a computation. Composed handlers can cooperate to interpret an abstract computation. Each handler operates on a particular subset of the abstract operations, leaving the remainder for other handlers.

We will demonstrate how to implement a cheat detection mechanism for the game by composing handlers. The idea is to signal cheating via an abstract operation Cheat. The operation Cheat is parameterised by the identifier of the player that was caught cheating. The return type of the operation is the empty type Zero, which we define as the empty variant type $\mathsf{Zero} \stackrel{\text{def}}{=} []$. We define a convenient function that raises the signal.

$$\begin{array}{l}\mathsf{cheat} : \mathsf{Player} \to \alpha!\{\mathsf{Cheat} : \mathsf{Player} \to \mathsf{Zero}, \varepsilon\} \\ \mathsf{cheat}\ p \stackrel{\text{def}}{=} \textbf{case}\ (\textbf{do}\ \mathsf{Cheat}\ p)\ \{\ \}\end{array}$$

We eliminate values of type Zero using an empty **case** construct. This allows us to ascribe a polymorphic return type to cheat. From the point of view of the computation, an invocation of Cheat will never return. Correspondingly, since there are no values of type Zero, it is not possible for a handler to invoke the resumption to continue execution after Cheat is invoked. Thus, the operation Cheat acts like an exception, and an interpretation of Cheat in an effect handler amounts to implementing an exception handler. A potential implementation of such a handler handles cheating players by assigning victory to the

opponent player, who wins by default:

$$\begin{aligned}
&\mathsf{defaultVictory} : (\langle\rangle \to \mathsf{Player}!\{\mathsf{Cheat} : \mathsf{Player} \to \mathsf{Zero}, \varepsilon\}) \to (\langle\rangle \to \mathsf{Player}!\{\varepsilon\}) \\
&\mathsf{defaultVictory}\ m \stackrel{\text{def}}{=} \lambda\langle\rangle.\mathbf{handle}\ m\ \langle\rangle\ \mathbf{with} \\
&\qquad\qquad\qquad\qquad \mathbf{return}\ x \quad\ \mapsto x \\
&\qquad\qquad\qquad\qquad \mathsf{Cheat\ Alice\ _} \mapsto \mathsf{Bob} \\
&\qquad\qquad\qquad\qquad \mathsf{Cheat\ Bob\ _} \ \mapsto \mathsf{Alice}
\end{aligned}$$

Next, we implement a handler that monitors a given game and raises Cheat if any player is caught cheating.

$$\begin{aligned}
&\mathsf{monitor} : \quad (\langle\rangle \to \alpha!\{\mathsf{Cheat} : \mathsf{Player} \to \mathsf{Zero}, \mathsf{Move} : \langle \mathsf{Player}, \mathsf{Int}\rangle \to \mathsf{Int}, \varepsilon\}) \\
&\qquad\qquad \to (\langle\rangle \to \alpha!\{\mathsf{Cheat} : \mathsf{Player} \to \mathsf{Zero}, \mathsf{Move} : \langle \mathsf{Player}, \mathsf{Int}\rangle \to \mathsf{Int}, \varepsilon\}) \\
&\mathsf{monitor}\ m \stackrel{\text{def}}{=} \lambda\langle\rangle.\mathbf{handle}\ m\ \langle\rangle\ \mathbf{with} \\
&\qquad\qquad\qquad\qquad \mathbf{return}\ x \qquad\qquad\ \mapsto x \\
&\qquad\qquad\qquad\qquad \mathsf{Move}\ \langle p, n\rangle\ \mathit{resume} \mapsto \mathbf{let}\ n' \leftarrow \mathbf{do}\ \mathsf{Move}\ \langle p, n\rangle\ \mathbf{in} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathbf{if}\ n' \in \mathsf{legalMoves}\ n\ \mathbf{then}\ \mathit{resume}\ n' \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathbf{else}\ \mathsf{cheat}\ p
\end{aligned}$$

The **return** clause forwards on the result of the computation. The interesting case is the operation case. To handle Move $\langle p, n\rangle$, the handler re-performs Move with the same parameters. That is, the handler explicitly forwards the operation to another enclosing handler. The result is stored in n' which is checked against the legal moves. If n' is a legal move, then the game continues. Otherwise, the handler raises the Cheat signal.

The input and output effect rows of monitor are identical. The operation Move appears in both rows, because the handler handles invocations of Move in the input computation by performing another invocation of Move, thus introducing the operation into the output row. The Cheat operation appears in the output row, because the handler performs the operation. The operation propagates to the input row to ensure that if the computation performs the operation, it has the same type. One may regard this property as introducing imprecision. However, it is a necessary artefact of the particular form of row polymorphism (Remy, 1993) that our effect system is based upon. As in our effect system, labels must distinct. Other effect systems have been designed which allow programmers to be more precise about effects generated by the input and output; one example is the Frank language (Lindley *et al.*, 2017; Convent *et al.*, 2020).

Composing the handlers defaultVictory and monitor with ps_vs_cs means Bob is caught cheating whenever $n \geq 5$, and thus Alice is declared the winner. Regular function composition composes handlers:

$$(\mathsf{ps\_vs\_cs} \circ \mathsf{defaultVictory} \circ \mathsf{monitor})\ (\mathsf{game}\ 7) \rightsquigarrow^{+} \mathsf{Alice}$$

The reader may have observed that the defaultVictory handler does not handle the Move operation. The above composition works because all handlers implicitly forward operations that they do not handle to their nearest, dynamically enclosing handler. This implicit forwarding behaviour is known as *effect forwarding*. Unlike explicit forwarding, as in monitor, implicit forwarding does not affect the typing of handlers.

2.5 Handling Stateful Computations and Instrumentation

In this section, we will show how to instrument a computation with logging using effect handlers. We provide an interface for accessing and updating the value of a state cell: $\mathsf{State}\ \beta \stackrel{\text{def}}{=} \{\mathsf{Get} : \langle\rangle \to \beta, \mathsf{Put} : \beta \to \langle\rangle\}$. The interface consists of the operation Get which accesses the value of type β, and Put which updates the value. The following handler provides an implementation of this interface.

$$
\begin{array}{l}
\mathsf{runState} : \beta \to (\langle\rangle \to \langle\alpha, \beta\rangle!\{\mathsf{State}\ \beta, \varepsilon\}) \to (\langle\rangle \to \langle\alpha, \beta\rangle!\{\varepsilon\}) \\
\mathsf{runState}\ \mathit{init}\ m \stackrel{\text{def}}{=} \lambda\langle\rangle.\mathbf{let}\ \mathit{run} \leftarrow \mathbf{handle}\ m\ \langle\rangle\ \mathbf{with} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{return}\ x \qquad\quad \mapsto \lambda \mathit{st}.\langle x, \mathit{st}\rangle \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{Get}\ \langle\rangle\ \mathit{resume} \mapsto \lambda \mathit{st}.(\mathit{resume}\ \mathit{st})\ \mathit{st} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{Put}\ \mathit{st}'\ \mathit{resume} \mapsto \lambda\_.(\mathit{resume}\ \langle\rangle)\ \mathit{st}' \\
\qquad\qquad\qquad\qquad\qquad \mathbf{in}\ \mathit{run}\ \mathit{init}
\end{array}
$$

The runState handler provides a generic way to interpret any stateful computation. It takes as its first parameter the initial value of the state cell. The second parameter is a stateful computation that may perform the Get and Put operations. Ultimately, the handler returns the value of the input computation along with the current value of the state cell.

This formulation of state handling is analogous to the standard monadic implementation of state handling (Wadler, 1995). In the context of handlers, the implementation uses a technique known as *parameter-passing* (Pretnar, 2015). The operations Get and Put are interpreted as functions that take the current state as input. Consequently, *resume* returns a function that expects to be passed the state for the rest of the computation. In the implementation we explicitly paranthesise invocations of *resume* to emphasise that it is an unary function returning another unary function. For example, the type of *resume* in the Get case is $\beta \to \beta \to \langle\alpha, \beta\rangle!\{\varepsilon\}$. The resumption threads the state value through to the subsequent activation of the handler via its second argument. In the Get case the state value is passed unchanged, whereas in the Put case the provided value st' is passed as the new state. A similar interpretation is given to the **return** case, although, in this case the function takes the *final* state as input and returns a pair consisting of the return value and the final state value.

Operationally, evaluation of the sub-computation m gets suspended when it either invokes an operation or returns a value upon which the corresponding clause in the handler definition returns a state accepting function. This function gets bound to *run* which is subsequently applied to the initial state *init*, thereby continuing evaluation of the stateful fragment of m.

Next, we implement a stateful handler computation, which intercepts and records move operations.

$$
\begin{array}{l}
\mathsf{history} : \quad (\langle\rangle \to \alpha!\{\mathsf{Move} : \langle\mathsf{Player}, \mathsf{Int}\rangle \to \mathsf{Int}, \mathsf{State}\ (\mathsf{List}\ \langle\mathsf{Player}, \mathsf{Int}\rangle), \varepsilon\}) \\
\qquad\quad \to (\langle\rangle \to \alpha!\{\mathsf{Move} : \langle\mathsf{Player}, \mathsf{Int}\rangle \to \mathsf{Int}, \mathsf{State}\ (\mathsf{List}\ \langle\mathsf{Player}, \mathsf{Int}\rangle), \varepsilon\}) \\
\mathsf{history}\ m \stackrel{\text{def}}{=} \lambda\langle\rangle.\mathbf{handle}\ m\ \langle\rangle\ \mathbf{with} \\
\qquad\qquad\qquad \mathbf{return}\ x \qquad\qquad\quad \mapsto x \\
\qquad\qquad\qquad \mathsf{Move}\ \langle p, n\rangle\ \mathit{resume} \mapsto \mathbf{let}\ n' \leftarrow \mathbf{do}\ \mathsf{Move}\ \langle p, n\rangle\ \mathbf{in} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathbf{do}\ \mathsf{Put}\ (\langle p, n'\rangle :: \mathbf{do}\ \mathsf{Get}\ \langle\rangle);\ \mathit{resume}\ n'
\end{array}
$$

The history handler uses Get and Put operations to accumulate a list of moves performed during a game. The handler works in a similar way to the defaultVictory handler from the previous section. It intercepts the Move operation and immediately re-performs it. The result is stored in n', which gets paired with the active player p, and cons-ed onto the current list of moves, that was retrieved via Get. The resulting list is given as argument to Put, which performs the update. The original invocation of Move is resumed with n'. Plugging everything together, using runState with the empty list $[\,]$ as the initial state, we obtain the winner of the game and the (reversed) list of moves performed by each player.

$$(\mathsf{ps\_vs\_ps} \circ (\mathsf{runState}\ [\,]) \circ \mathsf{history})\ (\mathsf{game}\ 7) \rightsquigarrow^{+} \langle \mathsf{Alice}, [\langle \mathsf{Alice}, 3\rangle, \langle \mathsf{Bob}, 1\rangle, \langle \mathsf{Alice}, 3\rangle]\rangle$$

Moreover, we can compose this logging infrastructure with the cheat detection machinery to witness that Bob is caught cheating.

$$\begin{array}{l}(\mathsf{ps\_vs\_cs} \circ (\mathsf{runState}\ [\,]) \circ \mathsf{history} \circ \mathsf{defaultVictory} \circ \mathsf{monitor})\ (\mathsf{game}\ 7) \\ \rightsquigarrow^{+} \langle \mathsf{Alice}, [\langle \mathsf{Bob}, 4\rangle, \langle \mathsf{Alice}, 3\rangle]\rangle\end{array}$$

Alice wins by default after Bob makes the illegal move "4".

Parameterised Handlers The parameter passing style used in runState is sufficiently common that it has its own specialised handling idiom known as *parameterised handlers*. A parameterised handler is a handler augmented with an explicit state parameter. Using a parameterised handler, the runState handler above may be rewritten as follows.

$$\begin{array}{l}\mathsf{runState}' : \beta \to (\langle\rangle \to \langle \alpha, \beta\rangle ! \{\mathsf{State}\ \beta, \varepsilon\}) \to (\langle\rangle \to \langle \alpha, \beta\rangle ! \{\varepsilon\}) \\ \mathsf{runState}'\ \mathit{init}\ m \stackrel{\text{def}}{=} \lambda\langle\rangle.\mathbf{handle}\ m\ \langle\rangle\ \mathbf{with} \\ \qquad\qquad \left(\begin{array}{ll} \mathit{st}.\ \mathbf{return}\ x & \mapsto \langle x, \mathit{st}\rangle \\ \quad \mathsf{Get}\ \langle\rangle\ \mathit{resume} & \mapsto \mathit{resume}\ \langle \mathit{st}, \mathit{st}\rangle \\ \quad \mathsf{Put}\ \mathit{st}'\ \mathit{resume} & \mapsto \mathit{resume}\ \langle\langle\rangle, \mathit{st}'\rangle \end{array}\right)\ \mathit{init}\end{array}$$

The binding form $st.\ \cdots$ binds the state parameter st in the handler definition. When the handler runs initially it will bind st to the value *init* by applying the parameterised handler definition to *init*. Subsequently, the parameter st is accessible in the **return** and operation cases. A key difference between ordinary handlers and parameterised handlers is that in the latter a resumption takes a pair as input, whose first component is the result of the operation invocation, and the second argument is the updated value of the parameter st. Thus parameterised handlers provide a primitive interface for the parameter-passing idiom. Indeed, using runState′ in place of runState in the previous example yields the same result.

$$(\mathsf{ps\_vs\_ps} \circ (\mathsf{runState}'\ [\,]) \circ \mathsf{history})\ (\mathsf{game}\ 7) \rightsquigarrow^{+} \langle \mathsf{Alice}, [\langle \mathsf{Alice}, 3\rangle, \langle \mathsf{Bob}, 1\rangle, \langle \mathsf{Alice}, 3\rangle]\rangle$$

The primary advantage of parameterised handlers is that they can implement parameter-passing efficiently. The runState handler above causes a large amount of closures allocations. In contrast, the parameterised handler runState′ need not allocate any closures. We discuss the operational semantics of parameterised handlers and their implementation in Section 7.

2.6 Shallow Handlers and Streaming

The machinery we have developed thus far only runs the game once. We now show how to run the game multiple times. Various forms of concurrency can be implemented using effect handlers, for example cooperative multi-threading (Bauer & Pretnar, 2015), message-passing (Hillerström, 2016), the async/await idiom (Dolan *et al.*, 2017; Leijen, 2017b), and synchronous streams via pipes (Kammar *et al.*, 2013). We will use the latter to implement a basic live scoring system that tracks the number of wins for each player. We will follow the Unix philosophy, i.e., decompose our system into a collection of modular processes, and then combine them in a pipeline.

The natural implementation of pipes is in terms of two mutually recursive handlers, which handle production and consumption of values, respectively. Using a deep handler to implement mutual recursion is at best cumbersome as it essentially amounts to encoding a mutumorphism using catamorphisms. Therefore, we switch to using *shallow* handlers to implement pipes. Unlike deep handlers, shallow handlers do not fix a particular recursion scheme. A shallow handler handles only a *single* invocation of an operation, whereas a deep handler handles *all* invocations of the same operation. Consequently shallow handlers offer more flexibility in how to handle subsequent operation invocations in a computation.

We mark shallow handlers in our calculus with a dagger (†). With shallow handlers we define a demand-driven Unix pipeline operator as follows.

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

$$\begin{array}{ll}
\mathsf{pipe}\,\langle p, c\rangle = & \mathsf{copipe}\,\langle c, p\rangle = \\
\quad \mathbf{handle}^{\dagger}\ c\,\langle\rangle\ \mathbf{with} & \quad \mathbf{handle}^{\dagger}\ p\,\langle\rangle\ \mathbf{with} \\
\quad\quad \mathbf{return}\ x \mapsto x & \quad\quad \mathbf{return}\ x \mapsto x \\
\quad\quad \mathsf{Await}\ \langle\rangle\ \mathit{resume} \mapsto \mathsf{copipe}\,\langle \mathit{resume}, p\rangle & \quad\quad \mathsf{Yield}\ s\ \mathit{resume} \mapsto \mathsf{pipe}\,\langle \mathit{resume}, \lambda\langle\rangle.c\,s\rangle
\end{array}$$

A pipe takes two suspended computations, a producer p and a consumer c. Each of the computations 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 pipe runs the consumer first. If the consumer terminates with a value, then the **return** clause is executed and returns that value as is. If the consumer performs the Await operation, then the copipe handler is invoked with the resumption of the consumer (*resume*) and the producer (p) as arguments.

The copipe function runs the producer to get a value to feed to the waiting consumer. If the producer performs the Yield operation, then pipe is invoked with the resumption of the producer along with a thunk that applies the consumer's resumption to the yielded value.

Example To illustrate the interaction between pipe and copipe, we will consider a concrete example with a simple producer, ones, which produces an infinite stream of ones, and an equally simple consumer, add_2, which awaits two integers and returns their sum.

$$\mathsf{ones} \stackrel{\text{def}}{=} \mathbf{rec}\ \mathit{ones}\ \langle\rangle.\mathbf{do}\ \mathsf{Yield}\ 1; \mathit{ones}\ \langle\rangle \qquad \mathsf{add}_2 \stackrel{\text{def}}{=} \lambda\langle\rangle.\mathbf{do}\ \mathsf{Await}\ \langle\rangle + \mathbf{do}\ \mathsf{Await}\ \langle\rangle$$

Let H_{pipe} and H_{copipe} denote the handler definitions of pipe and copipe respectively. We detail the reduction sequence of pipe $\langle \mathsf{ones}, \mathsf{add}_2 \rangle$.

$$
\begin{array}{ll}
 & \mathsf{pipe}\,\langle \mathsf{ones}, \mathsf{add}_2 \rangle \\
\rightsquigarrow & \quad \text{(definition of pipe)} \\
 & \mathbf{handle}^{\dagger}\ \mathsf{add}_2\ \langle\rangle\ \mathbf{with}\ H_{\mathsf{pipe}} \\
\rightsquigarrow^{+} & \quad \text{(definition of } \mathsf{add}_2 \text{ with } \mathscr{E}_{\mathsf{add}_2} = [\,] + \mathbf{do}\ \mathsf{Await}) \\
 & \mathbf{handle}^{\dagger}\ \mathscr{E}_{\mathsf{add}_2}[\mathbf{do}\ \mathsf{Await}\ \langle\rangle]\ \mathbf{with}\ H_{\mathsf{pipe}} \\
\rightsquigarrow & \quad \text{(Await clause definition)} \\
 & \mathsf{copipe}\langle \mathit{resume}, \mathsf{ones} \rangle[\lambda y.\mathscr{E}_{\mathsf{add}_2}[y]/\mathit{resume}] \\
= & \quad \text{(substitution)} \\
 & \mathsf{copipe}\langle (\lambda y.\mathscr{E}_{\mathsf{add}_2}[y]), \mathsf{ones} \rangle
\end{array}
$$

The resumption is simply the continuation $\mathscr{E}_{\mathsf{add}_2}$. In contrast to a deep handler, the shallow handler H_{pipe} is discarded after handling its operation. Evaluation continues with the application of copipe.

$$
\begin{array}{ll}
\rightsquigarrow & \quad \text{(definition of copipe)} \\
 & \mathbf{handle}^{\dagger}\ \mathit{ones}\ \langle\rangle\ \mathbf{with}\ H_{\mathsf{copipe}} \\
\rightsquigarrow^{+} & \quad \text{(definition of } \mathit{ones} \text{ with } \mathscr{E}_{\mathsf{ones}} = [\,]; \mathit{ones}\ \langle\rangle) \\
 & \mathbf{handle}^{\dagger}\ \mathscr{E}_{\mathsf{ones}}[\mathbf{do}\ \mathsf{Yield}\ 1]\ \mathbf{with}\ H_{\mathsf{copipe}} \\
\rightsquigarrow & \quad \text{(Await clause definition)} \\
 & \mathsf{pipe}\ \langle \mathit{resume}, (\lambda\ \langle\rangle.(\lambda y.\mathscr{E}_{\mathsf{add}_2}[y])\ s) \rangle[\lambda z.\mathscr{E}_{\mathsf{ones}}[z]/\mathit{resume}, 1/s] \\
\rightsquigarrow & \quad \text{(definition of pipe)} \\
 & \mathbf{handle}^{\dagger}\ (\lambda\ \langle\rangle.(\lambda y.\mathscr{E}_{\mathsf{add}_2}[y])\ 1)\ \langle\rangle\ \mathbf{with}\ H_{\mathsf{pipe}} \\
\rightsquigarrow^{+} & \quad \text{(beta reduction)} \\
 & \mathbf{handle}^{\dagger}\ \mathscr{E}_{\mathsf{add}_2}[1]\ \mathbf{with}\ H_{\mathsf{pipe}} \\
\rightsquigarrow & \quad \text{(definition of } \mathscr{E}_{\mathsf{add}_2} \text{ with } \mathscr{E}'_{\mathsf{add}_2} = 1 + [\,]) \\
 & \mathbf{handle}^{\dagger}\ \mathscr{E}'_{\mathsf{add}_2}[\mathbf{do}\ \mathsf{Await}\ \langle\rangle]\ \mathbf{with}\ H_{\mathsf{pipe}} \\
\rightsquigarrow & \quad \text{(Await clause definition)} \\
 & \mathsf{copipe}\langle (\lambda y.\mathscr{E}'_{\mathsf{add}_2}[y]), (\lambda z.\mathscr{E}_{\mathsf{ones}}[z]) \rangle \\
\rightsquigarrow & \quad \text{(definition of copipe)} \\
 & \mathbf{handle}^{\dagger}\ (\lambda z.\mathscr{E}_{\mathsf{ones}}[z])\ \langle\rangle\ \mathbf{with}\ H_{\mathsf{copipe}} \\
\rightsquigarrow^{+} & \quad \text{(the continuation of } \mathscr{E}_{\mathsf{ones}} \text{ eventually performs Yield 1)} \\
 & \mathbf{handle}^{\dagger}\ \mathscr{E}_{\mathsf{ones}}[\mathbf{do}\ \mathsf{Yield}\ 1]\ \mathbf{with}\ H_{\mathsf{copipe}} \\
\rightsquigarrow & \quad \text{(Yield clause definition)} \\
 & \mathsf{pipe}\langle (\lambda z.\mathscr{E}_{\mathsf{ones}}[z]), (\lambda\langle\rangle.(\lambda y.\mathscr{E}'_{\mathsf{add}_2}[y])\ 1) \rangle \\
\rightsquigarrow^{+} & \quad \text{(beta reduction, } \mathscr{E}'_{\mathsf{add}_2}[1] = 1 + 1) \\
 & \mathbf{handle}^{\dagger}\ 2\ \mathbf{with}\ H_{\mathsf{pipe}} \\
\rightsquigarrow & \quad (\mathbf{return} \text{ clause definition}) \\
 & 2
\end{array}
$$

Observe how the handling of operations alternates between pipe and copipe throughout the reduction sequence. Either handler handles exactly one operation before transferring control to the other. The crucial operational difference between shallow and deep handlers is that the former does not recursively wrap itself around resumptions. Effectively, shallow handlers provide the ability to perform case-splits on computations.

We return to the Nim game. We have to implement some processes to pipe together. First, we implement a process that receives an integer, which is used as the starting value for a game. As output, the process will yield the winner of the game.

$$\begin{aligned}&\mathsf{gameProc} : \langle\rangle \to \alpha!\{\mathsf{Await} : \langle\rangle \to \mathsf{Int}, \mathsf{Yield} : \mathsf{Player} \to \langle\rangle, \varepsilon\}\\&\mathsf{gameProc} \stackrel{\text{def}}{=} \lambda\langle\rangle.\ \mathbf{let}\ n \leftarrow \mathbf{do}\ \mathsf{Await}\ \langle\rangle\ \mathbf{in}\\&\qquad\qquad \mathbf{let}\ \mathit{winner} \leftarrow (\mathsf{ps\_vs\_ps} \circ \mathsf{defaultVictory} \circ \mathsf{monitor})\ (\mathit{game}\ n)\ \mathbf{in}\\&\qquad\qquad \mathbf{do}\ \mathsf{Yield}\ \mathit{winner};\ \mathsf{gameProc}\ \langle\rangle\end{aligned}$$

From the signature of gameProc we can see that the process will fit as an intermediate component in a pipeline, since it both Awaits and Yields values. The process first awaits the starting value n, which is used to start a new game with the cheat detection enabled and where Alice and Bob both adopt the perfect strategy. The winner of the game is yielded, before awaiting a new start value.

To obtain some start values for successive games, we define a recursive process which produces a monotonically increasing sequence of integers on demand.

$$\begin{aligned}&\mathsf{startFrom} : \mathsf{Int} \to \langle\rangle \to \alpha!\{\mathsf{Yield} : \mathsf{Int} \to \langle\rangle, \varepsilon\}\\&\mathsf{startFrom}\ j \stackrel{\text{def}}{=} \lambda\langle\rangle.\mathbf{do}\ \mathsf{Yield}\ j;\ \mathsf{startFrom}\ (j+1)\ \langle\rangle\end{aligned}$$

Given an initial value j the startFrom produces the infinite sequence:

$$j, j+1, j+2, j+3, \ldots$$

From the signature we can tell that this process acts exclusively as a producer.

Next, we implement a stateful process that keeps track of the number of games won by each player, and upon receiving the winner of a game displays the current score. We represent the current score as a simple pair $\mathsf{Score} \stackrel{\text{def}}{=} \langle \mathsf{Int}, \mathsf{Int}\rangle$, where the first component is the number of times Alice has won, and vice versa, the second component contains the number of times Bob has won.

$$\begin{aligned}&\mathsf{livescore} : \langle\rangle \to \alpha!\{\mathsf{Await} : \langle\rangle \to \mathsf{Player}, \mathsf{Get} : \langle\rangle \to \mathsf{Score}, \mathsf{Put} : \mathsf{Score} \to \langle\rangle, \varepsilon\}\\&\mathsf{livescore} \stackrel{\text{def}}{=} \lambda\langle\rangle.\ \mathbf{let}\ \langle \mathit{fst} = \mathit{aliceSc}, \mathit{snd} = \mathit{bobSc}\rangle =\\&\qquad\qquad \mathbf{case}\ \mathbf{do}\ \mathsf{Await}\ \langle\rangle\\&\qquad\qquad \left\{\begin{array}{l}\mathsf{Alice} \mapsto \langle \mathbf{do}\ \mathsf{Get}\ \langle\rangle\ \mathbf{with}\ \mathit{fst} = (\mathbf{do}\ \mathsf{Get}\ \langle\rangle).\mathit{fst} + 1\rangle\\ \mathsf{Bob}\ \ \mapsto \langle \mathbf{do}\ \mathsf{Get}\ \langle\rangle\ \mathbf{with}\ \mathit{snd} = (\mathbf{do}\ \mathsf{Get}\ \langle\rangle).\mathit{snd} + 1\rangle\end{array}\right\}\\&\qquad\qquad \mathbf{in}\ \mathbf{do}\ \mathsf{Put}\ \langle \mathit{aliceSc}, \mathit{bobSc}\rangle;\ \mathsf{display}\ \mathit{aliceSc}\ \mathit{bobSc};\ \mathsf{livescore}\ \langle\rangle\end{aligned}$$

The signature of livescore tells us not only that this process is a consumer, but also that it is stateful. First the process updates the current score by pattern matching on the winner, which is received via the Await operation. The current score is updated using functional record update. Subsequently, the value of the state cell is updated with the new score. It is worth noting that an alternative to using the abstract state operation Get and Put would be to make livescore a parameterised handler, thereby internalising the state. The advantage of internal state is that everything is kept local at the expense of modularity as its state handling would not be overloadable.

The function display prints the current score using a primitive operation, e.g. writing to standard out. With the processes we have defined so far, we can produce a live score board for an infinite series of games. In order to make the overall process finite, we define another

intermediate process, that halts the pipeline after a given number of steps:

$$\begin{aligned}&\mathsf{take} : \mathsf{Int} \to \langle\rangle \to \langle\rangle!\{\mathsf{Await} : \langle\rangle \to \beta, \mathsf{Yield} : \beta \to \langle\rangle, \varepsilon\}\\&\mathsf{take}\, j \stackrel{\text{def}}{=} \lambda\langle\rangle.\mathbf{if}\ j \leq 0\ \mathbf{then}\ \langle\rangle\ \mathbf{else\ do}\ \mathsf{Yield}\ (\mathbf{do}\ \mathsf{Await}\ \langle\rangle);\ \mathsf{take}\ (j-1)\ \langle\rangle\end{aligned}$$

This process uses explicit state passing to keep track of whether to halt. Alternatively, we could have used the state interface provided by Get and Put.

For convenience, we define an infix pipe operator for constructing pipelines, analogous to the Unix shell's pipe operator `p1 | p2`.

$$\begin{aligned}&\gg : (\langle\rangle \to \alpha!\{\mathsf{Yield} : \beta \to \langle\rangle, \varepsilon\}) \to (\langle\rangle \to \alpha!\{\mathsf{Await} : \langle\rangle \to \beta, \varepsilon\}) \to (\langle\rangle \to \alpha!\{\varepsilon\})\\&p \gg c \stackrel{\text{def}}{=} \lambda\langle\rangle.\mathsf{pipe}\ \langle p, c\rangle\end{aligned}$$

Plugging everything together, we report the live score during a series of consecutive games.

$$\begin{aligned}&(\mathsf{startFrom}\ 7 \gg \mathsf{gameProc} \gg \mathsf{take}\ 3 \gg \mathsf{ignore} \circ (\mathsf{runState}\ \langle 0,0\rangle\ \mathsf{livescore}))\ \langle\rangle\\&\quad\texttt{Alice 1 - 0 Bob}\\&\quad\texttt{Alice 1 - 1 Bob}\\&\quad\texttt{Alice 2 - 1 Bob}\\&\rightsquigarrow^{+} \langle\rangle\end{aligned}$$

We start an infinite series of games with increasing initial heap sizes, then we consume three of those games, and for each of the three games we report the winner. We post-compose $\mathsf{ignore} \stackrel{\text{def}}{=} \lambda x.\langle\rangle$ to disregard the return value of runState.

Implementing Pipes with Deep Handlers As we have seen, shallow handlers provide a direct way to implement Unix-style pipes. It is also possible to implement pipes using deep handlers, albeit in a much more roundabout way. Indeed, shallow handlers can always be simulated by deep handlers, as we show in Section 4, though not in a straightforward way. With deep handlers we cannot use term level recursion and choose how to handle the next step of the computation, instead we follow Kammar *et al.* (2013) and simulate a form of open recursion by using recursive types to parameterise a pair of mutually recursive functions, namely, the producer and consumer.

$$\begin{aligned}\mathsf{Producer}\ \varepsilon\ \alpha\ \beta &\stackrel{\text{def}}{=} \langle\rangle \to (\mathsf{Consumer}\ \varepsilon\ \alpha\ \beta \to \alpha!\{\varepsilon\})!\{\varepsilon\}\\\mathsf{Consumer}\ \varepsilon\ \alpha\ \beta &\stackrel{\text{def}}{=} \beta \to (\mathsf{Producer}\ \varepsilon\ \alpha\ \beta \to \alpha!\{\varepsilon\})!\{\varepsilon\}\end{aligned}$$

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. Both are parameterised by an effect variable ε, that denotes the allowed effects. Using these recursive types, we can define the pipe′ and copipe′ implementations

using deep handlers as follows:

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

$$\begin{aligned}&\mathsf{pipe}'\,c \stackrel{\text{def}}{=}\\ &\quad\textbf{handle}\ c\,\langle\rangle\ \textbf{with}\\ &\quad\quad\textbf{return}\ x \mapsto \lambda y.x\\ &\quad\quad\mathsf{Await}\ \langle\rangle\ \mathit{resume} \mapsto \lambda p.p\ \langle\rangle\ \mathit{resume}\end{aligned}\qquad\begin{aligned}&\mathsf{copipe}'\,p \stackrel{\text{def}}{=}\\ &\quad\textbf{handle}\ p\,\langle\rangle\ \textbf{with}\\ &\quad\quad\textbf{return}\ x \mapsto \lambda y.x\\ &\quad\quad\mathsf{Yield}\ s\ \mathit{resume} \mapsto \lambda c.c\ s\ \mathit{resume}\end{aligned}$$

$$p \gg' c \stackrel{\text{def}}{=} \lambda\langle\rangle.\mathsf{pipe}'\ c\ (\lambda\langle\rangle.\mathsf{copipe}'\ p)$$

Application of the pipe operator is no longer direct as extra plumbing is required to connect the handlers. The observable behaviour of $\gg'$ is the same as $\gg$. Indeed, the example yields the same result.

$$\begin{aligned}&(\mathsf{startFrom}\ 7 \gg' \mathsf{gameProc} \gg' \mathsf{take}\ 3 \gg' \mathsf{ignore} \circ (\mathsf{runState}\ \langle 0,0\rangle\ \mathsf{livescore}))\ \langle\rangle\\ &\quad\texttt{Alice 1 - 0 Bob}\\ &\quad\texttt{Alice 1 - 1 Bob}\\ &\quad\texttt{Alice 2 - 1 Bob}\\ &\rightsquigarrow^{+} \langle\rangle\end{aligned}$$

This example shows that, while it is possible use deep handlers for everything in our application, it is not always convenient to do so. The extra flexibility of shallow handlers make it possible to implement arbitrary recursion schemes in direct-style, while deep handlers require one to jump through hoops. Deep handlers may afford more efficient implementations than shallow handlers, as well-studied standard fusion techniques for catamorphisms (Meijer *et al.*, 1991) can be applied to deep handlers (Wu *et al.*, 2014). Thus a case can be made for providing both constructs in a single language. In the next section we introduce a calculus which includes both deep and shallow handlers. In Section 4 we study the relationship between deep and shallow handlers further.

3 Handler Calculus

In this section, we present $\lambda^{\dagger}$, a Church-style row-polymorphic call-by-value calculus with effect handlers (Hillerström & Lindley, 2018). The calculus captures the essence of the effect handlers as realised by the intermediate representation of Links. As in Links, our calculus includes both deep and shallow handlers, allowing us to compare both within the same language. Links also features parameterised handles. We will further extend the calculus with parameterised handlers in Section 7. A key ingredient of our calculus is row polymorphism. As in Links, row polymorphism is used to uniformly support extensible records and variants as well as an extensible effect system. We use Remy-style row polymorphism with presence polymorphism (Remy, 1993). The term syntax of our calculus also follows the syntax of the Links intermediate representation closely, as it is based on fine-grain call-by-value (Levy *et al.*, 2003). Fine-grain call-by-value is similar to A-normal form (Flanagan *et al.*, 1993) in that it names each intermediate computation, but unlike A-normal form is closed under β-reduction. Fine-grain call-by-value provides a convenient framework for working with delimited control as only the operational rules for let bindings

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

Fig. 2. Types, Kinds, and Environments

and handlers admit continuations. By contrast, in standard call-by-value every operational congruence rule admits a continuation.

3.1 Syntax of Types and Kinds, Kinding Rules

The syntax of types, kinds, and environments is given in Fig. 2.

Value Types Function types $A \to C$ classify functions that map values of type A to computations of type C. Polymorphic types $\forall \alpha^K.C$ quantify universally over a type variable α of kind K. Record types $\langle R \rangle$ represent records with fields constrained by the row R. Dually, variant types $[R]$ represents tagged sums constrained by the 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. Closed rows are further considered equivalent up to inclusion of explicitly absent labels. 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.

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

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

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

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

Fig. 3. Kinding Rules for $\lambda^{\dagger}$

Kinds We have six kinds: Type, Comp, Effect, Handler, $\mathsf{Row}_{\mathscr{L}}$, Presence, which respectively classify value types, computation types, effect types, handler types, row types, and presence types. Rows have the property that they mention each label at most once. To ensure this property, we annotate row kinds with a set of labels $\mathscr{L}$, which contains the labels mentioned by the row. The rule ExtendRow builds the set using disjoint union to ensure uniqueness. In other words, the kind $\mathsf{Row}_\emptyset$ denotes a complete row, whilst $\mathsf{Row}_{\mathscr{L}}$ for nonempty $\mathscr{L}$ denotes a partial row that may not mention the labels in $\mathscr{L}$. We write $\ell : A$ as sugar for $\ell : \mathsf{Pre}(A)$.

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.

Kinding Rules The kinding judgement $\Delta \vdash T : K$ states that type T has kind K in kind environment Δ. The kinding rules for $\lambda^{\dagger}$ are given in Fig. 3.

3.2 Terms

The terms are given in Fig. 4. 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

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

Fig. 4. Term Syntax

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 with 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 $\{\ell\, p\, r \mapsto N_\ell\}_{\ell \in \mathscr{L}}$. 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, whilst H^{ℓ} yields the set of either zero or the unique operation clause 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 sequential composition, pattern matching, n-ary record extension, n-ary case elimination, and n-ary tuples.

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

Values

$$\text{T-LAM}\ \frac{\Delta;\Gamma,x:A\vdash M:C}{\Delta;\Gamma\vdash \lambda x^{A}.M:A\to C} \qquad \text{T-REC}\ \frac{\Delta;\Gamma,f:A\to C,x:A\vdash M:C}{\Delta;\Gamma\vdash \mathbf{rec}\, f^{A\to C}\, x.M:A\to C} \qquad \text{T-POLYLAM}\ \frac{\Delta,\alpha:K;\Gamma\vdash M:C \qquad \alpha\notin FTV(\Gamma)}{\Delta;\Gamma\vdash \Lambda\alpha^{K}.M:\forall\alpha^{K}.C}$$

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

$$\text{T-VAR}\ \frac{x:A\in\Gamma}{\Delta;\Gamma\vdash x:A}$$

Computations

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

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

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

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

Handlers

$$\text{T-HANDLER}\ \frac{\begin{array}{c}C=A!\{(\ell_i:A_i\to B_i)_i;R\}\\ D=B!\{(\ell_i:P)_i;R\}\\ H=\{\mathbf{return}\,x\mapsto M\}\uplus\{\ell_i\,p_i\,r_i\mapsto N_i\}_i\\ \Delta;\Gamma,x:A\vdash M:D\\ [\Delta;\Gamma,p_i:A_i,r_i:B_i\to D\vdash N_i:D]_i\end{array}}{\Delta;\Gamma\vdash H:C\Rightarrow D} \qquad \text{T-HANDLER}^{\dagger}\ \frac{\begin{array}{c}C=A!\{(\ell_i:A_i\to B_i)_i;R\}\\ D=B!\{(\ell_i:P)_i;R\}\\ H=\{\mathbf{return}\,x\mapsto M\}\uplus\{\ell_i\,p_i\,r_i\mapsto N\}_i\\ \Delta;\Gamma,x:A\vdash M:D\\ [\Delta;\Gamma,p_i:A_i,r:B_i\to C\vdash N:D]_i\end{array}}{\Delta;\Gamma\vdash H:C\Rightarrow^{\dagger}D}$$

Fig. 5. Typing Rules for $\lambda^{\dagger}$

3.3 Typing Rules

The typing rules are given in Fig. 5. 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

$$
\begin{array}{lll}
\text{S-App} & (\lambda x.M)\,V \rightsquigarrow M[V/x] & \\
\text{S-TyApp} & (\Lambda\alpha.M)\,T \rightsquigarrow M[T/\alpha] & \\
\text{S-Split} & \mathbf{let}\ \langle \ell = x; y\rangle = \langle \ell = V; W\rangle\ \mathbf{in}\ N \rightsquigarrow N[V/x, W/y] & \\
\text{S-Case}_1 & \mathbf{case}\ \ell\, V\{\ell\, x \mapsto M; y \mapsto N\} \rightsquigarrow M[V/x] & \\
\text{S-Case}_2 & \mathbf{case}\ \ell\, V\{\ell'\, x \mapsto M; y \mapsto N\} \rightsquigarrow N[\ell\, V/y], & \text{if } \ell \neq \ell' \\
\text{S-Rec} & (\mathbf{rec}\, g\, x.M)\,V \rightsquigarrow M[(\mathbf{rec}\, g\, x.M)/g, V/x] & \\
\text{S-Let} & \mathbf{let}\ x \leftarrow \mathbf{return}\ V\ \mathbf{in}\ N \rightsquigarrow N[V/x] & \\
\text{S-Ret} & \mathbf{handle}^{\delta}\ (\mathbf{return}\ V)\ \mathbf{with}\ H \rightsquigarrow N[V/x], & \text{where } H^{\mathrm{ret}} = \{\mathbf{return}\ x \mapsto N\} \\
\text{S-Op} & \mathbf{handle}\ \mathscr{E}[\mathbf{do}\ \ell\ V]\ \mathbf{with}\ H \rightsquigarrow N[V/p, \lambda y.\mathbf{handle}\ \mathscr{E}[\mathbf{return}\ y]\ \mathbf{with}\ H/\mathit{resume}], & \\
 & & \text{where } \ell \notin BL(\mathscr{E}) \text{ and } H^{\ell} = \{\ell\, p\, r \mapsto N\} \\
\text{S-Op}^{\dagger} & \mathbf{handle}^{\dagger}\ \mathscr{E}[\mathbf{do}\ \ell\ V]\ \mathbf{with}\ H \rightsquigarrow N[V/p, \lambda y.\mathscr{E}[\mathbf{return}\ y]/\mathit{resume}], & \\
 & & \text{where } \ell \notin BL(\mathscr{E}) \text{ and } H^{\ell} = \{\ell\, p\, r \mapsto N\} \\
\text{S-Lift} & \mathscr{E}[M] \rightsquigarrow \mathscr{E}[N], & \text{if } M \rightsquigarrow N
\end{array}
$$

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

Fig. 6. Small-Step Operational Semantics

$\Delta;\Gamma \vdash H : C \Rightarrow^{\delta} D$ 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. The set $FTV(\Gamma)$ denotes the free type variables of Γ.

The interesting rules are those for performing and handling operations. The T-Handler and T-Handler† rules are where most of the work happens. The only difference between the two rules is the typing of resumptions. A deep resumption has the same return type, D, as its handler, whilst a shallow resumption has the same return type, C, as the input computation. The effect rows on the input computation type C and the output computation type D must mention every operation in the domain of the handler. In the output row those operations may be either present ($\mathsf{Pre}(A)$), absent (Abs), or polymorphic in their presence (θ), whilst in the input row they must be mentioned with a present type as those types are used to type operation clauses. The effect rows must also share the same suffix R, which 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. To exemplify all of this consider the handler definition in the function ps_vs_ps from Section 2.2. The domain of the handler definition is the singleton set $\{\mathsf{Move}\}$ and its $\lambda^{\dagger}$-type is:

$$\alpha!\{\mathsf{Move} : \mathsf{Pre}(\langle \mathsf{Player}, \mathsf{Int}\rangle \to \mathsf{Int}), \varepsilon\} \Rightarrow \alpha!\{\mathsf{Move} : \theta, \varepsilon\}$$

The operation Move is mentioned in the input row with a present type, because it appears in the domain of the handler definition. In the output row it is mentioned with a presence polymorphic type, allowing Move to be reintroduced again later. Alternatively, we could make it absent (Abs) to prevent it from being reintroduced. The two effect rows share the same suffix ε, meaning that any unmentioned operation is forwarded by the handler.

3.4 Operational Semantics

Figure 6 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. We write $BL(\mathscr{E})$ for the set of operation labels bound by $\mathscr{E}$.

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

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 $\ell \notin BL(\mathscr{E})$ 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 in the former the resumption reinserts the handler around the captured evaluation context.

Remark The S-LET rule eliminates a trivial computation term **return** V. It is not the only rule which does so. The rule S-HANDLE-RET also eliminates a **return** computation. In fact, we could omit let bindings altogether since any let binding can be thought of as syntactic sugar for a trivial handler

$$\mathbf{let}\ x \leftarrow M\ \mathbf{in}\ N \equiv \mathbf{handle}\ M\ \mathbf{with}\ \{\mathbf{return}\ x \rightarrow N\}$$

Nevertheless, we elect to keep let bindings as a distinguished syntactic form in our formalism as implementing let bindings using handlers is somewhat heavy weight. One can also view the computation **return** V as an operation invocation $\mathbf{return}_A\, V$, given by a type-indexed family of operations $\mathbf{return}_A : A \rightarrow \mathsf{Zero}$, that is exceptions which are handled uniformly by the **return** clause of the nearest enclosing handler.

For any relation R, we write R^{+} for the transitive closure of R, and R^{*} for the reflexive-transitive closure of R.

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

Theorem 1 (*Type Soundness*)
If $\vdash M : A!E$ then either M diverges or there exists $\vdash N : A!E$ such that $M \rightsquigarrow^{+} N$ such that N is normal with respect to E.

4 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 reductions. The latter construction generalises the example of pipes implemented using deep handlers that we gave in Section 2.6.

4.1 Deep as Shallow

The implementation of deep handlers using shallow handlers (and recursive functions) is by a direct local translation, similar to how one would implement a fold (catamorphism)

in terms of general recursion. 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 $\mathscr{S}[\![-]\!]$ is defined as the homomorphic extension of the following equations to all terms.

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

Example We illustrate the translation $\mathscr{S}[\![-]\!]$ on the ps_vs_ps handler from Section 2.2 (recall that the variable m is bound to the input computation). The example here is reproduced in ANF notation.

$$\mathscr{S}\left[\!\!\left[\begin{array}{ll}\mathbf{handle}\ m\ \langle\rangle\ \mathbf{with} & \\ \quad \mathbf{return}\ x & \mapsto \mathbf{return}\ x \\ \quad \mathsf{Move}\ \langle\_,n\rangle\ resume & \mapsto \mathbf{let}\ y \leftarrow ps\ n\ \mathbf{in}\ resume\ y\end{array}\right]\!\!\right] =$$

$$\left(\begin{array}{lll}\mathbf{rec}\ h\,f.\ \mathbf{handle}^{\dagger}\ f\ \langle\rangle\ \mathbf{with} & & \\ \quad \mathbf{return}\ x & \mapsto \mathbf{return}\ x \\ \quad \mathsf{Move}\ \langle\_,n\rangle\ resume & \mapsto & \\ \quad\quad \mathbf{let}\ r \leftarrow \mathbf{return}\ \lambda x.h(\lambda\langle\rangle.resume\ x)\ \mathbf{in} & & \\ \quad\quad \mathbf{let}\ y \leftarrow ps\ n\ \mathbf{in}\ r\ y & &\end{array}\right)(\lambda\langle\rangle.m\ \langle\rangle)$$

Theorem 2
If $\Delta;\Gamma \vdash M : C$ then $\Delta;\Gamma \vdash \mathscr{S}[\![M]\!] : C$.

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

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

4.2 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 $\mathscr{D}[\![-]\!]$ is

defined as the homomorphic extension of the following equations to all types, terms, and type environments.

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

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.

Example We demonstrate the translation $\mathscr{D}[\![-]\!]$ on the pipe handler from Section 2.6 (recall that the variables c and p are bound to the consumer and producer functions respectively). The example is reproduced in ANF notation.

$$\begin{aligned}
&\mathscr{D}\left[\!\!\left[\begin{array}{ll} \mathbf{handle}^\dagger\ c\,\langle\rangle\ \mathbf{with} & \\ \quad \mathbf{return}\ x & \mapsto \mathbf{return}\ x \\ \quad \mathsf{Await}\ \langle\rangle\ \mathit{resume} & \mapsto \mathsf{copipe}\ \langle \mathit{resume}, p\rangle \end{array}\right]\!\!\right] = \\
&\mathbf{let}\ z \leftarrow\ \mathbf{handle}\ c\ \langle\rangle\ \mathbf{with} \\
&\qquad \mathbf{return}\ x \mapsto \mathbf{return}\ \langle \lambda\langle\rangle.\mathbf{return}\ x, \lambda\langle\rangle.\mathbf{return}\ x\rangle \\
&\qquad \mathsf{Await}\ \langle\rangle\ \mathit{resume} \mapsto \\
&\qquad\quad \mathbf{let}\ r \leftarrow \lambda x.\mathbf{let}\ z \leftarrow \mathit{resume}\ x\ \mathbf{in\ let}\ \langle f, g\rangle = z\ \mathbf{in}\ f\ \langle\rangle\ \mathbf{in} \\
&\qquad\quad \mathbf{return}\ \langle \lambda\langle\rangle.\mathbf{let}\ x \leftarrow \mathbf{do}\ \ell\ p\ \mathbf{in}\ r\,x, \lambda\langle\rangle.\mathscr{D}[\![\mathsf{copipe}]\!]\langle r, p\rangle\rangle \\
&\mathbf{in\ let}\ \langle f, g\rangle = z\ \mathbf{in}\ g\ \langle\rangle
\end{aligned}$$

Theorem 4
If $\Delta; \Gamma \vdash M : C$ then $\mathscr{D}[\![\Delta]\!]; \mathscr{D}[\![\Gamma]\!] \vdash \mathscr{D}[\![M]\!] : \mathscr{D}[\![C]\!]$.

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 2 (*Administrative Evaluation Contexts*)
An evaluation context $\mathscr{E}$ is administrative, $\mathit{admin}(\mathscr{E})$, iff

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

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

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

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

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 1
For all shallow handlers H, the following context is administrative:

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

Theorem 5 (*Simulation up to Administrative Reduction*)
If $M' \gtrsim \mathscr{D}[\![M]\!]$ and $M \rightsquigarrow N$ then there exists N' such that $N' \gtrsim \mathscr{D}[\![N]\!]$ and $M' \rightsquigarrow^{+} N'$.

Proof
By induction on $\rightsquigarrow$ using a substitution lemma and Lemma 1. The interesting case is S-Op†, which uses Lemma 1 to approximate the body of the resumption up to administrative reduction. □

5 Continuation Passing Style for Effect Handlers

We now show how our effect handler calculus $\lambda^{\dagger}$ can be implemented via a continuation passing style (CPS) translation into a calculus without effect handlers. Beyond a practical implementation technique for $\lambda^{\dagger}$, the contribution of this section is the identification of the structure of generalised continuations that we need to correctly model the behaviour of dynamically nested handlers, and the way that deep and shallow handlers behave differently upon resumption. Once we have identified this structure in the CPS translation, we use it to design an CEK-like abstract machine for deep and shallow handlers in Section 6.

The basic idea of the translation is as follows. We upgrade the continuation argument from a standard CPS translation to be a stack of continuations, similar to CPS translations for delimited continuations (Materzok & Biernacki, 2012). Special to handlers, this stack is composed of alternating pure and handler continuations. Returning from a computation invokes the pure continuation, and invoking an operation initiates a search through the handler frames to find the one that handles it. The frames skipped over in the search are remembered so they can be reinstated after the operation has been handled and execution resumed.

We define our untyped target calculus in Section 5.1. We then present our CPS translation in stages. We start with a basic translation for fine-grain call-by-value without handlers in Section 5.2. We then formulate a sequence of first-order translations that progressively move from representing the dynamic stack of handlers as functions to explicit stacks in Section 5.3, gaining support for both deep and shallow handlers as we do so. These steps prepare us for our final higher-order one-pass translation (Danvy & Filinski, 1992) in

Syntax

Values	$V, W ::= x \mid \lambda x.M \mid \mathbf{rec}\, g\, x.M \mid \langle\rangle \mid \langle V, W\rangle \mid \ell$
Computations	$M, N ::= V \mid M\, W \mid \mathbf{let}\, \langle x, y\rangle = V\, \mathbf{in}\, N \mid \mathbf{case}\, V\, \{\ell \mapsto M; y \mapsto N\} \mid \mathbf{absurd}\, V$
Evaluation contexts	$\mathscr{E} ::= [\,] \mid \mathscr{E}\, W$

Reductions

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

Syntactic sugar

$$
\begin{array}{lll}
\mathbf{let}\, x = V\, \mathbf{in}\, N \equiv N[V/x] & \langle\rangle \equiv \ell_{\langle\rangle} & [] \equiv \ell_{[]} \\
\ell\, V \equiv \langle \ell; V\rangle & \langle \ell = V; W\rangle \equiv \langle \ell, \langle V, W\rangle\rangle & V :: W \equiv \langle \ell_{::}, \langle V, W\rangle\rangle \\
\mathbf{case}\, V\, \{\ell\, x \mapsto M; y \mapsto N\} \equiv & \mathbf{let}\, \langle \ell = x; y\rangle = V\, \mathbf{in}\, N \equiv & \\
\quad \mathbf{let}\, y = V\, \mathbf{in}\, \mathbf{let}\, \langle z, x\rangle = y\, \mathbf{in} & \quad \mathbf{let}\, \langle z, z'\rangle = V\, \mathbf{in}\, \mathbf{let}\, \langle x, y\rangle = z'\, \mathbf{in} & \\
\quad \mathbf{case}\, z\, \{\ell \mapsto M; z' \mapsto N\} & \quad \mathbf{case}\, z\, \{\ell \mapsto N; z'' \mapsto \ell_{\perp}\} &
\end{array}
$$

Fig. 7. Untyped Target Calculus for the CPS Translations

Section 5.4 that uses (higher-order) static computation at translation time to avoid run time administrative reductions.

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 ($\lambda x.M$); recursive functions ($\mathbf{rec}\, g\, x.M$); empty tuples ($\langle\rangle$); pairs ($\langle V, W\rangle$); and first-class labels (ℓ). In Section 5.3.3, we will extend the values to also include convenience constructors for building resumptions and invoking structured continuations. Computations (M) comprise: values (V); applications ($M\ V$); pair elimination ($\mathbf{let}\, \langle x, y\rangle = V\, \mathbf{in}\, N$); label elimination ($\mathbf{case}\, V\, \{\ell \mapsto M; x \mapsto N\}$); and explicit marking of unreachable code (**absurd**). 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 relaxation is used in our initial CPS translations, but will be ruled out in our final translation when we start to use explicit lists to represent stacks of handlers in Section 5.3.2.

The reductions for functions, pairs, and first-class labels are standard.

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. For desugaring records, we assume a failure constant $\ell_{\perp}$ (e.g. a divergent term) to cope with the case of pattern matching failure.

Values

$$\begin{aligned}
[\![x]\!] &= x \\
[\![\lambda x.M]\!] &= \lambda x.[\![M]\!] \\
[\![\Lambda \alpha.M]\!] &= \lambda k.[\![M]\!]\ k \\
[\![\mathbf{rec}\, g\, x.M]\!] &= \mathbf{rec}\, g\, x.[\![M]\!] \\
[\![\langle\rangle]\!] &= \langle\rangle \\
[\![\langle \ell = V; W\rangle]\!] &= \langle \ell = [\![V]\!]; [\![W]\!]\rangle \\
[\![\ell\ V]\!] &= \ell\ [\![V]\!]
\end{aligned}$$

Computations

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

Fig. 8. First-Order CPS Translation of Fine-Grain Call-By-Value

5.2 CPS Translation for Fine-Grain Call-By-Value

We start by giving a CPS translation of the operation and handler-free subset of $\lambda^{\dagger}$ in Figure 8. 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 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 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.

5.3 First-Order CPS Translations of Handlers

As is usual for CPS, the translation of a computation term by the basic CPS translation in Section 5.2 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.

On entry into 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, for example when executing a **let**. 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 resumption as appropriate, or forwards the operation to an outer handler. In the latter case, the resumption is modified to ensure that the context of the original operation invocation can be reinstated when the resumption is 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 (Section 5.3.1). Currying obstructs proper tail-recursion, so we move to an explicit representation of the stack (Section 5.3.2). Then, in order to avoid administrative reductions in our final higher-order one-pass translation we use an explicit representation of resumptions (Section 5.3.3). Finally, in order to support shallow handlers, we will use an explicit stack representation for pure continuations (Section 5.3.4).

5.3.1 Curried Translation

Our first translation builds upon the CPS translation of Figure 8. The extension to operations and handlers is localised to the additional features because 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 arguments representing the stack of pure and effect continuations. Thus, the translation of core constructs remain exactly the same as in Figure 8, where we imagine there being some number of extra continuation arguments that have been η-reduced. The translation of operations and handlers is as follows:

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

The translation of **do** $\ell\ V$ abstracts over the current pure (k) and effect (h) continuations passing an encoding of the operation into the latter. The operation is encoded as a triple consisting of the name ℓ, parameter $[\![V]\!]$, and resumption $\lambda x.k\ x\ h$, which ensures that any subsequent operations are handled by the same effect continuation h.

The translation of **handle** M **with** H invokes the translation of M with new pure and effect continuations 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.

The translation of complete programs feeds the translated term the identity pure continuation (which discards its handler argument), and an effect continuation that is never intended to be called:

$$\top[\![M]\!] = [\![M]\!]\ (\lambda x.\lambda h.x)\ (\lambda\langle z, \_\rangle.\mathbf{absurd}\ z)$$

Conceptually, this translation encloses the translated program in a top-level handler with an empty collection of operation clauses and an identity return clause.

There are three shortcomings of this initial translation that we address below.

- First, it is not *properly tail-recursive* (Danvy & Filinski, 1992; Steele, 1978) due to the curried representation of the continuation stack, as a result the image of the translation is not stackless, which makes it problematic to implement using a

trampoline in, say, JavaScript. (Properly tail-recursion CPS translations ensure that all calls to functions and continuations are in tail position, hence there is no need to maintain a stack.) We rectify this issue with an explicit list representation in the next subsection.

- Second, it yields administrative redexes (redexes that could be reduced statically). We will rectify this with a higher-order one-pass translation in Section 5.4.
- Third, this translation cannot cope with shallow handlers. The pure continuations k are abstract and include the return clause of the corresponding handler. Shallow handlers require that the return clause of a handler is discarded when one of its operations is invoked. We will fix this in Section 5.3.4, where we will represent pure continuations as explicit stacks.

To illustrate the first two issues, consider the following example:

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

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 7: the function position of an application can be a computation, and the calculus makes use of evaluation contexts.

Remark We originally derived this curried CPS translation for effect handlers by composing Forster *et al.*'s translation from effect handlers to delimited continuations (2017) with Materzok & Biernacki's CPS translation for delimited continuations (2012).

Because of the administrative reductions, simulation is not on the nose, but instead up to congruence. For reduction in the untyped target calculus we write $\rightsquigarrow_{\mathrm{cong}}$ for the smallest relation containing $\rightsquigarrow$ that is closed under the term formation constructs.

Theorem 6 (*Simulation*)
If $M \rightsquigarrow N$ then $\top[\![M]\!] \rightsquigarrow^{+}_{\mathrm{cong}} \top[\![N]\!]$.

Proof
The result follows by composing a call-by-value variant of Forster *et al.*'s translation from effect handlers to delimited continuations (2017) with Materzok & Biernacki's CPS translation for delimited continuations (2012). □

5.3.2 Continuations as Explicit Stacks

Following Materzok & Biernacki we uncurry our CPS translation in order to obtain a properly tail-recursive translation, representing the stacks of pure and effect continuations

explicitly as lists. The translation of return, let binding, operations, handlers, and top level programs is adjusted as follows for the new representation:

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

The other cases are as in the original CPS translation in Figure 8. The stacks of continuations are now lists, where pure continuations and effect continuations occupy alternating positions.

Since we now use a list representation for the stacks of continuations, we have had 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 8. 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. 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 this until our higher order translation in Section 5.4.

5.3.3 Resumptions as Explicit Reversed Stacks

In the CPS translations of operations and handlers that we have defined so far, resumptions have been represented as functions, and forwarding has been implemented using function composition. In order to avoid the administrative redexes arising from 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 $\mathbf{let}\ r = \mathbf{res}\,V\ \mathbf{in}\ N$ computation term that reduces as follows:

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

This reduction rule reverses the stack, pulls out the top continuation V_1, and prepends the remainder onto the current stack W. The stack representing a resumption and the remaining stack W are reminiscent of the zipper data structure for representing cursors in lists (Huet,

1997). Resumptions can therefore be thought of as representing pointers into the stack of handlers.

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 **res** construct to convert the resumption stack into a function, and M_{forward} extends the resumption stack with the current continuation pair.

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

Since we have only changed the representation of resumptions, the translation of top-level programs remains the same.

5.3.4 Shallow Handlers: Pure Continuations as Explicit Stacks

We now extend the CPS translation to allow deep and shallow handlers. Hillerström *et al.* (2017) sketched a translation based on the CPS translations given above, but this translation unfortunately contained a bug. The problem is that the return clause is integrated into the pure continuation of each stack frame, but the semantics of shallow handlers demands that this return clause is discarded when any of the operations is invoked. Using a functional representation of pure continuations means that there is no way to remove the (translation of the) return clause. Hillerström & Lindley (2018) fixed this by switching to a more intensional representation of pure continuations as explicit stacks. We present their solution again here, relating it to the CPS translations for only deep handlers presented above.

A stack frame is now a triple $\langle fs, \langle h^{\text{ret}}, h^{\text{ops}} \rangle \rangle$, where fs is list of stack frames representing the pure continuation for the computation occurring between the current execution and the handler, h^{ret} is the (translation of the) return clause of the enclosing handler, and h^{ops} is the (translation of the) operation clauses.

Since pure continuations are no longer represented simply as functions, we cannot invoke them by simple function application. Instead, we must inspect the structure of the pure stack fs and act appropriately. To package this neatly, we introduce a computation form **app** $V\ W$ that feeds a value W into the continuation stack represented by V. There are two reduction rules:

$$\begin{aligned}
&\text{U-KAPPNIL} && \mathbf{app}\ (\langle [], \langle h^{\text{ret}}, h^{\text{ops}} \rangle \rangle :: ks)\, W \rightsquigarrow h^{\text{ret}}\, W\, ks \\
&\text{U-KAPPCONS} && \mathbf{app}\ (\langle f :: fs, h \rangle :: ks)\, W \rightsquigarrow f\, W\, (\langle fs, h \rangle :: ks)
\end{aligned}$$

The first rule describes what happens when the pure continuation stack is exhausted and the return clause of the enclosing handler is invoked. The second rule describes the case when the pure continuation stack has at least one element: this continuation is invoked and the remainder of the stack is passed in as the new continuation.

Since the representation of stacks has changed, we must also change how resumptions (i.e. reversed stacks) are converted into functions that can be applied. Resumptions for deep handlers (**res** V) are similar to the previous section, except that we now use **app** to invoke the continuation. Resumptions for shallow handlers ($\mathbf{res}^{\dagger}\ V$) are more complex. Instead of taking all the frames and reverse appending them to the current stack, we remove the

current handler h and move the pure continuation $f_1 :: \ldots :: f_m :: []$ into the next frame. This captures the intended behaviour of shallow handlers: they are removed from the stack once they have been invoked. The reduction rules describing the behaviour of resumptions are:

$$\begin{array}{ll}
\text{U-RES} & \mathbf{let}\ r = \mathbf{res}\,(V_n :: \ldots :: V_1 :: [])\ \mathbf{in}\ N \rightsquigarrow N[\lambda x\,k.\mathbf{app}\,(V_1 :: \ldots V_n :: k)\,x/r] \\
\text{U-RES}^\dagger & \mathbf{let}\ r = \mathbf{res}^\dagger\,(\langle V_{f_1} :: \ldots :: V_{f_m} :: [], h\rangle :: V_n :: \ldots :: V_1 :: [])\ \mathbf{in}\ N \rightsquigarrow \\
& \qquad N[\lambda x\,k.\mathbf{let}\,\langle fs', h'\rangle :: k' = k\ \mathbf{in} \\
& \qquad\qquad \mathbf{app}\,(V_1 :: \ldots :: V_n :: \langle V_{f_1} :: \ldots :: V_{f_m} :: fs', h'\rangle :: k')\,x/r]
\end{array}$$

These constructs along with their reduction rules are macro-expressible in terms of the existing constructs. We choose to treat them as primitives in order to keep the presentation relatively concise.

The CPS translation is modified to take into account the new representation of stacks. We can now implement both deep and shallow handlers within a single translation by choosing the appropriate way to convert resumptions via the flag δ.

$$\begin{array}{rl}
[\![\mathbf{return}\ V]\!] & = \lambda ks.\,\mathbf{app}\ ks\ [\![V]\!] \\
[\![\mathbf{let}\ x \leftarrow M\ \mathbf{in}\ N]\!] & = \lambda\,\langle fs, h\rangle :: ks.[\![M]\!]\,(\langle(\lambda x.\lambda ks.[\![N]\!]\,ks) :: fs, h\rangle :: ks) \\
[\![\mathbf{do}\ \ell\ V]\!] & = \lambda\,\langle fs, \langle h^{\mathrm{ret}}, h^{\mathrm{ops}}\rangle\rangle :: ks.\,h^{\mathrm{ops}}\,\langle \ell, \langle [\![V]\!], \langle fs, \langle h^{\mathrm{ret}}, h^{\mathrm{ops}}\rangle\rangle :: []\rangle\rangle\,ks \\
[\![\mathbf{handle}^\delta\ M\ \mathbf{with}\ H]\!] & = \lambda ks.[\![M]\!](\langle [], \langle [\![H^{\mathrm{ret}}]\!], [\![H^{\mathrm{ops}}]\!]^\delta\rangle\rangle :: ks),\ \text{where} \\
[\![\{\mathbf{return}\ x \mapsto N\}]\!] & = \lambda x.\lambda ks.[\![N]\!]\,ks \\
[\![\{\ell\ p\ r \mapsto N_\ell\}_{\ell\in\mathscr{L}}]\!]^\delta & = \lambda\,\langle z, \langle p, rk\rangle\rangle.\lambda ks.\mathbf{case}\ z\ \{(\ell \mapsto \mathbf{let}\ r = \mathbf{res}^\delta\ rk\ \mathbf{in}\ [\![N_\ell]\!]\,ks)_{\ell\in\mathscr{L}}; \\
& \qquad\qquad y \mapsto M_{\mathrm{forward}}((y, p, rk), ks)\} \\
M_{\mathrm{forward}}((y, p, rk), ks) & = \mathbf{let}\ \langle s, \langle h^{\mathrm{ret}}, h^{\mathrm{ops}}\rangle\rangle :: ks' = ks\ \mathbf{in} \\
& \qquad \mathbf{let}\ rk' = \langle s, \langle h^{\mathrm{ret}}, h^{\mathrm{ops}}\rangle\rangle :: rk\ \mathbf{in} \\
& \qquad h^{\mathrm{ops}}\,\langle y, \langle p, rk'\rangle\rangle\,ks'
\end{array}$$

The translation of top-level programs feeds in an empty stack for the pure computation, the identity return clause, and an operations clause that never expects to be invoked:

$$\top[\![M]\!] = [\![M]\!]\,(\langle [], \langle \lambda x.\lambda ks.x, \lambda\,\langle z, \langle p, rk\rangle\rangle.\lambda ks.\mathbf{absurd}\ z\rangle\rangle :: [])$$

We now have a CPS translation of $\lambda^\dagger$ that handles deep and shallow handlers, and is tail recursive. There still remains the problem of administrative reductions, making translated code less efficient than it could be. We rectify this in the next section by using a higher-order translation that eliminates administrative reductions statically.

5.4 A Higher-Order Explicit Stack Translation

We now adapt the translation of Section 5.3.4 to a higher-order one-pass CPS translation (Danvy & Filinski, 1990) that partially evaluates administrative redexes at translation time. Following Danvy & Nielsen (2003), 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: $\overline{\text{overline}}$ denotes a static syntax constructor; $\underline{\text{underline}}$ denotes a dynamic syntax constructor. 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. To facilitate this notation we write application in both calculi with an infix "at" symbol (@).

Syntax

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

Syntactic sugar

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

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

Reductions

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

$$
\begin{array}{ll}
\text{U-Res} & \mathbf{let}\ r = \mathbf{res}(V_n :: \cdots :: V_1 :: [])\ \mathbf{in}\ N \rightsquigarrow \\
& \quad N[\underline{\lambda} x k.\mathbf{let}\ \underline{\langle} fs, \underline{\langle} h^{\mathrm{ret}}, h^{\mathrm{ops}} \underline{\rangle} \underline{\rangle} :: k' = k\ \mathbf{in} \\
& \qquad \mathbf{app}\ (V_1 :: \cdots :: V_n :: \underline{\langle} fs, \underline{\langle} h^{\mathrm{ret}}, h^{\mathrm{ops}} \underline{\rangle} \underline{\rangle} :: k')\ x/r] \\
\text{U-Res}^{\dagger} & \mathbf{let}\ r = \mathbf{res}^{\dagger}(\underline{\langle} V_{f_1} :: \cdots :: V_{f_m} :: [], h \underline{\rangle} :: V_n :: \cdots :: V_1 :: [])\ \mathbf{in}\ N \rightsquigarrow \\
& \quad N[\underline{\lambda} x k.\mathbf{let}\ \underline{\langle} s', h' \underline{\rangle} :: k' = k\ \mathbf{in} \\
& \qquad \mathbf{app}\ (q_1 :: \cdots :: q_n :: \underline{\langle} f_1 :: \cdots :: f_m :: s', h' \underline{\rangle} :: k')\ x/r]
\end{array}
$$

Fig. 9. Untyped Target Calculus for the Higher-Order CPS Translation

5.4.1 Dynamic Terms: The Target Calculus

The target calculus is given in Fig. 9. This is essentially the same as the target calculus described in Section 5.1, except that the application form ($U \mathbin{\underline{@}} V \mathbin{\underline{@}} W$) comprises three values, and all applications take two arguments: a function argument and a parameter. The calculus also includes the **app** and **let** $r = \mathbf{res}^{\delta}\ V$ **in** N constructs described in Section 5.3.4. There is a small difference in the reduction rules for the resumption constructs: for deep resumptions we do an additional pattern match on the current continuation stack (k). This is required to make the simulation proof for the CPS translation described below go through, because it makes the functions that resumptions get converted to have the same shape as the translation of source level functions – this is required because our operational semantics treats resumptions as special kinds of functions, not as first-class objects in their own right. As above, the **app** and resumption constructs, with their reduction rules, are macro-expressible in terms of the other constructs of the dynamic language.

5.4.2 Static Terms

Static constructs are marked in static blue, and their redexes are reduced as part of the translation (at compile time). We make use of static lambda abstractions, pairs, and lists. Reflection of dynamic language values into the static language is written as $\uparrow V$. We use κ for variables representing statically known continuations (frame stacks), θ for variables representing pure frame stacks, and χ for variables representing handlers. We let $\mathscr{V},\mathscr{W}$ range over meta language values, $\mathscr{M}$ range over static language expressions, and $\mathscr{P},\mathscr{Q}$ over static language patterns. We use list and record pattern matching in the meta language, which behaves as follows:

$$\begin{aligned}&(\overline{\lambda}\overline{\langle \mathscr{P},\mathscr{Q}\rangle}.\mathscr{M})\,\overline{@}\,\overline{\langle \mathscr{V},\mathscr{W}\rangle}\\ =\;&(\overline{\lambda}\,\mathscr{P}.\overline{\lambda}\,\mathscr{Q}.\mathscr{M})\,\overline{@}\,\mathscr{V}\,\overline{@}\,\mathscr{W}\\ =\;&(\overline{\lambda}(\mathscr{P}\mathbin{\overline{::}}\mathscr{Q}).\mathscr{M})\,\overline{@}\,(\mathscr{V}\mathbin{\overline{::}}\mathscr{W})\end{aligned}$$

Static language values, comprised of reflected values, pairs, and list conses, are reified as dynamic language values $\downarrow\mathscr{V}$ by induction on their structure:

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

We assume the static language is pure and hence respects the usual β and η equivalences.

5.4.3 The Translation

The CPS translation is given in Fig. 10. In essence, it is the same as the CPS translation for deep and shallow handlers we described in Section 5.3.4, albeit separated into static and dynamic parts. A major difference that has a large cosmetic effect on the presentation of the translation is that we maintain the invariant that the statically known stack (κ) always contains at least one frame, consisting of a triple $\overline{\langle\uparrow V_{fs},\overline{\langle\uparrow V_{ret},\uparrow V_{ops}\rangle}\rangle}$ of reflected dynamic pure frame stacks, return handlers, and operation handlers. Maintaining this invariant ensures that all translations are uniform in whether they appear statically within the scope of a handler or not, and this simplifies our correctness proof. To maintain the invariant, any place where a dynamically known stack is passed in (as a continuation parameter k), it is immediately decomposed using a dynamic language **let** and repackaged as a static value with reflected variable names. Unfortunately, this does add some clutter to the translation definition, as compared to the translations above. However, there is a payoff in the removal of administrative reductions at run time. The translations presented by Hillerström *et al.* (2017) and Hillerström & Lindley (2018) did not do this decomposition and repackaging step, which resulted in additional administrative reductions in the translation due to the translations of **let** and **do** being passed dynamic continuations when they were expecting statically known ones.

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

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

Values

$$\begin{aligned}
[\![x]\!] &= x \\
[\![\lambda x.M]\!] &= \underline{\lambda} x\, k.\mathbf{let}\ \underline{\langle} fs, \langle h^{\mathrm{ret}}, h^{\mathrm{ops}}\rangle \underline{\rangle} :: k' = k\ \mathbf{in}\ [\![M]\!]\ \overline{@}\ (\overline{\langle} \uparrow fs, \overline{\langle} \uparrow h^{\mathrm{ret}}, \uparrow h^{\mathrm{ops}} \overline{\rangle\rangle} :: \uparrow k') \\
[\![\Lambda \alpha.M]\!] &= \underline{\lambda} z\, k.\mathbf{let}\ \underline{\langle} fs, \langle h^{\mathrm{ret}}, h^{\mathrm{ops}}\rangle \underline{\rangle} :: k' = k\ \mathbf{in}\ [\![M]\!]\ \overline{@}\ (\overline{\langle} \uparrow fs, \overline{\langle} \uparrow h^{\mathrm{ret}}, \uparrow h^{\mathrm{ops}} \overline{\rangle\rangle} :: \uparrow k') \\
[\![\mathbf{rec}\, g\, x.M]\!] &= \mathbf{rec}\, g\, x\, k.\mathbf{let}\ \underline{\langle} fs, \langle h^{\mathrm{ret}}, h^{\mathrm{ops}}\rangle \underline{\rangle} :: k' = k\ \mathbf{in}\ [\![M]\!]\ \overline{@}\ (\overline{\langle} \uparrow fs, \overline{\langle} \uparrow h^{\mathrm{ret}}, \uparrow h^{\mathrm{ops}} \overline{\rangle\rangle} :: \uparrow k') \\
[\![\langle\rangle]\!] &= \langle\rangle \qquad [\![\langle \ell = V; W\rangle]\!] = \langle \ell = [\![V]\!]; [\![W]\!]\rangle \qquad [\![\ell\, V]\!] = \ell\, [\![V]\!]
\end{aligned}$$

Computations

$$\begin{aligned}
[\![V\, W]\!] &= \overline{\lambda}\, \kappa.[\![V]\!]\ \underline{@}\ [\![W]\!]\ \underline{@} \downarrow \kappa \\
[\![V\, T]\!] &= \overline{\lambda}\, \kappa.[\![V]\!]\ \underline{@}\ \langle\rangle\ \underline{@} \downarrow \kappa \\
[\![\mathbf{let}\ \langle \ell = x; y\rangle = V\ \mathbf{in}\ N]\!] &= \overline{\lambda}\, \kappa.\mathbf{let}\ \langle \ell = x; y\rangle = [\![V]\!]\ \mathbf{in}\ [\![N]\!]\ \overline{@}\ \kappa \\
[\![\mathbf{case}\ V\ \{\ell\, x \mapsto M; y \mapsto N\}]\!] &= \overline{\lambda}\, \kappa.\mathbf{case}\ [\![V]\!]\ \{\ell\, x \mapsto [\![M]\!]\ \overline{@}\ \kappa; y \mapsto [\![N]\!]\ \overline{@}\ \kappa\} \\
[\![\mathbf{absurd}\ V]\!] &= \overline{\lambda}\, \kappa.\mathbf{absurd}\ [\![V]\!]
\end{aligned}$$

$$\begin{aligned}
[\![\mathbf{return}\, V]\!] &= \overline{\lambda}\, \kappa.\mathbf{app}\ (\downarrow \kappa)\ [\![V]\!] \\
[\![\mathbf{let}\ x \leftarrow M\ \mathbf{in}\ N]\!] &= \overline{\lambda} \overline{\langle} \theta, \overline{\langle} \chi^{\mathrm{ret}}, \chi^{\mathrm{ops}} \overline{\rangle\rangle} :: \kappa. \\
&\qquad [\![M]\!]\ \overline{@}\ (\overline{\langle} \uparrow ((\underline{\lambda} x\, k.\mathbf{let}\ \underline{\langle} fs, \langle h^{\mathrm{ret}}, h^{\mathrm{ops}}\rangle \underline{\rangle} :: k' = k\ \mathbf{in} \\
&\qquad\qquad [\![N]\!]\ \overline{@}\ (\overline{\langle} \uparrow fs, \overline{\langle} \uparrow h^{\mathrm{ret}}, \uparrow h^{\mathrm{ops}} \overline{\rangle\rangle} :: \uparrow k')) :: \downarrow \theta), \\
&\qquad \overline{\langle} \chi^{\mathrm{ret}}, \chi^{\mathrm{ops}} \overline{\rangle\rangle} :: \kappa) \\
[\![\mathbf{do}\ \ell\ V]\!] &= \overline{\lambda} \overline{\langle} \theta, \overline{\langle} \chi^{\mathrm{ret}}, \chi^{\mathrm{ops}} \overline{\rangle\rangle} :: \kappa. \downarrow \chi^{\mathrm{ops}}\ \underline{@}\ \underline{\langle} \ell, \underline{\langle} [\![V]\!], \underline{\langle} \downarrow \theta, \underline{\langle} \downarrow \chi^{\mathrm{ret}}, \downarrow \chi^{\mathrm{ops}} \underline{\rangle\rangle} :: [] \underline{\rangle\rangle}\ \underline{@} \downarrow \kappa \\
[\![\mathbf{handle}^{\delta}\, M\ \mathbf{with}\ H]\!] &= \overline{\lambda}\, \kappa.[\![M]\!]\ \overline{@}\ (\overline{\langle} \uparrow [], [\![H]\!]^{\delta} \overline{\rangle} :: \kappa) \\
[\![H]\!]^{\delta} &= \overline{\langle} \uparrow [\![H^{\mathrm{ret}}]\!], \uparrow [\![H^{\mathrm{ops}}]\!]^{\delta} \overline{\rangle} \\
[\![\{\mathbf{return}\, x \mapsto N\}]\!] &= \underline{\lambda} x\, k.\mathbf{let}\ \underline{\langle} fs, \langle h^{\mathrm{ret}}, h^{\mathrm{ops}}\rangle \underline{\rangle} :: k' = k\ \mathbf{in} \\
&\qquad [\![N]\!]\ \overline{@}\ (\overline{\langle} \uparrow fs, \overline{\langle} \uparrow h^{\mathrm{ret}}, \uparrow h^{\mathrm{ops}} \overline{\rangle\rangle} :: \uparrow k') \\
[\![\{(\ell\ p\ r \mapsto N_\ell)_{\ell \in \mathscr{L}}\}]\!]^{\delta} &= \underline{\lambda} \underline{\langle} z, \underline{\langle} p, rk \underline{\rangle\rangle}\, k.\mathbf{case}\ z\ \{(\ell \mapsto \mathbf{let}\ r = \mathbf{res}^{\delta}\ rk\ \mathbf{in} \\
&\qquad\qquad \mathbf{let}\ \underline{\langle} fs, \langle h^{\mathrm{ret}}, h^{\mathrm{ops}}\rangle \underline{\rangle} :: k' = k\ \mathbf{in} \\
&\qquad\qquad [\![N_\ell]\!]\ \overline{@}\ (\overline{\langle} \uparrow fs, \overline{\langle} \uparrow h^{\mathrm{ret}}, \uparrow h^{\mathrm{ops}} \overline{\rangle\rangle} :: \uparrow k'))_{\ell \in \mathscr{L}} \\
&\qquad y \mapsto M_{\mathrm{forward}}((y, p, rk), k)\} \\
M_{\mathrm{forward}}((y, p, rk), k) &= \mathbf{let}\ \underline{\langle} fs, \langle h^{\mathrm{ret}}, h^{\mathrm{ops}}\rangle \underline{\rangle} :: k' = k\ \mathbf{in} \\
&\quad \mathbf{let}\ rk' = \underline{\langle} fs, \langle h^{\mathrm{ret}}, h^{\mathrm{ops}}\rangle \underline{\rangle} :: rk\ \mathbf{in} \\
&\quad h^{\mathrm{ops}}\ \underline{@}\ \underline{\langle} y, \underline{\langle} p, rk' \underline{\rangle\rangle}\ \underline{@}\ k'
\end{aligned}$$

Top-level program

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

Fig. 10. Higher-Order Uncurried CPS Translation of $\lambda^{\dagger}$

where $\mathscr{K}_{\top} = (\overline{\langle} [], \overline{\langle} \uparrow \underline{\lambda} x.\underline{\lambda} k.x, \uparrow \underline{\lambda} \underline{\langle} z, \underline{\langle} p, rk \underline{\rangle\rangle}.\underline{\lambda} k.\mathbf{absurd}\ z \overline{\rangle\rangle} :: \uparrow [])$ is the top-level handler. The resulting term passes Await directly to the dispatcher that implements the operation clauses of H. In contrast, our original first-order curried CPS translation yields the term

$$(\lambda k.\lambda h.h\ \langle \mathsf{Await}, \langle\rangle, \lambda x.k\ h\ x\rangle)\ [\![H^{\mathrm{ret}}]\!]\ [\![H^{\mathrm{ops}}]\!]\ K_{\top}\ H_{\top}$$

where $K_{\top} = \lambda x.\lambda h.x$ and $H_{\top} = \lambda \langle z, \_\rangle.\mathbf{absurd}\ z$, requiring two administrative reductions

$$[\![H^{\mathrm{ops}}]\!]\ \langle \mathsf{Await}, \langle\rangle, \lambda x.[\![H^{\mathrm{ret}}]\!]\ [\![H^{\mathrm{ops}}]\!]\ x\rangle\rangle\ K_{\top}\ H_{\top}$$

before Await may be passed to the dispatcher.

5.4.4 Correctness

To prove the correctness of our CPS translation (Theorem 7), we first state several lemmas describing how translated terms behave. In view of the invariant of the translation that we described above, we state each of these lemmas in terms of static continuation stacks where the shape of the top element is always known statically, i.e., it is of the form $\overline{\langle}\mathscr{V}_{fs}, \overline{\langle}\mathscr{V}_{ret}, \mathscr{V}_{ops}\overline{\rangle\rangle} :: \mathscr{W}$. Moreover, the static values $\mathscr{V}_{fs}$, $\mathscr{V}_{ret}$, and $\mathscr{V}_{ops}$ are all reflected dynamic terms (i.e., of the form $\uparrow V$). This fact is used implicitly in our proofs, which are given in Appendix A.

First, the higher-order CPS translation commutes with substitution:

Lemma 2 (*Substitution*)
The CPS translation $[\![-]\!]$ commutes with substitution in value terms

$$[\![W]\!][[\![V]\!]/x] = [\![W[V/x]]\!],$$

and with substitution in computation terms

$$\begin{aligned}&([\![M]\!] \mathbin{\overline{@}} (\overline{\langle}\mathscr{V}_{fs}, \overline{\langle}\mathscr{V}_{ret}, \mathscr{V}_{ops}\overline{\rangle\rangle} :: \mathscr{W}))[[\![V]\!]/x]\\ &= [\![M[V/x]]\!] \mathbin{\overline{@}} (\overline{\langle}\mathscr{V}_{fs}, \overline{\langle}\mathscr{V}_{ret}, \mathscr{V}_{ops}\overline{\rangle\rangle} :: \mathscr{W})[[\![V]\!]/x].\end{aligned}$$

In order to reason about the behaviour of the S-Op and S-Op† rules, which are defined in terms of evaluation contexts, we extend the CPS translation to evaluation contexts, using the same translations as for the corresponding constructs in $\lambda^\dagger$:

$$\begin{aligned}[\![[\,]]\!] &= \overline{\lambda}\,\kappa.\kappa\\ [\![\mathbf{let}\ x \leftarrow \mathscr{E}\ \mathbf{in}\ N]\!] &= \overline{\lambda}\,\overline{\langle}\theta, \overline{\langle}\chi^{\mathrm{ret}}, \chi^{\mathrm{ops}}\overline{\rangle\rangle} :: \kappa.\\ &\qquad [\![\mathscr{E}]\!] \mathbin{\overline{@}} (\overline{\langle}\uparrow((\underline{\lambda}\,x\,k.\mathbf{let}\ \underline{\langle}fs, \underline{\langle}h^{\mathrm{ret}}, h^{\mathrm{ops}}\underline{\rangle\rangle} :: k' = k\ \mathbf{in}\\ &\qquad\qquad [\![N]\!] \mathbin{\overline{@}} (\overline{\langle}\uparrow fs, \overline{\langle}\uparrow h^{\mathrm{ret}}, \uparrow h^{\mathrm{ops}}\overline{\rangle\rangle} :: \uparrow k')) :: \downarrow\theta),\\ &\qquad \overline{\langle}\chi^{\mathrm{ret}}, \chi^{\mathrm{ops}}\overline{\rangle\rangle} :: \kappa)\\ [\![\mathbf{handle}^\delta\ \mathscr{E}\ \mathbf{with}\ H]\!] &= \overline{\lambda}\,\kappa.[\![\mathscr{E}]\!] \mathbin{\overline{@}} (\overline{\langle}[], [\![H]\!]^\delta\overline{\rangle} :: \kappa)\end{aligned}$$

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

Lemma 3 (*Decomposition*)
$[\![\mathscr{E}[M]]\!] \mathbin{\overline{@}} (\overline{\langle}\mathscr{V}_{fs}, \overline{\langle}\mathscr{V}_{ret}, \mathscr{V}_{ops}\overline{\rangle\rangle} :: \mathscr{W}) = [\![M]\!] \mathbin{\overline{@}} ([\![\mathscr{E}]\!] \mathbin{\overline{@}} (\overline{\langle}\mathscr{V}_{fs}, \overline{\langle}\mathscr{V}_{ret}, \mathscr{V}_{ops}\overline{\rangle\rangle} :: \mathscr{W}))$.

By definition, reifying a reflected dynamic value is the identity ($\downarrow\uparrow V = V$), but we also need to reason about the inverse composition. As a result of the invariant that the translation always has static access to the top of the handler stack, the translated terms are insensitive to whether the remainder of the stack is statically known or is a reflected version of a reified stack. This is captured by the following lemma. The proof is by induction on the structure of M (after generalising the statement to stacks of arbitrary depth), and relies on the observation that translated terms either access the top of the handler stack, or reify the stack to use dynamically, whereupon the distinction between reflected and reified becomes void. Again, this lemma holds when the top of the static continuation stack is known:

Lemma 4 (*Reflect after reify*)
$[\![M]\!] \mathbin{\overline{@}} (\overline{\langle}\mathscr{V}_{fs}, \overline{\langle}\mathscr{V}_{ret}, \mathscr{V}_{ops}\overline{\rangle\rangle} :: \uparrow\downarrow\mathscr{W}) = [\![M]\!] \mathbin{\overline{@}} (\overline{\langle}\mathscr{V}_{fs}, \overline{\langle}\mathscr{V}_{ret}, \mathscr{V}_{ops}\overline{\rangle\rangle} :: \mathscr{W})$.

The next lemma states that the CPS translation correctly simulates forwarding. The proof is by inspection of how the translation of operation clauses treats non-handled operations.

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

$$[\![H_1^{\mathrm{ops}}]\!]^{\delta} \mathbin{@} \langle \ell, \langle V_p, V_{rk} \rangle \rangle \mathbin{@} (\langle V_{fs}, [\![H_2]\!]^{\delta} \rangle :: W) \rightsquigarrow^{+} [\![H_2^{\mathrm{ops}}]\!]^{\delta} \mathbin{@} \langle \ell, \langle V_p, \langle V_{fs}, [\![H_2]\!]^{\delta} \rangle :: V_{rk} \rangle \rangle \mathbin{@} W$$

The following lemma is central to our simulation theorem. It characterises the sense in which the translation respects the handling of operations. Note how the values substituted for the resumption variable r in both cases are in the image of the translation of λ-terms in the CPS translation. This is thanks to the precise way that the reductions rules for resumption construction works in our dynamic language, as described above.

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

1. $[\![\mathbf{do}\ \ell\ V]\!] \mathbin{\overline{@}} ([\![\mathscr{E}]\!] \mathbin{\overline{@}} (\overline{\langle} \uparrow [], [\![H]\!] \overline{\rangle} :: \overline{\langle} \mathscr{V}_{fs}, \overline{\langle} \mathscr{V}_{ret}, \mathscr{V}_{ops} \overline{\rangle\rangle} :: \mathscr{W})) \rightsquigarrow^{+}$
   $([\![N_{\ell}]\!] \mathbin{\overline{@}} \overline{\langle} \mathscr{V}_{fs}, \overline{\langle} \mathscr{V}_{ret}, \mathscr{V}_{ops} \overline{\rangle\rangle} :: \mathscr{W})$
   $[[\![V]\!]/p, \underline{\lambda} y\, k.\mathbf{let}\ \langle fs, \langle h^{\mathrm{ret}}, h^{\mathrm{ops}} \rangle \rangle :: k' = k\ \mathbf{in}$
   $[\![\mathbf{return}\ y]\!] \mathbin{\overline{@}} ([\![\mathscr{E}]\!] \mathbin{\overline{@}} (\overline{\langle} \uparrow [], [\![H]\!] \overline{\rangle} :: \overline{\langle} \uparrow s, \overline{\langle} \uparrow h^{\mathrm{ret}}, \uparrow h^{\mathrm{ops}} \overline{\rangle\rangle} :: \uparrow k'))/r],$
2. $[\![\mathbf{do}\ \ell\ V]\!] \mathbin{\overline{@}} ([\![\mathscr{E}]\!] \mathbin{\overline{@}} (\overline{\langle} \uparrow [], [\![H]\!]^{\dagger} \overline{\rangle} :: \overline{\langle} \mathscr{V}_{fs}, \overline{\langle} \mathscr{V}_{ret}, \mathscr{V}_{ops} \overline{\rangle\rangle} :: \mathscr{W})) \rightsquigarrow^{+}$
   $([\![N_{\ell}]\!] \mathbin{\overline{@}} \overline{\langle} \mathscr{V}_{fs}, \overline{\langle} \mathscr{V}_{ret}, \mathscr{V}_{ops} \overline{\rangle\rangle} :: \mathscr{W})$
   $[[\![V]\!]/p, \underline{\lambda} y\, k.\mathbf{let}\ \langle s, \langle h^{\mathrm{ret}}, h^{\mathrm{ops}} \rangle \rangle :: k' = k\ \mathbf{in}$
   $[\![\mathbf{return}\ y]\!] \mathbin{\overline{@}} ([\![\mathscr{E}]\!] \mathbin{\overline{@}} (\overline{\langle} \uparrow s, \overline{\langle} \uparrow h^{\mathrm{ret}}, \uparrow h^{\mathrm{ops}} \overline{\rangle\rangle} :: \uparrow k'))/r].$

Now our main result for the translation: a simulation result in the style of Plotkin (1975).

Theorem 7 (*Simulation*)
If $M \rightsquigarrow N$ then $[\![M]\!] \mathbin{\overline{@}} (\overline{\langle} \mathscr{V}_{fs}, \overline{\langle} \mathscr{V}_{ret}, \mathscr{V}_{ops} \overline{\rangle\rangle} :: \mathscr{W}) \rightsquigarrow^{+} [\![N]\!] \mathbin{\overline{@}} (\overline{\langle} \mathscr{V}_{fs}, \overline{\langle} \mathscr{V}_{ret}, \mathscr{V}_{ops} \overline{\rangle\rangle} :: \mathscr{W})$.

Proof
The proof is by case analysis on the reduction relation using Lemmas 3–6. In particular, the S-Op and S-Op† cases follow from Lemma 6. □

In common with most CPS translations, full abstraction does not hold (a function could count the number of handlers it is invoked within by examining the continuation stack, for example). However, as our semantics is deterministic it is straightforward to show a backward simulation result.

Lemma 7 (*Backwards simulation*)
If $\top[\![M]\!] \rightsquigarrow^{+} V$ then there exists W such that $M \rightsquigarrow^{*} W$ and $\top[\![W]\!] = V$.

Corollary 1
$M \rightsquigarrow^{*} V$ if and only if $\top[\![M]\!] \rightsquigarrow^{*} \top[\![V]\!]$.

$$
\begin{array}{lll}
\text{Configurations} & \mathscr{C} ::= \langle M \mid \gamma \mid \kappa \circ \kappa' \rangle & \\
\text{Value environments} & \gamma ::= \emptyset \mid \gamma[x \mapsto v] & \\
\text{Values} & v, w ::= (\gamma, \lambda x^A.M) \mid (\gamma, \Lambda\alpha^K.M) & \\
 & \quad \mid \ \langle\rangle \mid \langle \ell = v; w \rangle \mid (\ell\, v)^R \mid \kappa^A \mid (\kappa, \sigma)^A & \\
\text{Continuations} & \kappa ::= [] \mid \theta :: \kappa & \text{Continuation frames} \quad \theta ::= (\sigma, \chi) \\
 & & \text{Handler closures} \quad \chi ::= (\gamma, H)^\delta \\
\text{Pure continuations} & \sigma ::= [] \mid \phi :: \sigma & \text{Pure continuation frames} \quad \phi ::= (\gamma, x, N)
\end{array}
$$

Fig. 11. Abstract Machine Syntax

6 Abstract Machine

In this section we develop an abstract machine that supports deep and shallow handlers *simultaneously*, using the generalised continuation structure we identified in the previous section for the CPS translation. We also build upon prior work (Hillerström & Lindley, 2016) that developed an abstract machine for deep handlers by generalising the continuation structure of a CEK machine (Control, Environment, Kontinuation) (Felleisen & Friedman, 1986). Hillerström & Lindley (2016) sketched an adaptation for shallow handlers. It turns out that this adaptation had a subtle flaw, similar to the flaw in the sketched implementation of a CPS translation for shallow handlers given by Hillerström *et al.* (2017). We fix the flaw here with a full development of shallow handlers along with a statement of the correctness property.

6.1 The Machine

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), analogous to the structured frames used in the CPS translation in Section 5.4. 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. As in the CPS translation, 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. 11. A configuration $\mathscr{C} = \langle M \mid \gamma \mid \kappa \circ \kappa' \rangle$ of our abstract machine is a quadruple of a computation term (M), an

Transition function

$$
\begin{array}{lll}
\textsc{M-Init} & M \longrightarrow \langle M \mid \emptyset \mid [([], (\emptyset, \{\mathbf{return}\, x \mapsto \mathbf{return}\, 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\ T \mid \gamma \mid \kappa \rangle \longrightarrow \langle M[T/\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, & \text{if } [\![V]\!]\gamma = \ell\, v \\ \langle N \mid \gamma[y \mapsto \ell'\, v] \mid \kappa \rangle, & \text{if } [\![V]\!]\gamma = \ell'\, v \text{ and } \ell \neq \ell' \end{cases} & \\
\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. 12. Abstract Machine Semantics

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 Section 6.2 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 $\mathbf{let}\ x = [\,]\ \mathbf{in}\ N$ 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.

The abstract machine semantics defining the transition function $\longrightarrow$ is given in Fig. 12. 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. The (M-INIT) rule provides a canonical way to map a computation term onto a configuration.

Example To make the transition rules in Figure 12 concrete we give an example of the abstract machine in action. We reuse the small producer and consumer from Section 2.6. We reproduce their definitions here in ANF.

$$\mathsf{ones} \stackrel{\text{def}}{=} \mathbf{rec}\ \mathit{ones}\ \langle\rangle.\mathbf{do}\ \mathsf{Yield}\ 1; \mathit{ones}\ \langle\rangle$$
$$\mathsf{add}_2 \stackrel{\text{def}}{=} \lambda\langle\rangle.\mathbf{let}\ x \leftarrow \mathbf{do}\ \mathsf{Await}\ \langle\rangle\ \mathbf{in}\ \mathbf{let}\ y \leftarrow \mathbf{do}\ \mathsf{Await}\ \langle\rangle\ \mathbf{in}\ x + y$$

Let N_x denote the term $\mathbf{let}\ x \leftarrow \mathbf{do}\ \mathsf{Await}\ \langle\rangle\ \mathbf{in}\ N_y$ and N_y the term $\mathbf{let}\ y \leftarrow \mathbf{do}\ \mathsf{Await}\ \langle\rangle\ \mathbf{in}\ x + y$. Suppose ones, add_2, pipe, and copipe are bound in $\gamma_\top$. Furthermore, let H_{pipe} and H_{copipe}

denote the pipe and copipe handler definitions. The machine begins by applying pipe.

$$
\begin{array}{ll}
& \langle \mathsf{pipe}\ \langle \mathsf{ones}, \mathsf{add}_2 \rangle \mid \gamma_\top \mid \kappa_{\mathrm{id}} \rangle \\
\longrightarrow & \quad \text{(apply pipe)} \\
& \langle \mathbf{handle}^\dagger\ c\ \langle\rangle\ \mathbf{with}\ H_{\mathsf{pipe}} \mid \gamma_\top[c \mapsto (\emptyset, \mathsf{add}_2), p \mapsto (\emptyset, \mathsf{ones})] \mid \kappa_{\mathrm{id}} \rangle \\
\longrightarrow & \quad \text{(install } H_{\mathsf{pipe}} \text{ with } \gamma_{\mathsf{pipe}} = \gamma_\top[c \mapsto (\emptyset, \mathsf{add}_2), p \mapsto (\emptyset, \mathsf{ones})]) \\
& \langle c\ \langle\rangle \mid \gamma_{\mathsf{pipe}} \mid ([], (\gamma_{\mathsf{pipe}}, H_{\mathsf{pipe}})) :: \kappa_{\mathrm{id}} \rangle \\
\longrightarrow & \quad \text{(apply } c \text{ and } [\![c]\!]\gamma_{\mathsf{pipe}} = (\emptyset, \mathsf{add}_2)) \\
& \langle N_x \mid \emptyset \mid ([], (\gamma_{\mathsf{pipe}}, H_{\mathsf{pipe}})) :: \kappa_{\mathrm{id}} \rangle \\
\longrightarrow & \quad \text{(focus on left operand)} \\
& \langle \mathbf{do}\ \mathsf{Await}\ \langle\rangle \mid \emptyset \mid ([(\emptyset, x, N_y)], (\gamma_{\mathsf{pipe}}, H_{\mathsf{pipe}})) :: \kappa_{\mathrm{id}} \rangle \\
\longrightarrow & \quad \text{(shallow continuation capture } v_{\mathsf{Await}} = ([], [(\emptyset, x, N_y)])) \\
& \langle \mathsf{copipe}\ \langle \mathit{resume}, p \rangle \mid \gamma_{\mathsf{pipe}}[\mathit{resume} \mapsto v_{\mathsf{Await}}] \mid \kappa_{\mathrm{id}} \rangle
\end{array}
$$

The invocation of Await begins a search through the machine continuation to locate a matching handler. In this instance the top-most handler H_{pipe} handles Await. The complete shallow resumption consists of an empty continuation and a singleton pure continuation. The former is empty as H_{pipe} is a shallow handler, meaning that it is discarded.

Evaluation continues by applying copipe.

$$
\begin{array}{ll}
\longrightarrow & \quad \text{(apply copipe)} \\
& \langle \mathbf{handle}^\dagger\ p\ \langle\rangle\ \mathbf{with}\ H_{\mathsf{copipe}} \mid \gamma_\top[c \mapsto v_{\mathsf{Await}}, p \mapsto (\emptyset, \mathsf{ones})] \mid \kappa_{\mathrm{id}} \rangle \\
\longrightarrow & \quad \text{(install } H_{\mathsf{copipe}} \text{ with } \gamma_{\mathsf{copipe}} = \gamma_\top[c \mapsto v_{\mathsf{Await}}, p \mapsto (\emptyset, \mathsf{ones})]) \\
& \langle p\ \langle\rangle \mid \gamma_{\mathsf{copipe}} \mid (\emptyset, (\gamma_{\mathsf{copipe}}, H_{\mathsf{copipe}})) :: \kappa_{\mathrm{id}} \rangle \\
\longrightarrow^2 & \quad \text{(apply } p, [\![p]\!]\gamma_{\mathsf{copipe}} = (\emptyset, \mathsf{ones}), \text{ and } \gamma_{\mathsf{ones}} = \emptyset[\mathit{ones} \mapsto (\emptyset, \mathsf{ones})]) \\
& \langle \mathbf{do}\ \mathsf{Yield}\ 1 \mid \gamma_{\mathsf{ones}} \mid ([(\gamma_{\mathsf{ones}}, \_, \mathit{ones}\ \langle\rangle)], (\gamma_{\mathsf{copipe}}, H_{\mathsf{copipe}})) :: \kappa_{\mathrm{id}} \rangle \\
\longrightarrow & \quad \text{(shallow continuation capture } v_{\mathsf{Yield}} = ([], [(\gamma_{\mathsf{ones}}, \_, \mathit{ones}\ \langle\rangle)])) \\
& \langle \mathsf{pipe}\ \langle \mathit{resume}, \lambda\langle\rangle.c\ s \rangle \mid \gamma_{\mathsf{copipe}}[s \mapsto 1, \mathit{resume} \mapsto v_{\mathsf{Yield}}] \mid \kappa_{\mathrm{id}} \rangle
\end{array}
$$

At this point the situation is similar as before: the invocation of Yield causes the continuation to be unwound in order to find an appropriate handler, which happens to be H_{copipe}. Next pipe is applied.

$$
\begin{array}{ll}
\longrightarrow & \quad \text{(apply pipe and } \gamma'_{\mathsf{pipe}} = \gamma_\top[c \mapsto (\gamma_{\mathsf{copipe}}[c \mapsto v_{\mathsf{Await}}, s \mapsto 1]), p \mapsto v_{\mathsf{Yield}}])) \\
& \langle \mathbf{handle}^\dagger\ c\ \langle\rangle\ \mathbf{with}\ H_{\mathsf{pipe}} \mid \gamma'_{\mathsf{pipe}} \mid \kappa_{\mathrm{id}} \rangle \\
\longrightarrow & \quad \text{(install } H_{\mathsf{pipe}}) \\
& \langle c\ \langle\rangle \mid \gamma'_{\mathsf{pipe}} \mid ([], (\gamma'_{\mathsf{pipe}}, H_{\mathsf{pipe}})) :: \kappa_{\mathrm{id}} \rangle \\
\longrightarrow & \quad \text{(apply } c \text{ and } [\![c]\!]\gamma'_{\mathsf{pipe}} = (\gamma_{\mathsf{copipe}}[c \mapsto v_{\mathsf{Await}}, s \mapsto 1])) \\
& \langle c\ s \mid \gamma_{\mathsf{copipe}}[c \mapsto v_{\mathsf{Await}}, s \mapsto 1] \mid ([], (\gamma'_{\mathsf{pipe}}, H_{\mathsf{pipe}})) :: \kappa_{\mathrm{id}} \rangle \\
\longrightarrow & \quad \text{(shallow resume with } v_{\mathsf{Await}} = ([], [(\emptyset, x, N_y)])) \\
& \langle \mathbf{return}\ 1 \mid \gamma'_{\mathsf{pipe}} \mid ([(\emptyset, x, N_y)], (\gamma'_{\mathsf{pipe}}, H_{\mathsf{pipe}})) :: \kappa_{\mathrm{id}} \rangle
\end{array}
$$

Applying the resumption concatenates the first component (the continuation) with the machine continuation. The second component (pure continuation) gets concatenated with the pure continuation of the top-most frame of the machine continuation. Thus in this particular instance, the machine continuation is manipulated as follows.

$$
\begin{array}{l}
[] +\!\!+ ([(\emptyset, x, N_y)] +\!\!+ [], (\gamma'_{\mathsf{pipe}}, H_{\mathsf{pipe}})) :: \kappa_{\mathrm{id}} \\
= ([(\emptyset, x, N_y)], (\gamma'_{\mathsf{pipe}}, H_{\mathsf{pipe}})) :: \kappa_{\mathrm{id}}
\end{array}
$$

Because the control component contains the expression **return** 1 and the pure continuation is nonempty, the machine applies the pure continuation.

$$\begin{array}{ll}
\longrightarrow & \text{(apply pure continuation, } \gamma_{\mathsf{add}_2} = \emptyset[x \mapsto 1]) \\
& \langle N_y \mid \gamma_{\mathsf{add}_2} \mid ([], (\gamma'_{\mathsf{pipe}}, H_{\mathsf{pipe}})) :: \kappa_{\mathrm{id}} \rangle \\
\longrightarrow & \text{(focus on right operand)} \\
& \langle \mathbf{do}\ \mathsf{Await}\ \langle\rangle \mid \gamma_{\mathsf{add}_2} \mid ([(\gamma_{\mathsf{add}_2}, y, x+y)], (\gamma'_{\mathsf{pipe}}, H_{\mathsf{pipe}})) :: \kappa_{\mathrm{id}} \rangle \\
\longrightarrow^2 & \text{(shallow continuation capture } w_{\mathsf{Await}} = ([], [(\gamma_{\mathsf{add}_2}, y, x+y)]), \text{ apply copipe)} \\
& \langle \mathbf{handle}^{\dagger}\ p\ \langle\rangle\ \mathbf{with}\ \gamma_{\mathsf{copipe}} \mid \gamma_{\top}[c \mapsto w_{\mathsf{Await}}, p \mapsto v_{\mathsf{Yield}}] \mid \kappa_{\mathrm{id}} \rangle \\
\rightsquigarrow & \text{(install } H_{\mathsf{copipe}} \text{ with } \gamma_{\mathsf{copipe}} = \gamma_{\top}[c \mapsto w_{\mathsf{Await}}, p \mapsto v_{\mathsf{Yield}}]) \\
& \langle p\ \langle\rangle \mid \gamma_{\mathsf{copipe}} \mid ([], (\gamma_{\mathsf{copipe}}, H_{\mathsf{copipe}})) :: \kappa_{\mathrm{id}} \rangle
\end{array}$$

The variable p is bound to the shallow resumption v_{Yield}, thus invoking it will transfer control back to the ones computation.

$$\begin{array}{ll}
\longrightarrow & \text{(shallow resume with } v_{\mathsf{Yield}} = ([], [(\gamma_{\mathsf{ones}}, _, \mathit{ones}\ \langle\rangle)])) \\
& \langle \mathbf{return}\ \langle\rangle \mid \gamma_{\mathsf{copipe}} \mid ([(\gamma_{\mathsf{ones}}, _, \mathit{ones}\ \langle\rangle)], (\gamma_{\mathsf{copipe}}, H_{\mathsf{copipe}})) :: \kappa_{\mathrm{id}} \rangle \\
\longrightarrow^3 & \text{(apply pure continuation, apply } \mathit{ones}\text{, focus on } \mathsf{Yield}) \\
& \langle \mathbf{do}\ \mathsf{Yield}\ 1 \mid \gamma_{\mathsf{ones}} \mid ([(\gamma_{\mathsf{ones}}, _, \mathit{ones}\ \langle\rangle)], (\gamma_{\mathsf{copipe}}, H_{\mathsf{copipe}})) :: \kappa_{\mathrm{id}} \rangle
\end{array}$$

At this stage the machine repeats the transitions from before: the shallow continuation of **do** Yield 1 is captured, control passes to the Yield clause in H_{copipe}, which again invokes pipe and subsequently installs the H_{pipe} handler with an environment γ''_{pipe}. The handler runs the computation $c\ \langle\rangle$, where c is an abstraction over the resumption w_{Await} applied to the yielded value 1.

$$\begin{array}{ll}
\longrightarrow^6 & \text{(by the above reasoning, shallow resume with } w_{\mathsf{Await}} = ([], [(\gamma_{\mathsf{add}_2}, y, x+y)])) \\
& \langle x+y \mid \gamma_{\mathsf{add}_2}[y \mapsto 1] \mid ([], (\gamma''_{\mathsf{pipe}}, H_{\mathsf{pipe}})) :: \kappa_{\mathrm{id}} \rangle \\
\longrightarrow & (\llbracket x \rrbracket \gamma_{\mathsf{add}_2}[y \mapsto 1] = 1 \text{ and } \llbracket y \rrbracket \gamma_{\mathsf{add}_2}[y \mapsto 1] = 1) \\
& \langle \mathbf{return}\ 2 \mid \gamma_{\mathsf{add}_2}[y \mapsto 1] \mid ([], (\gamma''_{\mathsf{pipe}}, H_{\mathsf{pipe}})) :: \kappa_{\mathrm{id}} \rangle
\end{array}$$

Since the pure continuation is empty the **return** clause of H_{pipe} gets invoked with the value 2. Afterwards the **return** clause of the identity continuation in κ_{id} is invoked, ultimately transitioning to the following final configuration.

$$\begin{array}{ll}
\longrightarrow^2 & \text{(by the above reasoning)} \\
& \langle \mathbf{return}\ 2 \mid \emptyset \mid [] \rangle
\end{array}$$

Remark If the main continuation is empty then the machine gets stuck. This occurs when an operation is unhandled, and the forwarding continuation describes the succession of handlers that have failed to handle the operation along with any pure continuations that were encountered along the way. Assuming the input is a well-typed closed computation term $\vdash M : A!E$, the machine will either not terminate, return a value of type A, or get stuck failing to handle an operation appearing in E. We now make the correspondence between the operational semantics and the abstract machine more precise.

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

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

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. 13. Mapping from Abstract Machine Configurations to Terms

6.2 Correctness

Fig. 13 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 by how they relate to the operational semantics. The rules (M-INIT) and (M-RETTOP) concern only 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†),

$$
\begin{array}{rl}
F & ::= \cdots \mid \langle C, A \rangle \Rightarrow^{\ddagger} D \\
\delta & ::= \cdots \mid \ddagger \\
M, N & ::= \cdots \mid \mathbf{handle}^{\ddagger}\ M\ \mathbf{with}\ H^{\ddagger}(W) \\
H^{\ddagger} & ::= q^{A}.\ H
\end{array}
$$

Fig. 14. Syntax Extensions for Parameterised Handlers

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 (§3.2) implements the top-level identity continuation.

Theorem 8 (*Simulation*)
If $M \rightsquigarrow N$, then for any $\mathscr{C}$ such that $(\!|\mathscr{C}|\!) = Id(M)$ there exists $\mathscr{C}'$ such that $\mathscr{C} \Longrightarrow \mathscr{C}'$ and $(\!|\mathscr{C}'|\!) = Id(N)$.

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

Corollary 2
If $\vdash M : A!E$ and $M \rightsquigarrow^{+} N \not\rightsquigarrow$, then $M \longrightarrow^{+} \mathscr{C}$ with $(\!|\mathscr{C}|\!) = N$.

7 Parameterised Handlers

In Section 2.5 we informally presented parameterised handlers as a useful idiom for handling stateful computations. We now consider parameterised handlers as a primitive in $\lambda^{\dagger}$, and show how to extend the CPS and abstract machine implementations to support them. We also show that parameterised handlers can always be simulated by deep handlers.

7.1 Syntax and Semantics

Syntax Figure 14 extends the syntax of $\lambda^{\dagger}$ with parameterised handlers. Syntactically a parameterised handler is new binding form $(q^{A}.\ H)$, where q is the name of the parameter, whose type is A. The name is bound in the ordinary handler definition H. The elimination form ($\mathbf{handle}^{\ddagger}\ M\ \mathbf{with}\ H^{\ddagger}(W)$) is similar to the form for ordinary deep handlers, except that the parameterised handler definition is applied to a value W, which is the initial value of the parameter q.

Typing and Dynamic Semantics We require two additional rules to type parameterised handlers. The rules are given in Figure 15. The main differences between the T-HANDLER and T-HANDLER‡ are that in the latter the return and operation cases are typed with respect to the parameter q, and that resumptions r have type $\langle B_\ell, A' \rangle \to D$, that is they accept a pair as input. Operationally, a parameterised resumption uses the first component as the return

$$\textsc{T-Param-Handle}\quad \frac{\Gamma \vdash M : C \qquad \Gamma \vdash W : A \qquad \Gamma \vdash H^{\ddagger} : \langle C, A \rangle \Rightarrow D}{\Gamma \vdash \mathbf{handle}^{\ddagger}\ M\ \mathbf{with}\ H^{\ddagger}(W) : D}$$

$$\textsc{T-Handler}^{\ddagger}\quad \frac{\begin{array}{l} C = A!\{(\ell_i : A_i \to B_i)_i; R\} \\ D = B!\{(\ell_i : P)_i; R\} \\ H = \{\mathbf{return}\ x \mapsto M\} \uplus \{\ell_i\ p_i\ r_i \mapsto N_i\}_i \\ \Delta; \Gamma, q : A', x : A \vdash M : D \\ [\Delta; \Gamma, q : A', p_i : A_i, r_i : B_i \to D \vdash N_i : D]_i \end{array}}{\Delta; \Gamma \vdash H : \langle C, A' \rangle \Rightarrow^{\ddagger} D}$$

Fig. 15. Typing Rules for Parameterised Handlers

value of the operation, and the second component as the updated value of the handler parameter q. This operational behaviour is formalised by following reduction rule S-Op‡.

$$\begin{array}{l} \mathbf{handle}^{\ddagger}\ \mathscr{E}[\mathbf{do}\ \ell\ V]\ \mathbf{with}\ (q.\ H)(W) \\ \rightsquigarrow N[V/p, W/q, \lambda\langle y, q'\rangle.\mathbf{handle}\ \mathscr{E}[\mathbf{return}\ y]\ \mathbf{with}\ (q.\ H)(q')/r] \\ \quad \text{where } \ell \notin BL(\mathscr{E}) \text{ and } H^{\ell} = \{\ell\ p\ r \mapsto N\} \end{array}$$

The parameter value W is substituted for the parameter name q into the operation case body N. As with ordinary deep handlers, the resumption rewraps its handler, but with the slight twist that the parameterised handler definition is applied to the updated parameter value q' rather than the original value W. The reduction rule for handling the return of a computation is as follows.

$$\mathbf{handle}^{\ddagger}\ (\mathbf{return}\ V)\ \mathbf{with}\ (q.\ H)(W) \rightsquigarrow N[V/x, W/q], \text{where } H^{\text{ret}} = \{\mathbf{return}\ x \mapsto N\}$$

Both the return value V and the parameter value W are substituted into the return case body N for their respective binders.

7.2 Implementing Parameterised Handlers

Continuation Passing Style To accommodate parameterised handlers, we generalise the notion of continuations once more. A continuation becomes a triple consisting of a pure continuation, effect continuation, and the handler parameter. This effectively amounts to explicit state passing as the parameter value gets threaded through every function application. The pure continuation invocation rule U-KAppNil is slightly modified to account for the third component.

$$\mathbf{app}\ (\underline{\langle [], \underline{\langle h^{\text{ret}}, h^{\text{ops}}, q\rangle}\rangle} :: k)\ V \rightsquigarrow h^{\text{ret}}\ \underline{@}\ \langle V, q\rangle\ \underline{@}\ k$$

The pure continuation v is now applied to a pair consisting of the return value V and the current value of the handler parameter q. The resumption rule U-Res is adapted to update the value of the handler parameter.

$$\begin{array}{l} \mathbf{let}\ r = \mathbf{res}\ (\underline{\langle} h^{\text{ret}}, h^{\text{ops}}, \_\underline{\rangle} :: \cdots :: q_1 :: [])\ \mathbf{in}\ N \rightsquigarrow \\ \quad N[\lambda\langle x, p\rangle\ k.\mathbf{app}\ (q_1 :: \cdots :: \underline{\langle} h^{\text{ret}}, h^{\text{ops}}, p\underline{\rangle} :: k)\ x/r] \end{array}$$

Computations

$$\begin{aligned}
[\![\mathbf{do}\ \ell\ V]\!] &= \overline{\lambda}\,\langle\theta, \overline{\langle}\chi^{\mathrm{ret}}, \chi^{\mathrm{ops}}, \xi\overline{\rangle} :: \kappa.\\
&\qquad \downarrow\chi^{\mathrm{ops}}\ \underline{@}\ \langle\ell\ \underline{\langle}[\![V]\!], \underline{\langle}\downarrow\theta, \underline{\langle}\downarrow\chi^{\mathrm{ret}}, \downarrow\chi^{\mathrm{ops}}, \downarrow\xi\underline{\rangle} :: []\underline{\rangle}, \downarrow\xi\underline{\rangle}\ \underline{@}\ \downarrow\kappa\\
[\![\mathbf{handle}^{\ddagger}\ M\ \mathbf{with}\ (q.\ H)(W)]\!] &= \overline{\lambda}\,\kappa.[\![M]\!]\ \overline{@}\ \overline{\langle}\uparrow[], [\![(q.\ H)(W)]\!]^{\ddagger}\overline{\rangle} :: \kappa\\
[\![(q.\ H)(W)]\!]^{\ddagger} &= \overline{\langle}\uparrow[\![H^{\mathrm{ret}}]\!]^{\ddagger}_{q}, \uparrow[\![H^{\mathrm{ops}}]\!]^{\ddagger}_{q}, \uparrow[\![W]\!]\overline{\rangle}\\
[\![\{\mathbf{return}\ x \mapsto N\}]\!]^{\ddagger}_{q} &= \underline{\lambda}\,\underline{\langle}x, q\underline{\rangle}k.\mathbf{let}\,\underline{\langle}fs, \underline{\langle}h^{\mathrm{ret}}, h^{\mathrm{ops}}, q\underline{\rangle} :: k' = k\ \mathbf{in}\\
&\qquad [\![N]\!]\ \overline{@}\ (\overline{\langle}\uparrow fs, \overline{\langle}\uparrow h^{\mathrm{ret}}, \uparrow h^{\mathrm{ops}}, \uparrow q\overline{\rangle}\overline{\rangle} :: \uparrow k')\\
[\![\{(\ell\ p\ r \mapsto N_{\ell})_{\ell\in\mathscr{L}}\}]\!]^{\ddagger}_{q} &= \underline{\lambda}\,\underline{\langle}x, q\underline{\rangle}\,k.\mathbf{let}\ \underline{\langle}z, \underline{\langle}p, rk\underline{\rangle}\underline{\rangle} = x\ \mathbf{in}\\
&\qquad \mathbf{case}\ z\ \{(\ell \mapsto \mathbf{let}\,\underline{\langle}fs, \underline{\langle}h^{\mathrm{ret}}, h^{\mathrm{ops}}, q\underline{\rangle}\underline{\rangle} :: k' = k\ \mathbf{in}\\
&\qquad\qquad \mathbf{let}\ r = \mathbf{res}^{\ddagger}\ rk\ \mathbf{in}\\
&\qquad\qquad [\![N_{\ell}]\!]\ \overline{@}\ (\overline{\langle}\uparrow fs, \overline{\langle}\uparrow h^{\mathrm{ret}}, \uparrow h^{\mathrm{ops}}, \uparrow q\overline{\rangle}\overline{\rangle} :: \uparrow k'))_{\ell\in\mathscr{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}h^{\mathrm{ret}}, h^{\mathrm{ops}}, q\underline{\rangle}\underline{\rangle} :: k' = k\ \mathbf{in}\\
&\qquad h^{\mathrm{ops}}\ \underline{@}\ \overline{\langle}y\ \underline{\langle}p, \underline{\langle}s', \underline{\langle}h^{\mathrm{ret}}, h^{\mathrm{ops}}, q\underline{\rangle}\underline{\rangle} :: rk\underline{\rangle}, q\underline{\rangle}\ \underline{@}\ k'
\end{aligned}$$

Top-level program

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

Fig. 16. CPS Translation for Parameterised Handlers

Thus, the parameter of the top-most handler in the resumption stack is ignored and replaced by a new value p. The translation is updated accordingly to account for the triple structure. This involves updating all the parts that previously dynamically decomposed and statically recomposed frames to now include the additional state parameter. The key updated translation clauses are shown in Figure 16. The translation of **do** invokes the effect continuation $\downarrow\chi^{\mathrm{ops}}$ with a pair consisting of the operation and the value of the handler parameter. The parameter is also pushed onto the reversed resumption stack. This is necessary to account for the case where the effect continuation $\downarrow\chi^{\mathrm{ops}}$ does not handle operation ℓ.

The translation of the return and operation clauses yields functions that take a pair as input in addition to the current continuation. The forwarding case is adjusted in much the same way as the translation for **do**. The current continuation k is destructed in order to identify the next effect continuation h^{ops} and its parameter q. Then h^{ops} is invoked with the updated resumption stack and the value of its parameter q.

The amended CPS translation for parameterised handlers is not a zero cost translation for shallow and ordinary deep handlers as they will have to thread a "dummy" parameter value through. In contrast, the abstract machine implementation of parameterised handlers does not impose an overhead on shallow and deep handlers.

Abstract Machine Semantics The abstract machine requires two modest changes to accommodate parameterised handlers. The handler installation transition rule M-HANDLE now binds the parameter in the closure environment.

$$\langle\mathbf{handle}^{\ddagger}\ M\ \mathbf{with}\ (q.\ H)(W) \mid \gamma \mid \kappa\rangle \longrightarrow \langle M \mid \gamma \mid ([], (\gamma[q \mapsto [\![W]\!]\gamma], H)) :: \kappa\rangle$$

The parameter q is bound to the interpretation of its initial value W. Otherwise there are no differences from installing an ordinary deep handler.

The resumption application rule M-AppCont is adapted to update the value of parameter in the handler closure environment.

$$\langle V\ \langle V', W'\rangle \mid \gamma \mid \kappa\rangle \longrightarrow \langle \mathbf{return}\ V' \mid \gamma \mid (\kappa' +\!\!+ [(\sigma, (\gamma'[q \mapsto [\![W']\!]\gamma], H))]) +\!\!+ \kappa\rangle, \quad \text{if } [\![V]\!]\gamma = (\kappa' +\!\!+ [(\sigma, (\gamma', H))])^A$$

Besides the environment update, the rule is the same as for ordinary deep handlers. In contrast to the CPS translation, the extension of the abstract machine does implement parameterised handlers as a zero-cost abstraction. This is because the abstract machine has intensional access to the environments that a CPS translation does not.

7.3 Parameterised Handlers as Ordinary Deep Handlers

As mentioned in Section 2.5, parameterised handlers codify the parameter-passing idiom. They may be seen as an optimised form of parameter-passing deep handlers. We now show formally that parameterised handlers are special instances of ordinary deep handlers. We define a local transformation $\mathscr{P}[\![-]\!]$ which translates parameterised handlers into ordinary deep handlers. We omit the homomorphic cases and show only the interesting cases.

$$\begin{aligned}
\mathscr{P}[\![\langle C, A\rangle \Rightarrow^{\ddagger} B!E]\!] &= \mathscr{P}[\![C]\!] \Rightarrow (\mathscr{P}[\![A]\!] \to \mathscr{P}[\![B!E]\!])!\mathscr{P}[\![E]\!] \\
\mathscr{P}[\![\mathbf{handle}^{\ddagger}\ M\ \mathbf{with}\ (q.\ H)(W)]\!] &= (\mathbf{handle}\ \mathscr{P}[\![M]\!]\ \mathbf{with}\ \mathscr{P}[\![H]\!]_q)\ \ \mathscr{P}[\![W]\!] \\
\mathscr{P}[\![\{\mathbf{return}\ x \mapsto M\}]\!]_q &= \{\mathbf{return}\ x \mapsto \lambda q.\mathscr{P}[\![M]\!]\} \\
\mathscr{P}[\![\{\ell\ p\ r \mapsto M\}]\!]_q &= \left\{\begin{array}{c}\ell\ p\ r \mapsto \lambda q.\mathbf{let}\ r' \leftarrow \mathbf{return}\ \lambda\langle x, q'\rangle.r\ x\ q' \\ \mathbf{in}\ \mathscr{P}[\![M]\!][r'/r]\end{array}\right\}
\end{aligned}$$

The parameterised $\mathbf{handle}^{\ddagger}$ construct becomes an application of a **handle** construct to the translation of the parameter. The bodies of return and operation clauses are each enclosed in a lambda abstraction whose formal parameter is the handler parameter q. As a result the ordinary deep resumption r is a curried function. However, the uses of r in M expects a binary function. To repair this discrepancy, we construct an uncurried interface of r via the function r'.

This translation of parameterised handlers simulates the native semantics. As with the simulation of deep handlers via shallow handlers in Section 4.1, this simulation is only up to congruence due to the need for an application of a pure function to a variable to be reduced. The interesting cases of the proof appear in Appendix B.

Theorem 9 (*Simulation of Parameterised Handlers by Deep Handlers*)
If $M \rightsquigarrow N$ then $\mathscr{P}[\![M]\!] \rightsquigarrow^{+}_{\mathrm{cong}} \mathscr{P}[\![N]\!]$.

8 Implementation

In this section we briefly discuss our experiences with using the CPS transforms from Section 5 and the abstract machine from Section 6 as implementation techniques in practice. Our implementation in Links (available at `https://github.com/links-lang/links`) relies on both the higher-order CPS translation and abstract machine.

We retrofitted Links with effect handlers by leveraging most of the infrastructure that was already in place. Server-side code is interpreted by an abstract machine. The existing abstract machine was reminiscent of the CEK machine by Felleisen & Friedman (1986). By

modest effort we were able to generalise the machine's notion of continuation to support handlers. The new abstract machine is in fact parameterised by the notion of continuation, allowing us to switch effect handlers on and off as a language extension.

On the client-side, Links has long used a CPS translation to JavaScript, relying on a trampoline for supporting lightweight concurrency and responsive user-interfaces. The trampoline periodically discards the call stack, therefore it is essential that the CPS translation is 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 track and successfully implemented a properly tail-recursive higher-order uncurried translation along the lines of the one described in Section 5.4. Our implementation selectively CPS transform terms based on their purity, which is crucial in practice for performance. The CPS compiler is also parameterised by the notion of continuation in order to support toggling of handlers.

Our implementation also supports n-ary parameterised handlers. Initially, we desugared parameterised handlers into ordinary deep handlers using a source-to-source translation along the lines of the translation presented in Section 7.3. Now, we provide native support for them in the abstract machine and CPS transform. Anecdotally, we have observed a performance boost when handling stateful computations using parameterised handlers.

9 Related Work

Interdefinability of Control Operators In Section 4 we confirmed the folklore that deep and shallow handlers are interdefinable. This result is similar in spirit to the result of Shan (2007) in the setting of delimited continuations. Shan demonstrated that static delimited control can simulate dynamic delimited control. Our results are obtained using different techniques; Shan uses a CPS transform with recursive delimited continuation, whilst we use a variation of the technique of Fokkinga (1990) to implement a mutumorphism (shallow handler) using a tuple and a catamorphism (deep handler).

CPS Translations for Handlers We draw on insights from literature on CPS translations for delimited control operators such as shift and reset (Danvy & Filinski, 1990, 1992; Danvy & Nielsen, 2003; Materzok & Biernacki, 2012) to devise our one-pass higher-order CPS translation for deep and shallow effect handlers. Other CPS translations for handlers use a monadic approach. For example, Leijen (2017c) implements deep and parameterised handlers in Koka (Leijen, 2014) by translating them into a free monad primitive in the runtime. Leijen uses a selective CPS translation to lift code into the monad. The selective aspect is important in practice to avoid overhead in code that does not use effect handlers. Scala Effekt (Brachthäuser & Schuster, 2017; Brachthäuser *et al.*, 2020) provide an implementation of effect handlers as a library for the Scala programming language. The implementation is based closely on the monadic delimited control framework of Dybvig *et al.* (2007). A variation of the Scala Effekt library is used to implement effect handlers as an interface for programming with delimited continuations

in Java (Brachthäuser *et al.*, 2018). The implementation of delimited continuations depend on special byte code instructions, inserted via a selective type-driven CPS translation.

There are clear connections between our CPS translations and the continuation monad implementation of Kammar *et al.* (2013). Whereas Kammar *et al.* present a practical Haskell implementation depending on sophisticated features such as type classes, which to some degree obscures the essential structure, here we have focused on a foundational formal treatment. Kammar *et al.* obtain impressive performance results by taking advantage of the second class nature of type classes in Haskell coupled with the aggressive fusion optimisations GHC performs (Wu & Schrijvers, 2015).

Abstract Machines for Handlers Biernacki *et al.* (2018) make use of a variation of our abstract machine, which they later refined to be used as the runtime basis for the Helium language (Biernacki *et al.*, 2019). Their machine supports only deep handlers.

The Frank language (Lindley *et al.*, 2017; Convent *et al.*, 2020) compiles to a dynamically typed variation of itself called Shonky (McBride, 2016). Shonky uses an abstract machine as a basis for its runtime. In contrast to our machine, the Shonky machine uses a flat continuation structure with syntactic markers to delimit the extent of handlers.

Stacks for Handlers Leijen (2017a) implements effect handlers in C as a library using a stack copying technique similar to the technique used by Kiselyov (2012) to implement multi-prompt continuations and shift and reset in OCaml as a library. The multicore extension to OCaml (Dolan *et al.*, 2014) adds native support for effect handlers in OCaml. Handlers are intended as the primary means for implementing and structuring concurrency (Dolan *et al.*, 2015). Their implementation uses a variation of segmented stacks (Bruggeman *et al.*, 1996) to implement efficient one-shot resumptions. Multi-shot continuations can be simulated by using an explicit continuation cloning primitive. The cloning primitive has been used by both Hillerström *et al.* (2016) and Kiselyov & Sivaramakrishnan (2016). Hillerström *et al.* used it to implement multi-shot semantics for effect handlers in their prototype compiler for Links which targets the intermediate Lambda layer of the Multicore OCaml compiler. Kiselyov & Sivaramakrishnan (2016) used it to embed the Eff language (Bauer & Pretnar, 2015) directly in Multicore OCaml.

10 Conclusions and Future Work

In this paper we have examined effect handlers and their implementation in depth. We have explored the design space of handlers: deep, shallow, and parameterised, and formally and informally compared their expressiveness. We have also given two formally presented implementation strategies for all the different kinds effect handlers: a CPS translation into a calculus without effect handlers, and an abstract machine designed explicitly for executing languages with effect handlers. This is the first full CPS translation for effect handlers: our translations go all the way to lambda calculus without relying on a special low-level handling construct as Leijen (2017c) does. As well as the formal development, we have also demonstrated the practicality of our implementation techniques by implementing them in the Links programming language.

A key finding of this work is the structure of generalised continuations needed to correctly implement effect handlers in both the CPS translation and the abstract machine. For us, this turned out to be surprisingly subtle, and we had several broken designs before arriving at the ones presented here. The presence of an implementation in the Links interpreter/compiler made the process of discovering buggy translations much easier.

Our formal translations between the different kinds of handlers shows that they are, in a sense, equally expressive, and one might consider a language with only deep handlers as this most closely matches Plotkin and Pretnar's original vision. However, as we demonstrated in Section 2, shallow handlers allow more natural programming for some problems. Moreover, parameterised handlers appear to offer an efficiency boost for a common idiomatic use of deep handlers. Comprehensive benchmarking is still required to prove this conclusively though.

For future work, we wish to further explore the range of possible idioms of effect handlers and their implementations. Of particular interest are handlers that use their resumptions *linearly* (exactly once), or *affine linearly* (at most once). The latter restriction is enforced dynamically in Multicore OCaml (Dolan *et al.*, 2017) because it allows an implementation that does not have to copy runtime stacks for multiple resumptions after an operation. Extending the abstract machine we presented in Section 6 to accurately model linearly used continuations seems feasible, as does attempting to linearly type the CPS translation of Section 5, following the linearly used continuation passing style described by Berdine *et al.* (2002).

We would also like to make our CPS translation typed, so that the type safety guarantees of the source language are carried through automatically to the translation. The appendix of Hillerström *et al.* (2017) sketches a type preserving CPS translation for deep handlers, but it remains to extend this to shallow and parameterised handlers.

Finally, there is now a wide diversity of implementation strategies for a wide variety of styles of effect handlers. We plan to perform comprehensive benchmarking of competing implementation strategies, especially in the setting of "advanced control abstraction hostile" environments, such JavaScript.

Acknowledgements We thank Nicolas Oury for originally suggesting the Nim game as an example to demonstrate programming with handlers. We thank John Longley for insightful discussions about the inter-encodings of deep and shallow handlers. We thank KC Sivaramakrishnan for his generosity. We thank Simon Fowler, Ohad Kammar, James McKinna, Craig McLaughlin, Gabriel Scherer, and the anonymous reviewers for helpful feedback, suggestions, and discussions. This work was supported by EPSRC grants EP/L01503X/1 (EPSRC Centre for Doctoral Training in Pervasive Parallelism) and EP/K034413/1 (From Data Types to Session Types—A Basis for Concurrency and Distribution).

Bibliography

Appel, Andrew W. (1992). *Compiling with continuations*. Cambridge University Press.

Bauer, Andrej, & Pretnar, Matija. (2015). Programming with algebraic effects and handlers. *J. log. algebr. meth. program.*, **84**(1), 108–123.

Berdine, Josh, O'Hearn, Peter W., Reddy, Uday S., & Thielecke, Hayo. (2002). Linear continuation-passing. *Higher-order and symbolic computation*, **15**(2-3), 181–208.

Biernacki, Dariusz, Piróg, Maciej, Polesiuk, Piotr, & Sieczkowski, Filip. (2018). Handle with care: relational interpretation of algebraic effects and handlers. *PACMPL*, **2**(POPL), 8:1–8:30.

Biernacki, Dariusz, Piróg, Maciej, Polesiuk, Piotr, & Sieczkowski, Filip. (2019). Abstracting algebraic effects. *PACMPL*, **3**(POPL), 6:1–6:28.

Bingham, Eli, Chen, Jonathan P., Jankowiak, Martin, Obermeyer, Fritz, Pradhan, Neeraj, Karaletsos, Theofanis, Singh, Rohit, Szerlip, Paul, Horsfall, Paul, & Goodman, Noah D. (2018). Pyro: Deep Universal Probabilistic Programming. *Journal of machine learning research*.

Bouton, Charles L. (1901). Nim, a game with a complete mathematical theory. *Annals of mathematics*, **3**(1/4), 35–39.

Brachthäuser, Jonathan Immanuel, & Schuster, Philipp. (2017). Effekt: extensible algebraic effects in scala (short paper). *Pages 67–72 of: SCALA@SPLASH*. ACM.

Brachthäuser, Jonathan Immanuel, Schuster, Philipp, & Ostermann, Klaus. (2018). Effect handlers for the masses. *PACMPL*, **2**(OOPSLA), 111:1–111:27.

Brachthäuser, Jonathan Immanuel, Schuster, Philipp, & Ostermann, Klaus. (2020). Effekt: Capability-passing style for type- and effect-safe, extensible effect handlers in Scala. *J. funct. program.*, **30**. To appear.

Bruggeman, Carl, Waddell, Oscar, & Dybvig, R. Kent. (1996). Representing control in the presence of one-shot continuations. *Pages 99–107 of:* Fischer, Charles N. (ed), *Proceedings of the ACM SIGPLAN'96 Conference on Programming Language Design and Implementation (PLDI), Philadephia, Pennsylvania, May 21-24, 1996*. ACM.

Convent, Lukas, Lindley, Sam, McBride, Conor, & McLaughlin, Craig. (2020). Doo bee doo bee doo. *J. funct. program.*, **30**. To appear.

Cooper, Ezra, Lindley, Sam, Wadler, Philip, & Yallop, Jeremy. (2006). Links: Web programming without tiers. *Pages 266–296 of: FMCO*. LNCS, vol. 4709. Springer.

Danvy, Olivier, & Filinski, Andrzej. (1990). Abstracting control. *Pages 151–160 of: LISP and Functional Programming*.

Danvy, Olivier, & Filinski, Andrzej. (1992). Representing control: A study of the CPS transformation. *Mathematical structures in computer science*, **2**(4), 361–391.

Danvy, Olivier, & Nielsen, Lasse R. (2003). A first-order one-pass CPS transformation. *Theor. comput. sci.*, **308**(1-3), 239–257.

Dolan, Stephen, White, Leo, & Madhavapeddy, Anil. (2014). *Multicore OCaml*. OCaml Workshop.

Dolan, Stephen, White, Leo, Sivaramakrishnan, KC, Yallop, Jeremy, & Madhavapeddy, Anil. (2015). *Effective concurrency through algebraic effects*. OCaml Workshop.

Dolan, Stephen, Eliopoulos, Spiros, Hillerström, Daniel, Madhavapeddy, Anil, Sivaramakrishnan, KC, & White, Leo. (2017). Concurrent system programming with effect handlers. *Pages 98–117 of: TFP*. Lecture Notes in Computer Science, vol. 10788. Springer.

Dybvig, R. Kent, Peyton Jones, Simon L., & Sabry, Amr. (2007). A monadic framework for delimited continuations. *J. funct. program.*, **17**(6), 687–730.

Felleisen, Matthias, & Friedman, Daniel P. (1986). Control operators, the SECD-machine, and the λ-calculus. *Pages 193–217 of: Formal Description of Programming Concepts III*. Elsevier.

Flanagan, Cormac, Sabry, Amr, Duba, Bruce F., & Felleisen, Matthias. (1993). The essence of compiling with continuations. *Pages 237–247 of: PLDI*. ACM.

Fokkinga, M. M. (1990). Tupling and mutumorphisms. *The squiggolist*, **1**(4), 81–82.

Forster, Yannick, Kammar, Ohad, Lindley, Sam, & Pretnar, Matija. (2017). On the expressive power of user-defined effects: Effect handlers, monadic reflection, delimited control. *PACMPL*, **1**(ICFP).

Forster, Yannick, Kammar, Ohad, Lindley, Sam, & Pretnar, Matija. (2019). On the expressive power of user-defined effects: Effect handlers, monadic reflection, delimited control. *J. funct. program.*, **29**, e15.

Hillerström, Daniel, & Lindley, Sam. (2016). Liberating effects with rows and handlers. *Pages 15–27 of: TyDe@ICFP*. ACM.

Hillerström, Daniel, & Lindley, Sam. (2018). Shallow effect handlers. *Pages 415–433 of: APLAS*, vol. 11275. Springer International Publishing.

Hillerström, Daniel, Lindley, Sam, Atkey, Robert, & Sivaramakrishnan, KC. (2017). Continuation passing style for effect handlers. *Pages 18:1–18:19 of: FSCD*. LIPIcs, vol. 84. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik.

Hillerström, Daniel. (2015). *Handlers for algebraic effects in Links*. MSc thesis, The University of Edinburgh, Scotland.

Hillerström, Daniel. (2016). *Compilation of effect handlers and their applications in concurrency*. MSc(R) thesis, The University of Edinburgh, Scotland.

Hillerström, Daniel, Lindley, Sam, & Sivaramakrishnan, KC. (2016). *Compiling Links effect handlers to the OCaml backend*. ML Workshop.

Huet, Gérard P. (1997). The zipper. *J. funct. program.*, **7**(5), 549–554.

James, Roshan P, & Sabry, Amr. (2011). Yield: Mainstream delimited continuations. *TPDC*.

Kammar, Ohad, Lindley, Sam, & Oury, Nicolas. (2013). Handlers in action. *Pages 145–158 of: ICFP*. ACM.

Kennedy, Andrew. (2007). Compiling with continuations, continued. *Pages 177–190 of: ICFP*. ACM.

Kiselyov, Oleg. (2012). Delimited control in OCaml, abstractly and concretely. *Theor. comput. sci.*, **435**, 56–76.

Kiselyov, Oleg, & Sivaramakrishnan, KC. (2016). *Eff directly in OCaml*. ML Workshop.

Kranz, D., Kelsey, R., Rees, J., Hudak, P., Philbin, J., & Adams, N. (1986). ORBIT: An optimizing compiler for Scheme. **21**(7), 219–233. Proc. Sigplan '86 Symp. on Compiler Construction.

Leijen, Daan. (2014). Koka: Programming with row polymorphic effect types. *Pages 100–126 of: MSFP*. EPTCS, vol. 153.

Leijen, Daan. (2017a). Implementing algebraic effects in C - "monads for free in C". *Pages 339–363 of: APLAS*. Lecture Notes in Computer Science, vol. 10695. Springer.

Leijen, Daan. (2017b). Structured asynchrony with algebraic effects. *Pages 16–29 of: TyDe@ICFP*. ACM.

Leijen, Daan. (2017c). Type directed compilation of row-typed algebraic effects. *Pages 486–499 of: POPL*. ACM.

Levy, Paul Blain, Power, John, & Thielecke, Hayo. (2003). Modelling environments in call-by-value programming languages. *Inf. comput.*, **185**(2), 182–210.

Lindley, Sam, McBride, Conor, & McLaughlin, Craig. (2017). Do be do be do. *Pages 500–514 of: POPL*. ACM.

Materzok, Marek, & Biernacki, Dariusz. (2012). A dynamic interpretation of the CPS hierarchy. *Pages 296–311 of: APLAS*. LNCS, vol. 7705. Springer.

McBride, Conor. (2016). *Shonky*. `https://github.com/pigworker/shonky`.

Meijer, Erik, Fokkinga, Maarten M., & Paterson, Ross. (1991). Functional programming with bananas, lenses, envelopes and barbed wire. *Pages 124–144 of: FPCA*. Lecture Notes in Computer Science, vol. 523. Springer.

Piróg, Maciej, Polesiuk, Piotr, & Sieczkowski, Filip. (2019). Typed equivalence of effect handlers and delimited control. *Pages 30:1–30:16 of: FSCD*. LIPIcs, vol. 131. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik.

Plotkin, Gordon D. (1975). Call-by-name, call-by-value and the lambda-calculus. *Theor. comput. sci.*, **1**(2), 125–159.

Plotkin, Gordon D., & Power, John. (2001). Adequacy for algebraic effects. *Pages 1–24 of: FoSSaCS*. LNCS, vol. 2030. Springer.

Plotkin, Gordon D., & Pretnar, Matija. (2013). Handling algebraic effects. *Logical methods in computer science*, **9**(4).

Pretnar, Matija. (2015). An introduction to algebraic effects and handlers. *Electr. notes theor. comput. sci.*, **319**, 19–35. Invited tutorial paper.

Remy, Didier. (1993). *Syntactic theories and the algebra of record terms*. Tech. rept. RR-1869. INRIA.

Shan, Chung-chieh. (2007). A static simulation of dynamic delimited control. *Higher-order and symbolic computation*, **20**(4), 371–401.

Steele, Guy. (1978). *RABBIT: A compiler for SCHEME*. Tech. rept. AITR-474. INRIA.

Wadler, Philip. (1995). Monads for functional programming. *Pages 24–52 of: Advanced Functional Programming*. Lecture Notes in Computer Science, vol. 925. Springer.

Wu, Nicolas, & Schrijvers, Tom. (2015). Fusion for free - efficient algebraic effect handlers. *Pages 302–322 of: MPC*. Lecture Notes in Computer Science, vol. 9129. Springer.

Wu, Nicolas, Schrijvers, Tom, & Hinze, Ralf. (2014). Effect handlers in scope. *Pages 1–12 of:* Swierstra, Wouter (ed), *Proceedings of the 2014 ACM SIGPLAN symposium on Haskell, Gothenburg, Sweden, September 4-5, 2014*. ACM.

Yallop, Jeremy. (2017). Staged generic programming. *PACMPL*, **1**(ICFP), 29:1–29:29.

A Proofs of Correctness of the Higher-Order Uncurried CPS Translation

Lemma 2 (*Substitution*)
The CPS translation $[\![-]\!]$ commutes with substitution in value terms

$$[\![W]\!][[\![V]\!]/x] = [\![W[V/x]]\!],$$

and with substitution in computation terms

$$\begin{aligned}&([\![M]\!]\mathbin{\overline{@}}(\overline{\langle}\mathscr{V}_{fs},\overline{\langle}\mathscr{V}_{ret},\mathscr{V}_{ops}\overline{\rangle}\rangle\mathbin{::}\mathscr{W}))[[\![V]\!]/x]\\ =\ &[\![M[V/x]]\!]\mathbin{\overline{@}}(\overline{\langle}\mathscr{V}_{fs},\overline{\langle}\mathscr{V}_{ret},\mathscr{V}_{ops}\overline{\rangle}\rangle\mathbin{::}\mathscr{W})[[\![V]\!]/x].\end{aligned}$$

Proof
The proof is by mutual induction on the structure of the computation term M and the value term W. For most of the cases, the existence of the top level frame on the stack is not important, so we just refer to the whole static continuation stack as $\mathscr{W}$. Note that we make implicit use of the fact that the parts of the continuation stack that are statically known are all of the form of right nested triples of reflected dynamic terms.

Case $M = V'\,W$.

$$\begin{aligned}&([\![V'\,W]\!]\mathbin{\overline{@}}\mathscr{W})[[\![V]\!]/x]\\ =\ &\quad\text{(definition of } [\![-]\!])\\ &((\overline{\lambda}\kappa.[\![V']\!]\mathbin{\underline{@}}[\![W]\!]\mathbin{\underline{@}}{\downarrow}\kappa)\mathbin{\overline{@}}\mathscr{W})[[\![V]\!]/x]\\ =\ &\quad\text{(static } \beta\text{-conversion)}\\ &([\![V']\!]\mathbin{\underline{@}}[\![W]\!]\mathbin{\underline{@}}{\downarrow}\mathscr{W})[[\![V]\!]/x]\\ =\ &\quad\text{(definition of } [-])\\ &([\![V']\!][[\![V]\!]/x])\mathbin{\underline{@}}([\![W]\!][[\![V]\!]/x])\mathbin{\underline{@}}{\downarrow}\mathscr{W}[[\![V]\!]/x]\\ =\ &\quad\text{(IH 2, twice)}\\ &[\![V'[V/x]]\!]\mathbin{\underline{@}}[\![W[V/x]]\!]\mathbin{\underline{@}}{\downarrow}\mathscr{W}[[\![V]\!]/x]\\ =\ &\quad\text{(static } \beta\text{-conversion)}\\ &(\overline{\lambda}\kappa.[\![V'[V/x]]\!]\mathbin{\underline{@}}[\![W[V/x]]\!]\mathbin{\underline{@}}{\downarrow}\kappa)\mathbin{\overline{@}}\mathscr{W}[[\![V]\!]/x]\\ =\ &\quad\text{(definition of } [\![-]\!])\\ &[\![(V'[V/x])\,(W[V/x])]\!]\mathbin{\overline{@}}\mathscr{W}[[\![V]\!]/x]\\ =\ &\quad\text{(definition of } [-])\\ &[\![(V'\,W)[V/x]]\!]\mathbin{\overline{@}}\mathscr{W}[[\![V]\!]/x]\end{aligned}$$

Case $M = W\,T$.

$$\begin{aligned}&([\![W\,T]\!]\mathbin{\overline{@}}\mathscr{W})[[\![V]\!]/x]\\ =\ &\quad\text{(definition of } [\![-]\!])\\ &((\overline{\lambda}\kappa.[\![W]\!]\mathbin{\underline{@}}\langle\rangle\mathbin{\underline{@}}{\downarrow}\kappa)\mathbin{\overline{@}}\mathscr{W})[[\![V]\!]/x]\\ =\ &\quad\text{(static } \beta\text{-conversion)}\\ &([\![W]\!]\mathbin{\underline{@}}\langle\rangle\mathbin{\underline{@}}{\downarrow}\mathscr{W})[[\![V]\!]/x]\\ =\ &\quad\text{(definition of } [-])\\ &[\![W]\!][[\![V]\!]/x]\mathbin{\underline{@}}\langle\rangle\mathbin{\underline{@}}{\downarrow}\mathscr{W}[[\![V]\!]/x]\\ =\ &\quad\text{(IH 2)}\\ &[\![W[V/x]]\!]\mathbin{\underline{@}}\langle\rangle\mathbin{\underline{@}}{\downarrow}\mathscr{W}[[\![V]\!]/x]\end{aligned}$$

$$
\begin{array}{ll}
= & \text{(static } \beta\text{-conversion)} \\
& (\overline{\lambda}\,\kappa.[\![W[V/x]]\!] \mathbin{\overline{@}} \langle\rangle \mathbin{\overline{@}} \downarrow\kappa) \mathbin{\overline{@}} \mathscr{W}[[\![V]\!]/x] \\
= & \text{(definition of } [\![-]\!]) \\
& [\![W[V/x]\,T]\!] \mathbin{\overline{@}} \mathscr{W}[[\![V]\!]/x] \\
= & \text{(definition of } [-]) \\
& [\![(W\,T)[V/x]]\!] \mathbin{\overline{@}} \mathscr{W}[[\![V]\!]/x]
\end{array}
$$

Case $M = \mathbf{let}\ \langle \ell = x'; y\rangle = W\ \mathbf{in}\ N$.

$$
\begin{array}{ll}
& ([\![\mathbf{let}\ \langle \ell = x'; y\rangle = W\ \mathbf{in}\ N]\!] \mathbin{\overline{@}} \mathscr{W})[[\![V]\!]/x] \\
= & \text{(definition of } [\![-]\!]) \\
& ((\overline{\lambda}\,\kappa.\mathbf{let}\ \langle \ell = x'; y\rangle = [\![W]\!]\ \mathbf{in}\ [\![N]\!] \mathbin{\overline{@}} \kappa) \mathbin{\overline{@}} \mathscr{W})[[\![V]\!]/x] \\
= & \text{(static } \beta\text{-conversion)} \\
& (\mathbf{let}\ \langle \ell = x'; y\rangle = [\![W]\!]\ \mathbf{in}\ [\![N]\!] \mathbin{\overline{@}} \mathscr{W})[[\![V]\!]/x] \\
= & \text{(definition of } [-]) \\
& \mathbf{let}\ \langle \ell = x'; y\rangle = [\![W]\!][[\![V]\!]/x]\ \mathbf{in}\ ([\![N]\!] \mathbin{\overline{@}} \mathscr{W})[[\![V]\!]/x] \\
= & \text{(IH 1 and IH 2)} \\
& \mathbf{let}\ \langle \ell = x'; y\rangle = [\![W[V/x]]\!]\ \mathbf{in}\ [\![N[V/x]]\!] \mathbin{\overline{@}} \mathscr{W}[[\![V]\!]/x] \\
= & \text{(static } \beta\text{-conversion)} \\
& (\overline{\lambda}\,\kappa.\mathbf{let}\ \langle \ell = x'; y\rangle = [\![W[V/x]]\!]\ \mathbf{in}\ [\![N[V/x]]\!] \mathbin{\overline{@}} \kappa) \mathbin{\overline{@}} \mathscr{W}[[\![V]\!]/x] \\
= & \text{(definition of } [\![-]\!]) \\
& [\![\mathbf{let}\ \langle \ell = x'; y\rangle = W[V/x]\ \mathbf{in}\ N[V/x]]\!] \mathbin{\overline{@}} \mathscr{W}[[\![V]\!]/x] \\
= & \text{(definition of } [-]) \\
& [\![(\mathbf{let}\ \langle \ell = x'; y\rangle = W\ \mathbf{in}\ N)[V/x]]\!] \mathbin{\overline{@}} \mathscr{W}[[\![V]\!]/x]
\end{array}
$$

Case $M = \mathbf{case}\ V\ \{\ell\, x \mapsto M; y \mapsto N\}$. Similar to the $M = \mathbf{let}\ \langle \ell = x; y\rangle = V\ \mathbf{in}\ N$ case.

Case $M = \mathbf{absurd}\ W$.

$$
\begin{array}{ll}
& ([\![\mathbf{absurd}\ W]\!] \mathbin{\overline{@}} \mathscr{W})[[\![V]\!]/x] \\
= & \text{(definition of } [\![-]\!]) \\
& ((\overline{\lambda}\,\kappa.\mathbf{absurd}\ W) \mathbin{\overline{@}} \mathscr{W})[[\![V]\!]/x] \\
= & \text{(static } \beta\text{-conversion)} \\
& (\mathbf{absurd}\ [\![W]\!])[[\![V]\!]/x] \\
= & \text{(definition of } [-]) \\
& \mathbf{absurd}\ [\![W]\!][[\![V]\!]/x] \\
= & \text{(IH 2)} \\
& \mathbf{absurd}\ [\![W[V/x]]\!] \\
= & \text{(static } \beta\text{-conversion)} \\
& (\overline{\lambda}\,\kappa.\mathbf{absurd}\ [\![W[V/x]]\!]) \mathbin{\overline{@}} \mathscr{W}[[\![V]\!]/x] \\
= & \text{(definition of } [\![-]\!]) \\
& [\![\mathbf{absurd}\ W[V/x]]\!] \mathbin{\overline{@}} \mathscr{W}[[\![V]\!]/x] \\
= & \text{(definition of } [-]) \\
& [\![(\mathbf{absurd}\ W)[V/x]]\!] \mathbin{\overline{@}} \mathscr{W}[[\![V]\!]/x]
\end{array}
$$

Case $M = \mathbf{return}\ W$.

$$
\begin{array}{ll}
 & (\llbracket \mathbf{return}\ W \rrbracket \mathbin{\overline{@}} \mathscr{W})[\llbracket V \rrbracket/x] \\
= & \quad \text{(definition of } \llbracket - \rrbracket) \\
 & ((\overline{\lambda}\,\kappa.\mathbf{app} \downarrow\kappa\ \llbracket W \rrbracket) \mathbin{\overline{@}} \mathscr{W})[\llbracket V \rrbracket/x] \\
= & \quad \text{(static } \beta\text{-conversion)} \\
 & (\mathbf{app} \downarrow\mathscr{W}\ \llbracket W \rrbracket)[\llbracket V \rrbracket/x] \\
= & \quad \text{(definition of } [-/-]) \\
 & \mathbf{app} \downarrow(\mathscr{W}[\llbracket V \rrbracket/x])\ (\llbracket W \rrbracket[\llbracket V \rrbracket/x]) \\
= & \quad \text{(IH 2)} \\
 & \mathbf{app} \downarrow(\mathscr{W}[\llbracket V \rrbracket/x])\ \llbracket W[V/x] \rrbracket \\
= & \quad \text{(static } \beta\text{-conversion)} \\
 & (\overline{\lambda}\,\kappa.\mathbf{app} \downarrow\kappa\ \llbracket W[V/x] \rrbracket) \mathbin{\overline{@}} \mathscr{W}[\llbracket V \rrbracket/x] \\
= & \quad \text{(definition of } \llbracket - \rrbracket) \\
 & \llbracket \mathbf{return}\ (W[V/x]) \rrbracket \mathbin{\overline{@}} \mathscr{W}[\llbracket V \rrbracket/x] \\
= & \quad \text{(definition of } [-/-]) \\
 & \llbracket (\mathbf{return}\ W)[V/x] \rrbracket \mathbin{\overline{@}} \mathscr{W}[\llbracket V \rrbracket/x]
\end{array}
$$

Case $M = \mathbf{let}\ y \leftarrow M'\ \mathbf{in}\ N$. We have:

$$
\begin{array}{ll}
 & (\llbracket \mathbf{let}\ y \leftarrow M'\ \mathbf{in}\ N \rrbracket \mathbin{\overline{@}} (\overline{\langle} \mathscr{V}_{fs}, \overline{\langle} \mathscr{V}_{ret}, \mathscr{V}_{ops} \overline{\rangle\rangle} :: \mathscr{W}))[\llbracket V \rrbracket/x] \\
= & \quad \text{(definition of } \llbracket - \rrbracket) \\
 & ((\overline{\lambda}\langle \theta, \overline{\langle} \chi^{\mathrm{ret}}, \chi^{\mathrm{ops}} \overline{\rangle\rangle} :: \kappa. \\
 & \qquad \llbracket M' \rrbracket \mathbin{\overline{@}} (\overline{\langle} \uparrow((\underline{\lambda} y\, k.\, \mathbf{let}\ \underline{\langle} fs, \underline{\langle} h^{\mathrm{ret}}, h^{\mathrm{ops}} \underline{\rangle\rangle} :: k' = k\ \mathbf{in} \\
 & \qquad\qquad \llbracket N \rrbracket \mathbin{\overline{@}} (\overline{\langle} \uparrow fs, \overline{\langle} \uparrow h^{\mathrm{ret}}, \uparrow h^{\mathrm{ops}} \overline{\rangle\rangle} :: \uparrow k')) \mathbin{\underline{::}} \downarrow\theta), \overline{\langle} \chi^{\mathrm{ret}}, \chi^{\mathrm{ops}} \overline{\rangle\rangle} :: \kappa)) \\
 & \quad \mathbin{\overline{@}} (\overline{\langle} \mathscr{V}_{fs}, \overline{\langle} \mathscr{V}_{ret}, \mathscr{V}_{ops} \overline{\rangle\rangle} :: \mathscr{W}))[\llbracket V \rrbracket/x] \\
= & \quad \text{(static } \beta\text{-conversion)} \\
 & (\llbracket M' \rrbracket \mathbin{\overline{@}} (\overline{\langle} \uparrow((\underline{\lambda} y\, k.\, \mathbf{let}\ \underline{\langle} fs, \underline{\langle} h^{\mathrm{ret}}, h^{\mathrm{ops}} \underline{\rangle\rangle} :: k' = k\ \mathbf{in} \\
 & \qquad\qquad \llbracket N \rrbracket \mathbin{\overline{@}} (\overline{\langle} \uparrow fs, \overline{\langle} \uparrow h^{\mathrm{ret}}, \uparrow h^{\mathrm{ops}} \overline{\rangle\rangle} :: \uparrow k')) \mathbin{\underline{::}} \downarrow\mathscr{V}_{fs}), \mathscr{V}_h \overline{\rangle} :: \mathscr{W}'))[\llbracket V \rrbracket/x] \\
= & \quad \text{(IH 1 on } M') \\
 & \llbracket M'[V/x] \rrbracket \mathbin{\overline{@}} ((\overline{\langle} \uparrow((\underline{\lambda} y\, k.\, \mathbf{let}\ \underline{\langle} fs, \underline{\langle} h^{\mathrm{ret}}, h^{\mathrm{ops}} \underline{\rangle\rangle} :: k' = k\ \mathbf{in} \\
 & \qquad\qquad \llbracket N \rrbracket \mathbin{\overline{@}} (\overline{\langle} \uparrow fs, \overline{\langle} \uparrow h^{\mathrm{ret}}, \uparrow h^{\mathrm{ops}} \overline{\rangle\rangle} :: \uparrow k')) \mathbin{\underline{::}} \downarrow\mathscr{V}_{fs}), \mathscr{V}_h \overline{\rangle} :: \mathscr{W}')[\llbracket V \rrbracket/x]) \\
= & \quad \text{(definition of } [-/-]) \\
 & \llbracket M'[V/x] \rrbracket \mathbin{\overline{@}} (\overline{\langle} \uparrow((\underline{\lambda} y\, k.\, \mathbf{let}\ \underline{\langle} fs, \underline{\langle} h^{\mathrm{ret}}, h^{\mathrm{ops}} \underline{\rangle\rangle} :: k' = k\ \mathbf{in} \\
 & \qquad\qquad (\llbracket N \rrbracket \mathbin{\overline{@}} (\overline{\langle} \uparrow fs, \overline{\langle} \uparrow h^{\mathrm{ret}}, \uparrow h^{\mathrm{ops}} \overline{\rangle\rangle} :: \uparrow k'))[\llbracket V \rrbracket/x]) \mathbin{\underline{::}} \downarrow(\mathscr{V}_{fs}[\llbracket V \rrbracket/x])), \\
 & \qquad \mathscr{V}_h[\llbracket V \rrbracket/x] \overline{\rangle} :: (\mathscr{W}'[\llbracket V \rrbracket/x])) \\
= & \quad \text{(IH 1 on } N) \\
 & \llbracket M'[V/x] \rrbracket \mathbin{\overline{@}} (\overline{\langle} \uparrow((\underline{\lambda} y\, k.\, \mathbf{let}\ \underline{\langle} fs, \underline{\langle} h^{\mathrm{ret}}, h^{\mathrm{ops}} \underline{\rangle\rangle} :: k' = k\ \mathbf{in} \\
 & \qquad\qquad \llbracket N[V/x] \rrbracket \mathbin{\overline{@}} (\overline{\langle} \uparrow fs, \overline{\langle} \uparrow h^{\mathrm{ret}}, \uparrow h^{\mathrm{ops}} \overline{\rangle\rangle} :: \uparrow k')) \mathbin{\underline{::}} (\downarrow\mathscr{V}_{fs}[\llbracket V \rrbracket/x])), \\
 & \qquad \mathscr{V}_h[\llbracket V \rrbracket/x] \overline{\rangle} :: (\mathscr{W}'[\llbracket V \rrbracket/x])) \\
= & \quad \text{(static } \beta\text{-conversion and definition of } \llbracket - \rrbracket) \\
 & \llbracket \mathbf{let}\ y \leftarrow M'[V/x]\ \mathbf{in}\ N[V/x] \rrbracket \mathbin{\overline{@}} ((\overline{\langle} \mathscr{V}_{fs}, \overline{\langle} \mathscr{V}_{ret}, \mathscr{V}_{ops} \overline{\rangle\rangle} :: \mathscr{W})[\llbracket V \rrbracket/x]) \\
= & \quad \text{(definition of } [-/-]) \\
 & \llbracket (\mathbf{let}\ y \leftarrow M'\ \mathbf{in}\ N)[V/x] \rrbracket \mathbin{\overline{@}} ((\overline{\langle} \mathscr{V}_{fs}, \overline{\langle} \mathscr{V}_{ret}, \mathscr{V}_{ops} \overline{\rangle\rangle} :: \mathscr{W})[\llbracket V \rrbracket/x])
\end{array}
$$

Case $M = \mathbf{do}\ (\ell\, W)^E$. We have:

$$
\begin{aligned}
& (\llbracket \mathbf{do}\ (\ell\, W)^E \rrbracket \mathbin{\overline{@}} \overline{\langle} \mathscr{V}_{fs}, \overline{\langle} \mathscr{V}_{ret}, \mathscr{V}_{ops} \overline{\rangle\rangle} :: \mathscr{W})[\llbracket V \rrbracket/x] \\
= \quad & \text{(definition of } \llbracket - \rrbracket) \\
& ((\overline{\lambda}\overline{\langle}\theta, \overline{\langle}\chi^{\mathrm{ret}}, \chi^{\mathrm{ops}}\overline{\rangle\rangle} :: \kappa.{\downarrow}\chi^{\mathrm{ops}} \mathbin{\underline{@}} \underline{\langle}\ell, \underline{\langle}\llbracket W \rrbracket, \underline{\langle}{\downarrow}\theta, \underline{\langle}{\downarrow}\chi^{\mathrm{ret}}, {\downarrow}\chi^{\mathrm{ops}}\underline{\rangle\rangle} :: \underline{[]\rangle\rangle} \mathbin{\underline{@}} {\downarrow}\kappa) \\
& \quad \mathbin{\overline{@}} (\overline{\langle} \mathscr{V}_{fs}, \overline{\langle} \mathscr{V}_{ret}, \mathscr{V}_{ops}\overline{\rangle\rangle} :: \mathscr{W}))[\llbracket V \rrbracket/x] \\
= \quad & \text{(static } \beta\text{-conversion)} \\
& ({\downarrow}\mathscr{V}_{ops} \mathbin{\underline{@}} \underline{\langle}\ell, \underline{\langle}\llbracket W \rrbracket, \underline{\langle}{\downarrow}\mathscr{V}_{fs}, \underline{\langle}{\downarrow}\mathscr{V}_{ret}, {\downarrow}\mathscr{V}_{ops}\underline{\rangle\rangle} :: \underline{[]\rangle\rangle} \mathbin{\underline{@}} {\downarrow}\mathscr{W})[\llbracket V \rrbracket/x] \\
= \quad & \text{(definition of } [-/-]) \\
& {\downarrow}\mathscr{V}_{ops}[\llbracket V \rrbracket/x] \underline{@} \underline{\langle}\ell, \underline{\langle}\llbracket W \rrbracket[\llbracket V \rrbracket/x], \underline{\langle}{\downarrow}\mathscr{V}_{fs}[\llbracket V \rrbracket/x], \underline{\langle}{\downarrow}\mathscr{V}_{ret}[\llbracket V \rrbracket/x], {\downarrow}\mathscr{V}_{ops}[\llbracket V \rrbracket/x]\underline{\rangle\rangle} :: \underline{[]\rangle\rangle} \\
& \quad \underline{@}{\downarrow}\mathscr{W}[\llbracket V \rrbracket/x] \\
= \quad & \text{(IH 2 on } W) \\
& {\downarrow}\mathscr{V}_{ops}[\llbracket V \rrbracket/x] \underline{@} \underline{\langle}\ell, \underline{\langle}\llbracket W[V/x] \rrbracket, \underline{\langle}{\downarrow}\mathscr{V}_{fs}[\llbracket V \rrbracket/x], \underline{\langle}{\downarrow}\mathscr{V}_{ret}[\llbracket V \rrbracket/x], {\downarrow}\mathscr{V}_{ops}[\llbracket V \rrbracket/x]\underline{\rangle\rangle} :: \underline{[]\rangle\rangle} \\
& \quad \underline{@}{\downarrow}\mathscr{W}[\llbracket V \rrbracket/x] \\
= \quad & \text{(static } \beta\text{-conversion)} \\
& (\overline{\lambda}\overline{\langle}\theta, \overline{\langle}\chi^{\mathrm{ret}}, \chi^{\mathrm{ops}}\overline{\rangle\rangle} :: \kappa.h^{\mathrm{ops}} \mathbin{\underline{@}} \underline{\langle}\ell, \underline{\langle}\llbracket W[V/x] \rrbracket, \underline{\langle}{\downarrow}\theta, \underline{\langle}{\downarrow}\chi^{\mathrm{ret}}, {\downarrow}\chi^{\mathrm{ops}}\underline{\rangle\rangle} :: \underline{[]\rangle\rangle} \mathbin{\underline{@}} {\downarrow}\kappa) \\
& \quad \mathbin{\overline{@}} (\overline{\langle} \mathscr{V}_{fs}, \overline{\langle} \mathscr{V}_{ret}, \mathscr{V}_{ops}\overline{\rangle\rangle} :: \mathscr{W})[\llbracket V \rrbracket/x] \\
= \quad & \text{(definition of } \llbracket - \rrbracket) \\
& (\llbracket \mathbf{do}\ (\ell\, W[V/x])^E \rrbracket \mathbin{\overline{@}} (\overline{\langle} \mathscr{V}_{fs}, \overline{\langle} \mathscr{V}_{ret}, \mathscr{V}_{ops}\overline{\rangle\rangle} :: \mathscr{W})[\llbracket V \rrbracket/x] \\
= \quad & \text{(definition of } [-/-]) \\
& (\llbracket (\mathbf{do}\ (\ell\, W)^E)[V/x] \rrbracket \mathbin{\overline{@}} (\overline{\langle} \mathscr{V}_{fs}, \overline{\langle} \mathscr{V}_{ret}, \mathscr{V}_{ops}\overline{\rangle\rangle} :: \mathscr{W})[\llbracket V \rrbracket/x]
\end{aligned}
$$

Case $M = \mathbf{handle}^{\delta}\ M'\ \mathbf{with}\ H$. We make use of two auxiliary results.

1. $\llbracket H^{\mathrm{ret}} \rrbracket[\llbracket V \rrbracket/x] = \llbracket H^{\mathrm{ret}}[V/x] \rrbracket$
2. $\llbracket H^{\mathrm{ops}} \rrbracket^{\delta}[\llbracket V \rrbracket/x] = \llbracket H^{\mathrm{ops}}[V/x] \rrbracket^{\delta}$

Proof
Suppose $H^{\mathrm{ret}} = \{\mathbf{return}\ y \mapsto N\}$.

$$
\begin{aligned}
& \llbracket H^{\mathrm{ret}} \rrbracket[\llbracket V \rrbracket/x] \\
= \quad & \text{(definition of } \llbracket - \rrbracket) \\
& (\underline{\lambda} y\, k.\mathbf{let}\ \underline{\langle} fs, \underline{\langle} h^{\mathrm{ret}}, h^{\mathrm{ops}}\underline{\rangle\rangle} :: k' = k\ \mathbf{in}\ \llbracket N \rrbracket \mathbin{\overline{@}} (\overline{\langle}{\uparrow}fs, \overline{\langle}{\uparrow}h^{\mathrm{ret}}, {\uparrow}h^{\mathrm{ops}}\overline{\rangle\rangle} :: {\uparrow}k'))[\llbracket V \rrbracket/x] \\
= \quad & \text{(definition of } [-/-]) \\
& \underline{\lambda} y\, k.\mathbf{let}\ \underline{\langle} fs, \underline{\langle} h^{\mathrm{ret}}, h^{\mathrm{ops}}\underline{\rangle\rangle} :: k' = k\ \mathbf{in}\ (\llbracket N \rrbracket \mathbin{\overline{@}} (\overline{\langle}{\uparrow}fs, \overline{\langle}{\uparrow}h^{\mathrm{ret}}, {\uparrow}h^{\mathrm{ops}}\overline{\rangle\rangle} :: {\uparrow}k'))[\llbracket V \rrbracket/x] \\
= \quad & \text{(IH 1 for } N) \\
& \underline{\lambda} y\, k.\mathbf{let}\ \underline{\langle} fs, \underline{\langle} h^{\mathrm{ret}}, h^{\mathrm{ops}}\underline{\rangle\rangle} :: k' = k\ \mathbf{in}\ \llbracket N[V/x] \rrbracket \mathbin{\overline{@}} (\overline{\langle}{\uparrow}fs, \overline{\langle}{\uparrow}h^{\mathrm{ret}}, {\uparrow}h^{\mathrm{ops}}\overline{\rangle\rangle} :: {\uparrow}k') \\
= \quad & \text{(definition of } \llbracket - \rrbracket) \\
& \llbracket H^{\mathrm{ret}}[V/x] \rrbracket
\end{aligned}
$$

The $H^{\mathrm{ops}} = \{(\ell\, p\, r \mapsto N_\ell)_{\ell \in \mathscr{L}}\}$ case goes through similarly. □

We can now prove that substitution commutes with the translation of handlers:

$$
\begin{array}{ll}
& (\llbracket \mathbf{handle}^{\delta}\ M'\ \mathbf{with}\ H \rrbracket\ \overline{@}\ \mathscr{W})[\llbracket V \rrbracket/x] \\
= & \quad \text{(definition of } \llbracket - \rrbracket) \\
& (\overline{\lambda}\,\kappa.\llbracket M' \rrbracket\ \overline{@}\ \overline{\langle}\uparrow\underline{[]}, \overline{\langle} \llbracket H^{\mathrm{ret}} \rrbracket, \llbracket H^{\mathrm{ops}} \rrbracket^{\delta} \overline{\rangle\rangle}\ \overline{::}\ \kappa)\ \overline{@}\ \mathscr{W})[\llbracket V \rrbracket/x] \\
= & \quad \text{(static } \beta\text{-conversion)} \\
& (\llbracket M' \rrbracket\ \overline{@}\ \overline{\langle}\uparrow\underline{[]}, \overline{\langle} \llbracket H^{\mathrm{ret}} \rrbracket, \llbracket H^{\mathrm{ops}} \rrbracket^{\delta} \overline{\rangle\rangle}\ \overline{::}\ \mathscr{W})[\llbracket V \rrbracket/x] \\
= & \quad \text{(IH 1 for } M') \\
& \llbracket M'[V/x] \rrbracket\ \overline{@}\ \overline{\langle}\uparrow\underline{[]}, \overline{\langle} \llbracket H^{\mathrm{ret}} \rrbracket[\llbracket V \rrbracket/x], \llbracket H^{\mathrm{ops}} \rrbracket^{\delta}[\llbracket V \rrbracket/x] \overline{\rangle\rangle}\ \overline{::}\ \mathscr{W}[\llbracket V \rrbracket/x] \\
= & \quad \text{((1) and (2))} \\
& \llbracket M'[V/x] \rrbracket\ \overline{@}\ \overline{\langle}\uparrow\underline{[]}, \overline{\langle} \llbracket H^{\mathrm{ret}}[V/x] \rrbracket, \llbracket H^{\mathrm{ops}}[V/x] \rrbracket^{\delta} \overline{\rangle\rangle}\ \overline{::}\ \mathscr{W}[\llbracket V \rrbracket/x] \\
= & \quad \text{(static } \beta\text{-conversion)} \\
& (\overline{\lambda}\,\kappa.\llbracket M'[V/x] \rrbracket\ \overline{@}\ \overline{\langle}\uparrow\underline{[]}, \overline{\langle} \llbracket H^{\mathrm{ret}}[V/x] \rrbracket, \llbracket H^{\mathrm{ops}}[V/x] \rrbracket^{\delta} \overline{\rangle\rangle}\ \overline{::}\ \kappa)\ \overline{@}\ \mathscr{W}[\llbracket V \rrbracket/x]) \\
= & \quad \text{(definition of } \llbracket - \rrbracket) \\
& \llbracket \mathbf{handle}^{\delta}\ M'[V/x]; \mathbf{with}\ H[V/x] \rrbracket\ \overline{@}\ \mathscr{W}[\llbracket V \rrbracket/x] \\
= & \quad \text{(definition of } [-/-]) \\
& \llbracket (\mathbf{handle}^{\delta}\ M'\ \mathbf{with}\ H)[V/x] \rrbracket\ \overline{@}\ \mathscr{W}[\llbracket V \rrbracket/x]
\end{array}
$$

□

Lemma 8 (*Type erasure*)

1. $\llbracket M \rrbracket\ \overline{@}\ \mathscr{W} = \llbracket M[T/\alpha] \rrbracket\ \overline{@}\ \mathscr{W}$
2. $\llbracket W \rrbracket = \llbracket W[T/\alpha] \rrbracket$

Proof

Follows from the observation that the translation is oblivious to types. □

Lemma 3 (*Decomposition*)

$\llbracket \mathscr{E}[M] \rrbracket\ \overline{@}\ (\overline{\langle} \mathscr{V}_{fs}, \overline{\langle} \mathscr{V}_{ret}, \mathscr{V}_{ops} \overline{\rangle\rangle}\ \overline{::}\ \mathscr{W}) = \llbracket M \rrbracket\ \overline{@}\ (\llbracket \mathscr{E} \rrbracket\ \overline{@}\ (\overline{\langle} \mathscr{V}_{fs}, \overline{\langle} \mathscr{V}_{ret}, \mathscr{V}_{ops} \overline{\rangle\rangle}\ \overline{::}\ \mathscr{W}))$.

Proof

For reference, we repeat the translation of evaluations contexts here:

$$
\begin{array}{rl}
\llbracket [\,] \rrbracket = & \overline{\lambda}\,\kappa.\kappa \\
\llbracket \mathbf{let}\ x \leftarrow \mathscr{E}\ \mathbf{in}\ N \rrbracket = & \overline{\lambda}\,\overline{\langle}\theta, \overline{\langle}\chi^{\mathrm{ret}}, \chi^{\mathrm{ops}} \overline{\rangle\rangle}\ \overline{::}\ \kappa. \\
& \quad \llbracket \mathscr{E} \rrbracket\ \overline{@}\ (\overline{\langle}\uparrow((\underline{\lambda} x\,k.\mathbf{let}\ \underline{\langle} fs, \underline{\langle} h^{\mathrm{ret}}, h^{\mathrm{ops}} \overline{\rangle\rangle}\ \underline{::} k' = k\ \mathbf{in} \\
& \quad\quad \llbracket N \rrbracket\ \overline{@}\ (\overline{\langle}\uparrow fs, \overline{\langle}\uparrow h^{\mathrm{ret}}, \uparrow h^{\mathrm{ops}} \overline{\rangle\rangle}\ \overline{::}\ \uparrow k'))\ \underline{::} \downarrow\theta), \\
& \quad\quad \overline{\langle} \chi^{\mathrm{ret}}, \chi^{\mathrm{ops}} \overline{\rangle\rangle}\ \overline{::}\ \kappa) \\
\llbracket \mathbf{handle}^{\delta}\ \mathscr{E}\ \mathbf{with}\ H \rrbracket = & \overline{\lambda}\,\kappa.\llbracket \mathscr{E} \rrbracket\ \overline{@}\ (\overline{\langle}[], \llbracket H \rrbracket^{\delta} \overline{\rangle}\ \overline{::}\ \kappa)
\end{array}
$$

The proof proceeds by structural induction on the evaluation context $\mathscr{E}$.

Case $\mathscr{E} = [\,]$.

$$
\begin{array}{ll}
& \llbracket \mathscr{E}[M] \rrbracket\ \overline{@}\ (\mathscr{V} \overline{::} \mathscr{W}) \\
= & \quad \text{(assumption)} \\
& \llbracket M \rrbracket\ \overline{@}\ (\mathscr{V} \overline{::} \mathscr{W}) \\
= & \quad \text{(static } \beta\text{-conversion)} \\
& \llbracket M \rrbracket\ \overline{@}\ ((\overline{\lambda}\,\kappa.\kappa)\ \overline{@}\ (\mathscr{V} \overline{::} \mathscr{W})) \\
= & \quad \text{(definition of } \llbracket - \rrbracket) \\
& \llbracket M \rrbracket\ \overline{@}\ (\llbracket \mathscr{E} \rrbracket\ \overline{@}\ (\mathscr{V} \overline{::} \mathscr{W}))
\end{array}
$$

Case $\mathscr{E} = \mathbf{let}\ x \leftarrow \mathscr{E}'[-]\ \mathbf{in}\ N$.

$$
\begin{aligned}
& [\![\mathscr{E}[M]]\!]\ \overline{@}\ (\overline{\langle} \mathscr{V}_{fs}, \overline{\langle} \mathscr{V}_{ret}, \mathscr{V}_{ops} \overline{\rangle\rangle} :: \mathscr{W}) \\
= \quad & \text{(assumption)} \\
& [\![\mathbf{let}\ x \leftarrow \mathscr{E}'[M]\ \mathbf{in}\ N]\!]\ \overline{@}\ (\overline{\langle} \mathscr{V}_{fs}, \overline{\langle} \mathscr{V}_{ret}, \mathscr{V}_{ops} \overline{\rangle\rangle} :: \mathscr{W}) \\
= \quad & \text{(definition of } [\![-]\!]) \\
& (\overline{\lambda} \overline{\langle} \theta, \overline{\langle} \chi^{\mathrm{ret}}, \chi^{\mathrm{ops}} \overline{\rangle\rangle} :: \kappa. \\
& \qquad [\![\mathscr{E}'[M]]\!]\ \overline{@}\ (\overline{\langle} \uparrow((\underline{\lambda} x\, k.\mathbf{let}\ \underline{\langle} fs, \underline{\langle} h^{\mathrm{ret}}, h^{\mathrm{ops}} \underline{\rangle\rangle} :: k' = k\ \mathbf{in} \\
& \qquad\qquad [\![N]\!]\ \overline{@}\ (\overline{\langle} \uparrow fs, \overline{\langle} \uparrow h^{\mathrm{ret}}, \uparrow h^{\mathrm{ops}} \overline{\rangle\rangle} :: \uparrow k')) :: \underline{\downarrow} \theta), \overline{\langle} \chi^{\mathrm{ret}}, \chi^{\mathrm{ops}} \overline{\rangle\rangle} :: \kappa)) \\
& \quad \overline{@}\ (\overline{\langle} \mathscr{V}_{fs}, \overline{\langle} \mathscr{V}_{ret}, \mathscr{V}_{ops} \overline{\rangle\rangle} :: \mathscr{W}) \\
= \quad & \text{(static } \beta\text{-conversion)} \\
& [\![\mathscr{E}'[M]]\!]\ \overline{@}\ (\overline{\langle} \uparrow((\underline{\lambda} x\, k.\mathbf{let}\ \underline{\langle} fs, \underline{\langle} h^{\mathrm{ret}}, h^{\mathrm{ops}} \underline{\rangle\rangle} :: k' = k\ \mathbf{in} \\
& \qquad\qquad [\![N]\!]\ \overline{@}\ (\overline{\langle} \uparrow fs, \overline{\langle} \uparrow h^{\mathrm{ret}}, \uparrow h^{\mathrm{ops}} \overline{\rangle\rangle} :: \uparrow k')) :: \underline{\downarrow} \mathscr{V}_{fs}), \overline{\langle} \mathscr{V}_{ret}, \mathscr{V}_{ops} \overline{\rangle\rangle} :: \mathscr{W}) \\
= \quad & \text{(IH for } \mathscr{E}'[-]) \\
& [\![M]\!]\ \overline{@}\ ([\![\mathscr{E}']\!]\ \overline{@} \\
& \qquad (\overline{\langle} \uparrow((\underline{\lambda} x\, k.\mathbf{let}\ \underline{\langle} fs, \underline{\langle} h^{\mathrm{ret}}, h^{\mathrm{ops}} \underline{\rangle\rangle} :: k' = k\ \mathbf{in} \\
& \qquad\qquad [\![N]\!]\ \overline{@}\ (\overline{\langle} \uparrow fs, \overline{\langle} \uparrow h^{\mathrm{ret}}, \uparrow h^{\mathrm{ops}} \overline{\rangle\rangle} :: \uparrow k')) :: \underline{\downarrow} \mathscr{V}_{fs}), \overline{\langle} \mathscr{V}_{ret}, \mathscr{V}_{ops} \overline{\rangle\rangle} :: \mathscr{W})) \\
= \quad & \text{(static } \beta\text{-conversion)} \\
& [\![M]\!]\ \overline{@}\ ((\overline{\lambda} \overline{\langle} \theta, \overline{\langle} \chi^{\mathrm{ret}}, \chi^{\mathrm{ops}} \overline{\rangle\rangle} :: \kappa. \\
& \qquad [\![\mathscr{E}']\!]\ \overline{@}\ (\overline{\langle} \uparrow((\underline{\lambda} x\, k.\mathbf{let}\ \underline{\langle} fs, \underline{\langle} h^{\mathrm{ret}}, h^{\mathrm{ops}} \underline{\rangle\rangle} :: k' = k\ \mathbf{in} \\
& \qquad\qquad [\![N]\!]\ \overline{@}\ \overline{\langle} \uparrow fs, \overline{\langle} \uparrow h^{\mathrm{ret}}, \uparrow h^{\mathrm{ops}} \overline{\rangle\rangle} :: \uparrow k')) :: \underline{\downarrow} \theta), \\
& \qquad \overline{\langle} \chi^{\mathrm{ret}}, \chi^{\mathrm{ops}} \overline{\rangle\rangle} :: \kappa))\ \overline{@}\ (\overline{\langle} \mathscr{V}_{fs}, \overline{\langle} \mathscr{V}_{ret}, \mathscr{V}_{ops} \overline{\rangle\rangle} :: \mathscr{W})) \\
= \quad & \text{(definition of } [\![-]\!]) \\
& [\![M]\!]\ \overline{@}\ ([\![\mathbf{let}\ x \leftarrow \mathscr{E}'[M]\ \mathbf{in}\ N]\!])\ \overline{@}\ (\overline{\langle} \mathscr{V}_{fs}, \overline{\langle} \mathscr{V}_{ret}, \mathscr{V}_{ops} \overline{\rangle\rangle} :: \mathscr{W})) \\
= \quad & \text{(assumption)} \\
& [\![M]\!]\ \overline{@}\ ([\![\mathscr{E}]\!]\ \overline{@}\ (\overline{\langle} \mathscr{V}_{fs}, \overline{\langle} \mathscr{V}_{ret}, \mathscr{V}_{ops} \overline{\rangle\rangle} :: \mathscr{W}))
\end{aligned}
$$

Case $\mathscr{E} = \mathbf{handle}^{\delta}\ \mathscr{E}'\ \mathbf{with}\ H$.

$$
\begin{aligned}
& [\![\mathscr{E}[M]]\!]\ \overline{@}\ (\mathscr{V} :: \mathscr{W}) \\
= \quad & \text{(assumption)} \\
& [\![\mathbf{handle}^{\delta}\ \mathscr{E}'[M]\ \mathbf{with}\ H]\!]\ \overline{@}\ (\mathscr{V} :: \mathscr{W}) \\
= \quad & \text{(definition of } [\![-]\!]) \\
& (\overline{\lambda} \kappa. [\![\mathscr{E}'[M]]\!]\ \overline{@}\ (\overline{\langle} [], [\![H]\!]^{\delta} \overline{\rangle} :: \kappa))\ \overline{@}\ (\mathscr{V} :: \mathscr{W}) \\
= \quad & \text{(static } \beta\text{-conversion)} \\
& [\![\mathscr{E}'[M]]\!]\ \overline{@}\ (\overline{\langle} [], [\![H]\!]^{\delta} \overline{\rangle} :: (\mathscr{V} :: \mathscr{W})) \\
= \quad & \text{(IH)} \\
& [\![M]\!]\ \overline{@}\ ([\![\mathscr{E}']\!]\ \overline{@}\ (\overline{\langle} [], [\![H]\!]^{\delta} \overline{\rangle} :: (\mathscr{V} :: \mathscr{W}))) \\
= \quad & \text{(static } \beta\text{-conversion)} \\
& [\![M]\!]\ \overline{@}\ ((\overline{\lambda} \kappa. [\![\mathscr{E}']\!]\ \overline{@}\ (\overline{\langle} [], [\![H]\!]^{\delta} \overline{\rangle} :: \kappa))\ \overline{@}\ (\mathscr{V} :: \mathscr{W})) \\
= \quad & \text{(definition of } [\![-]\!]) \\
& [\![M]\!]\ \overline{@}\ ([\![\mathbf{handle}^{\delta}\ \mathscr{E}'\ \mathbf{with}\ H]\!]\ \overline{@}\ (\mathscr{V} :: \mathscr{W})) \\
= \quad & \text{(assumption)} \\
& [\![M]\!]\ \overline{@}\ ([\![\mathscr{E}]\!]\ \overline{@}\ (\mathscr{V} :: \mathscr{W}))
\end{aligned}
$$

□

Lemma 4 (*Reflect after reify*)
$[\![M]\!]\,\overline{@}\,(\overline{\langle}\mathscr{V}_{fs},\overline{\langle}\mathscr{V}_{ret},\mathscr{V}_{ops}\overline{\rangle\rangle}\,\overline{::}\uparrow\downarrow\mathscr{W}) = [\![M]\!]\,\overline{@}\,(\overline{\langle}\mathscr{V}_{fs},\overline{\langle}\mathscr{V}_{ret},\mathscr{V}_{ops}\overline{\rangle\rangle}\,\overline{::}\,\mathscr{W})$.

Proof

For an inductive proof to go through in the presence of **let** and **handle**, which alter or extend the continuation stack, we generalise the lemma statement to include an arbitrary list of handler frames:

$$[\![M]\!]\,\overline{@}\,(\overline{\langle}\mathscr{V}_{fs},\overline{\langle}\mathscr{V}_{ret},\mathscr{V}_{ops}\overline{\rangle\rangle}\,\overline{::}\,\mathscr{V}_1\,\overline{::}\ldots\mathscr{V}_n\,\overline{::}\uparrow\downarrow\mathscr{W}) = [\![M]\!]\,\overline{@}\,(\overline{\langle}\mathscr{V}_{fs},\overline{\langle}\mathscr{V}_{ret},\mathscr{V}_{ops}\overline{\rangle\rangle}\,\overline{::}\,\mathscr{V}_1\,\overline{::}\ldots\mathscr{V}_n\,\overline{::}\,\mathscr{W})$$

This is the lemma statement when $n = 0$. The proof now proceeds by induction on the structure of M. Most of the translated terms do not examine the top of the continuation stack, so we will write $\mathscr{V}_0$ for $\overline{\langle}\mathscr{V}_{fs},\overline{\langle}\mathscr{V}_{ret},\mathscr{V}_{ops}\overline{\rangle\rangle}$ to save space.

Case $M = V\,W$.

$$\begin{array}{ll}
& [\![V\,W]\!]\,\overline{@}\,(\mathscr{V}_0\,\overline{::}\cdots\overline{::}\,\mathscr{V}_n\,\overline{::}\uparrow\downarrow\mathscr{W}) \\
= & \quad(\text{definition of } [\![-]\!]) \\
& (\overline{\lambda}\,\kappa.[\![V]\!]\,\underline{@}\,[\![W]\!]\,\underline{@}\downarrow\kappa)\,\overline{@}\,(\mathscr{V}_0\,\overline{::}\cdots\overline{::}\,\mathscr{V}_n\,\overline{::}\uparrow\downarrow\mathscr{W}) \\
= & \quad(\text{static } \beta\text{-conversion}) \\
& [\![V]\!]\,\underline{@}\,[\![W]\!]\,\underline{@}\downarrow(\mathscr{V}_0\,\overline{::}\cdots\overline{::}\,\mathscr{V}_n\,\overline{::}\uparrow\downarrow\mathscr{W}) \\
= & \quad(\text{definition of } \downarrow) \\
& [\![V]\!]\,\underline{@}\,[\![W]\!]\,\underline{@}\,(\mathscr{V}_1\,\underline{::}\cdots\underline{::}\,\mathscr{V}_n\,\underline{::}\downarrow\mathscr{W}) \\
= & \quad(\text{definition of } \downarrow) \\
& [\![V]\!]\,\underline{@}\,[\![W]\!]\,\underline{@}\downarrow(\mathscr{V}_0\,\overline{::}\cdots\overline{::}\,\mathscr{V}_n\,\overline{::}\,\mathscr{W}) \\
= & \quad(\text{static } \beta\text{-conversion}) \\
& (\overline{\lambda}\,\kappa s.[\![V]\!]\,\underline{@}\,[\![W]\!]\,\underline{@}\downarrow\kappa s)\,\overline{@}\,(\mathscr{V}_0\,\overline{::}\cdots\overline{::}\,\mathscr{V}_n\,\overline{::}\,\mathscr{W}) \\
= & \quad(\text{definition of } [\![-]\!]) \\
& [\![V\,W]\!]\,\overline{@}\,(\mathscr{V}_0\,\overline{::}\cdots\overline{::}\,\mathscr{V}_n\,\overline{::}\,\mathscr{W})
\end{array}$$

Case $M = V\,T$. Similar to the $M = V\,W$ case.

Case $M = \mathbf{let}\,\langle\ell = x; y\rangle = V\,\mathbf{in}\,N$.

$$\begin{array}{ll}
& [\![\mathbf{let}\,\langle\ell = x; y\rangle = V\,\mathbf{in}\,N]\!]\,\overline{@}\,(\mathscr{V}_0\,\overline{::}\cdots\overline{::}\,\mathscr{V}_n\,\overline{::}\uparrow\downarrow\mathscr{W}) \\
= & \quad(\text{definition of } [\![-]\!]) \\
& (\overline{\lambda}\,\kappa.\mathbf{let}\,\langle\ell = x; y\rangle = [\![V]\!]\,\mathbf{in}\,[\![N]\!]\,\overline{@}\,\kappa)\,\overline{@}\,(\mathscr{V}_0\,\overline{::}\cdots\overline{::}\,\mathscr{V}_n\,\overline{::}\uparrow\downarrow\mathscr{W}) \\
= & \quad(\text{static } \beta\text{-conversion}) \\
& \mathbf{let}\,\langle\ell = x; y\rangle = [\![V]\!]\,\mathbf{in}\,[\![N]\!]\,\overline{@}\,(\mathscr{V}_0\,\overline{::}\cdots\overline{::}\,\mathscr{V}_n\,\overline{::}\uparrow\downarrow\mathscr{W}) \\
= & \quad(\text{IH}) \\
& \mathbf{let}\,\langle\ell = x; y\rangle = [\![V]\!]\,\mathbf{in}\,[\![N]\!]\,\overline{@}\,(\mathscr{V}_0\,\overline{::}\cdots\overline{::}\,\mathscr{V}_n\,\overline{::}\,\mathscr{W}) \\
= & \quad(\text{static } \beta\text{-conversion}) \\
& (\overline{\lambda}\,\kappa.\mathbf{let}\,\langle\ell = x; y\rangle = [\![V]\!]\,\mathbf{in}\,[\![N]\!]\,\overline{@}\,\kappa)\,\overline{@}\,(\mathscr{V}_0\,\overline{::}\cdots\overline{::}\,\mathscr{V}_n\,\overline{::}\,\mathscr{W}) \\
= & \quad(\text{definition of } [\![-]\!]) \\
& [\![\mathbf{let}\,\langle\ell = x; y\rangle = [\![V]\!]\,\mathbf{in}\,[\![N]]\!]\,\overline{@}\,(\mathscr{V}_0\,\overline{::}\cdots\overline{::}\,\mathscr{V}_n\,\overline{::}\,\mathscr{W})
\end{array}$$

Case $M = \mathbf{case}\,V\{\ell\,x \mapsto M; y \mapsto N\}$. Similar to the $M = \mathbf{let}\,\langle\ell = x; y\rangle = V\,\mathbf{in}\,N$ case.

Case $M = \mathbf{absurd}\ V$.

$$
\begin{array}{ll}
& [\![\mathbf{absurd}\ V]\!] \mathbin{\overline{@}} (\mathscr{V}_0 :\!\overline{:} \cdots :\!\overline{:} \mathscr{V}_n :\!\overline{:} \uparrow\downarrow \mathscr{W}) \\
= & \quad \text{(definition of } [\![-]\!]) \\
& (\overline{\lambda}\, \kappa.\mathbf{absurd}\ [\![V]\!]) \mathbin{\overline{@}} (\mathscr{V}_0 :\!\overline{:} \cdots :\!\overline{:} \mathscr{V}_n :\!\overline{:} \uparrow\downarrow \mathscr{W}) \\
= & \quad \text{(static } \beta\text{-conversion)} \\
& \mathbf{absurd}\ [\![V]\!] \\
= & \quad \text{(static } \beta\text{-conversion)} \\
& (\overline{\lambda}\, \kappa s.\mathbf{absurd}\ [\![V]\!]) \mathbin{\overline{@}} (\mathscr{V}_0 :\!\overline{:} \cdots :\!\overline{:} \mathscr{V}_n :\!\overline{:} \mathscr{W}) \\
= & \quad \text{(definition of } [\![-]\!]) \\
& [\![\mathbf{absurd}\ V]\!] \mathbin{\overline{@}} (\mathscr{V}_0 :\!\overline{:} \cdots :\!\overline{:} \mathscr{V}_n :\!\overline{:} \mathscr{W})
\end{array}
$$

Case $M = \mathbf{return}\ V$.

$$
\begin{array}{ll}
& [\![\mathbf{return}\ V]\!] \mathbin{\overline{@}} (\mathscr{V}_0 :\!\overline{:} \cdots :\!\overline{:} \mathscr{V}_n :\!\overline{:} \uparrow\downarrow \mathscr{W}) \\
= & \quad \text{(definition of } [\![-]\!]) \\
& (\overline{\lambda}\, \kappa.\mathbf{app}\ (\downarrow\kappa)\ [\![V]\!]) \mathbin{\overline{@}} (\mathscr{V}_0 :\!\overline{:} \cdots :\!\overline{:} \mathscr{V}_n :\!\overline{:} \uparrow\downarrow \mathscr{W}) \\
= & \quad \text{(static } \beta\text{-conversion)} \\
& \mathbf{app}\ (\downarrow(\mathscr{V}_0 :\!\overline{:} \cdots :\!\overline{:} \mathscr{V}_n :\!\overline{:} \uparrow\downarrow \mathscr{W}))\ [\![V]\!] \\
= & \quad \text{(definition of } \downarrow) \\
& \mathbf{app}\ (\downarrow\mathscr{V}_0 :\!\underline{:} \cdots :\!\underline{:} \downarrow\mathscr{V}_n :\!\underline{:} \downarrow\mathscr{W}))\ [\![V]\!] \\
= & \quad \text{(definition of } \downarrow) \\
& \mathbf{app}\ (\downarrow(\mathscr{V}_0 :\!\overline{:} \cdots :\!\overline{:} \mathscr{V}_n :\!\overline{:} \mathscr{W}))\ [\![V]\!] \\
= & \quad \text{(static } \beta\text{-conversion)} \\
& (\overline{\lambda}\, \kappa.\mathbf{app}\ (\downarrow\kappa)\ [\![V]\!]) \mathbin{\overline{@}} (\mathscr{V}_0 :\!\overline{:} \cdots :\!\overline{:} \mathscr{V}_n :\!\overline{:} \mathscr{W}) \\
= & \quad \text{(definition of } [\![-]\!]) \\
& [\![\mathbf{return}\ V]\!] \mathbin{\overline{@}} (\mathscr{V}_0 :\!\overline{:} \cdots :\!\overline{:} \mathscr{V}_n :\!\overline{:} \mathscr{W})
\end{array}
$$

Case $M = \mathbf{let}\ x \leftarrow M'\ \mathbf{in}\ N$.

$$
\begin{array}{ll}
& [\![\mathbf{let}\ x \leftarrow M'\ \mathbf{in}\ N]\!] \mathbin{\overline{@}} (\overline{\langle} \mathscr{V}_{fs}, \overline{\langle} \mathscr{V}_{ret}, \mathscr{V}_{ops} \overline{\rangle\rangle} :\!\overline{:} \mathscr{V}_1 :\!\overline{:} \cdots :\!\overline{:} \mathscr{V}_n :\!\overline{:} \uparrow\downarrow \mathscr{W}) \\
= & \quad \text{(definition of } [\![-]\!]) \\
& (\overline{\lambda}\, \overline{\langle} \theta, \overline{\langle} \chi^{\mathrm{ret}}, \chi^{\mathrm{ops}} \overline{\rangle\rangle} :\!\overline{:} \kappa.[\![M']\!] \mathbin{\overline{@}} (\overline{\langle} \uparrow((\underline{\lambda} x k.\mathbf{let}\ \underline{\langle} fs, \underline{\langle} h^{\mathrm{ret}}, h^{\mathrm{ops}} \underline{\rangle\rangle} :\!\underline{:} k' = k; \mathbf{in} \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad [\![N]\!] \mathbin{\overline{@}} (\overline{\langle} \uparrow fs, \overline{\langle} \uparrow h^{\mathrm{ret}}, \uparrow h^{\mathrm{ops}} \overline{\rangle\rangle} :\!\overline{:} \uparrow k')) :\!\underline{:} \downarrow\theta), \\
& \qquad\qquad\qquad\qquad\qquad\qquad \overline{\langle} \chi^{\mathrm{ret}}, \chi^{\mathrm{ops}} \overline{\rangle\rangle} :\!\overline{:} \kappa))) \\
& \qquad \mathbin{\overline{@}} (\overline{\langle} \mathscr{V}_{fs}, \overline{\langle} \mathscr{V}_{ret}, \mathscr{V}_{ops} \overline{\rangle\rangle} :\!\overline{:} \mathscr{V}_1 :\!\overline{:} \cdots :\!\overline{:} \mathscr{V}_n :\!\overline{:} \uparrow\downarrow \mathscr{W}) \\
= & \quad \text{(static } \beta\text{-conversion)} \\
& [\![M']\!] \mathbin{\overline{@}} (\overline{\langle} \uparrow((\underline{\lambda} x k.\mathbf{let}\ \underline{\langle} fs, \underline{\langle} h^{\mathrm{ret}}, h^{\mathrm{ops}} \underline{\rangle\rangle} :\!\underline{:} k' = k; \mathbf{in} \\
& \qquad\qquad\qquad\qquad [\![N]\!] \mathbin{\overline{@}} (\overline{\langle} \uparrow fs, \overline{\langle} \uparrow h^{\mathrm{ret}}, \uparrow h^{\mathrm{ops}} \overline{\rangle\rangle} :\!\overline{:} \uparrow k')) :\!\underline{:} \downarrow\mathscr{V}_{fs}), \\
& \qquad\qquad \overline{\langle} \mathscr{V}_{ret}, \mathscr{V}_{ops} \overline{\rangle\rangle} :\!\overline{:} \mathscr{V}_1 :\!\overline{:} \cdots :\!\overline{:} \mathscr{V}_n :\!\overline{:} \uparrow\downarrow \mathscr{W}) \\
= & \quad \text{(IH on } M) \\
& [\![M']\!] \mathbin{\overline{@}} (\overline{\langle} \uparrow((\underline{\lambda} x k.\mathbf{let}\ \underline{\langle} fs, \underline{\langle} h^{\mathrm{ret}}, h^{\mathrm{ops}} \underline{\rangle\rangle} :\!\underline{:} k' = k; \mathbf{in} \\
& \qquad\qquad\qquad\qquad [\![N]\!] \mathbin{\overline{@}} (\overline{\langle} \uparrow fs, \overline{\langle} \uparrow h^{\mathrm{ret}}, \uparrow h^{\mathrm{ops}} \overline{\rangle\rangle} :\!\overline{:} \uparrow k')) :\!\underline{:} \downarrow\mathscr{V}_{fs}), \\
& \qquad\qquad \overline{\langle} \mathscr{V}_{ret}, \mathscr{V}_{ops} \overline{\rangle\rangle} :\!\overline{:} \mathscr{V}_1 :\!\overline{:} \cdots :\!\overline{:} \mathscr{V}_n :\!\overline{:} \mathscr{W})
\end{array}
$$

$$
\begin{array}{ll}
= & \text{(static } \beta\text{-conversion)} \\
& (\overline{\lambda}\,\overline{\langle}\theta, \overline{\langle}\chi^{\mathrm{ret}}, \chi^{\mathrm{ops}}\overline{\rangle}\rangle \mathbin{::} \kappa.[\![M']\!]\ \overline{@}\ (\overline{\langle}\uparrow((\underline{\lambda} x\,k.\mathbf{let}\ \underline{\langle} fs, \underline{\langle} h^{\mathrm{ret}}, h^{\mathrm{ops}}\underline{\rangle}\rangle \mathbin{::} k' = k; \mathbf{in} \\
& \qquad\qquad [\![N]\!]\ \overline{@}\ (\overline{\langle}\uparrow fs, \overline{\langle}\uparrow h^{\mathrm{ret}}, \uparrow h^{\mathrm{ops}}\overline{\rangle}\rangle \mathbin{::} \uparrow k')) \mathbin{::} \downarrow\theta), \\
& \qquad\qquad \overline{\langle}\chi^{\mathrm{ret}}, \chi^{\mathrm{ops}}\overline{\rangle}\rangle \mathbin{::} \kappa))) \\
& \overline{@}(\overline{\langle}\mathscr{V}_{fs}, \overline{\langle}\mathscr{V}_{ret}, \mathscr{V}_{ops}\overline{\rangle}\rangle \mathbin{::} \mathscr{V}_1 \mathbin{::} \cdots \mathbin{::} \mathscr{V}_n \mathbin{::} \mathscr{W}) \\
= & \text{(definition of } [\![-]\!]) \\
& [\![\mathbf{let}\ x \leftarrow M'\ \mathbf{in}\ N]\!]\ \overline{@}\ (\overline{\langle}\mathscr{V}_{fs}, \overline{\langle}\mathscr{V}_{ret}, \mathscr{V}_{ops}\overline{\rangle}\rangle \mathbin{::} \mathscr{V}_1 \mathbin{::} \cdots \mathbin{::} \mathscr{V}_n \mathbin{::} \mathscr{W})
\end{array}
$$

Case $M = \mathbf{do}\ \ell\ V$.

$$
\begin{array}{ll}
& [\![\mathbf{do}\ \ell\ V]\!]\ \overline{@}\ (\overline{\langle}\mathscr{V}_{fs}, \overline{\langle}\mathscr{V}_{ret}, \mathscr{V}_{ops}\overline{\rangle}\rangle \mathbin{::} \mathscr{V}_1 \mathbin{::} \cdots \mathbin{::} \mathscr{V}_n \mathbin{::} \uparrow\downarrow\mathscr{W}) \\
= & \text{(definition of } [\![-]\!]) \\
& (\overline{\lambda}\,\overline{\langle}\theta, \overline{\langle}\chi^{\mathrm{ret}}, \chi^{\mathrm{ops}}\overline{\rangle}\rangle \mathbin{::} \kappa.\downarrow\chi^{\mathrm{ops}}\ \underline{@}\ \underline{\langle}\ell, \underline{\langle}[\![V]\!], \underline{\langle}\downarrow\theta, \underline{\langle}\downarrow\chi^{\mathrm{ret}}, \downarrow\chi^{\mathrm{ops}}\underline{\rangle}\rangle \mathbin{::} [\underline{]}\rangle\rangle\ \underline{@}\ \downarrow\kappa) \\
& \quad \overline{@}(\overline{\langle}\mathscr{V}_{fs}, \overline{\langle}\mathscr{V}_{ret}, \mathscr{V}_{ops}\overline{\rangle}\rangle \mathbin{::} \mathscr{V}_1 \mathbin{::} \cdots \mathbin{::} \mathscr{V}_n \mathbin{::} \uparrow\downarrow\mathscr{W}) \\
= & \text{(static } \beta\text{-conversion)} \\
& \downarrow\mathscr{V}_{ops}\ \underline{@}\ \underline{\langle}\ell, \underline{\langle}[\![V]\!], \underline{\langle}\downarrow\mathscr{V}_{fs}, \underline{\langle}\downarrow\mathscr{V}_{ret}, \downarrow\mathscr{V}_{ops}\underline{\rangle}\rangle \mathbin{::} [\underline{]}\rangle\rangle\ \underline{@}\ \downarrow(\mathscr{V}_1 \mathbin{::} \cdots \mathbin{::} \mathscr{V}_n \mathbin{::} \uparrow\downarrow\mathscr{W}) \\
= & \text{(definition of } \downarrow) \\
& \downarrow\mathscr{V}_{ops}\ \underline{@}\ \underline{\langle}\ell, \underline{\langle}[\![V]\!], \underline{\langle}\downarrow\mathscr{V}_{fs}, \underline{\langle}\downarrow\mathscr{V}_{ret}, \downarrow\mathscr{V}_{ops}\underline{\rangle}\rangle \mathbin{::} [\underline{]}\rangle\rangle\ \underline{@}\ (\downarrow\mathscr{V}_1 \mathbin{::} \cdots \mathbin{::} \downarrow\mathscr{V}_n \mathbin{::} \downarrow\mathscr{W}) \\
= & \text{(definition of } \downarrow) \\
& \downarrow\mathscr{V}_{ops}\ \underline{@}\ \underline{\langle}\ell, \underline{\langle}[\![V]\!], \underline{\langle}\downarrow\mathscr{V}_{fs}, \underline{\langle}\downarrow\mathscr{V}_{ret}, \downarrow\mathscr{V}_{ops}\underline{\rangle}\rangle \mathbin{::} [\underline{]}\rangle\rangle\ \underline{@}\ \downarrow(\mathscr{V}_1 \mathbin{::} \cdots \mathbin{::} \mathscr{V}_n \mathbin{::} \mathscr{W}) \\
= & \text{(static } \beta\text{-conversion)} \\
& (\overline{\lambda}\,\overline{\langle}\theta, \overline{\langle}\chi^{\mathrm{ret}}, \chi^{\mathrm{ops}}\overline{\rangle}\rangle \mathbin{::} \kappa.\downarrow\chi^{\mathrm{ops}}\ \underline{@}\ \underline{\langle}\ell, \underline{\langle}[\![V]\!], \underline{\langle}\downarrow\theta, \underline{\langle}\downarrow\chi^{\mathrm{ret}}, \downarrow\chi^{\mathrm{ops}}\underline{\rangle}\rangle \mathbin{::} [\underline{]}\rangle\rangle\ \underline{@}\ \downarrow\kappa) \\
& \quad \overline{@}(\overline{\langle}\mathscr{V}_{fs}, \overline{\langle}\mathscr{V}_{ret}, \mathscr{V}_{ops}\overline{\rangle}\rangle \mathbin{::} \mathscr{V}_1 \mathbin{::} \cdots \mathbin{::} \mathscr{V}_n \mathbin{::} \mathscr{W}) \\
= & \text{(definition of } [\![-]\!]) \\
& [\![\mathbf{do}\ \ell\ V]\!]\ \overline{@}\ (\overline{\langle}\mathscr{V}_{fs}, \overline{\langle}\mathscr{V}_{ret}, \mathscr{V}_{ops}\overline{\rangle}\rangle \mathbin{::} \mathscr{V}_1 \mathbin{::} \cdots \mathbin{::} \mathscr{V}_n \mathbin{::} \mathscr{W})
\end{array}
$$

Case $\mathbf{handle}^{\delta}\ M\ \mathbf{with}\ H$.

$$
\begin{array}{ll}
& [\![\mathbf{handle}^{\delta}\ M\ \mathbf{with}\ H]\!]\ \overline{@}\ (\mathscr{V}_0 \mathbin{::} \cdots \mathbin{::} \mathscr{V}_n \mathbin{::} \uparrow\downarrow\mathscr{W}) \\
= & \text{(definition of } [\![-]\!]) \\
& (\overline{\lambda}\,\kappa.[\![M]\!]\ \overline{@}\ (\overline{\langle}\uparrow[\underline{]}, [\![H]\!]^{\delta}\overline{\rangle} \mathbin{::} \kappa))\ \overline{@}\ (\mathscr{V}_0 \mathbin{::} \cdots \mathbin{::} \mathscr{V}_n \mathbin{::} \uparrow\downarrow\mathscr{W}) \\
= & \text{(static } \beta\text{-conversion)} \\
& [\![M]\!]\ \overline{@}\ (\overline{\langle}\uparrow[\underline{]}, [\![H]\!]^{\delta}\overline{\rangle} \mathbin{::} \mathscr{V}_0 \mathbin{::} \cdots \mathbin{::} \mathscr{V}_n \mathbin{::} \uparrow\downarrow\mathscr{W}) \\
= & \text{(IH)}
\end{array}
$$

$$
\begin{array}{ll}
& [\![M]\!]\ \overline{@}\ (\overline{\langle}\uparrow[\underline{]}, [\![H]\!]^{\delta}\overline{\rangle} \mathbin{::} \mathscr{V}_0 \mathbin{::} \cdots \mathbin{::} \mathscr{V}_n \mathbin{::} \mathscr{W}) \\
= & \text{(static } \beta\text{-conversion)} \\
& (\overline{\lambda}\,\kappa.[\![M]\!]\ \overline{@}\ (\overline{\langle}\uparrow[\underline{]}, [\![H]\!]^{\delta}\overline{\rangle} \mathbin{::} \kappa))\ \overline{@}\ (\mathscr{V}_0 \mathbin{::} \cdots \mathbin{::} \mathscr{V}_n \mathbin{::} \mathscr{W}) \\
= & \text{(definition of } [\![-]\!]) \\
& [\![\mathbf{handle}^{\delta}\ M\ \mathbf{with}\ H]\!]\ \overline{@}\ (\mathscr{V}_0 \mathbin{::} \cdots \mathbin{::} \mathscr{V}_n \mathbin{::} \mathscr{W})
\end{array}
$$

□

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

$$
\begin{array}{l}
[\![H_1^{\mathrm{ops}}]\!]^{\delta}\ \underline{@}\ \underline{\langle}\ell, \underline{\langle}V_p, V_{rk}\underline{\rangle}\rangle\ \underline{@}\ (\underline{\langle}V_{fs}, [\![H_2]\!]^{\delta}\underline{\rangle} \mathbin{::} W) \rightsquigarrow^{+} \\
\qquad [\![H_2^{\mathrm{ops}}]\!]^{\delta}\ \underline{@}\ \underline{\langle}\ell, \underline{\langle}V_p, \underline{\langle}V_{fs}, [\![H_2]\!]^{\delta}\underline{\rangle} \mathbin{::} V_{rk}\underline{\rangle}\rangle\ \underline{@}\ W
\end{array}
$$

Proof

$$
\begin{array}{ll}
& [\![H_1^{\mathrm{ops}}]\!]^{\delta} \mathbin{@} \langle \ell, \langle V_p, V_{rk} \rangle \rangle \mathbin{@} (\langle V_{fs}, [\![H_2]\!]^{\delta} \rangle :: W) \\
\rightsquigarrow^{+} & \\
& M_{forward}((\ell, V_p, V_{rk}), \langle V_{fs}, [\![H_2]\!]^{\delta} \rangle :: W) \\
= & \\
& \textbf{let } \langle fs, \langle h^{\mathrm{ret}}, h^{\mathrm{ops}} \rangle \rangle :: k' = \langle V_{fs}, [\![H_2]\!]^{\delta} \rangle :: W \textbf{ in} \\
& \textbf{let } rk' = \langle fs, \langle h^{\mathrm{ret}}, h^{\mathrm{ops}} \rangle \rangle :: V_{rk} \textbf{ in} \\
& h^{\mathrm{ops}} \mathbin{@} \langle \ell, \langle V_p, rk' \rangle \rangle \mathbin{@} k' \\
\rightsquigarrow^{+} & \\
& [\![H_2^{\mathrm{ops}}]\!]^{\delta} \mathbin{@} \langle \ell, \langle V_p, \langle V_{fs}, [\![H_2]\!]^{\delta} \rangle :: V_{rk} \rangle \rangle \mathbin{@} W
\end{array}
$$

□

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

1. $[\![\mathbf{do}\ \ell\ V]\!] \mathbin{@} ([\![\mathscr{E}]\!] \mathbin{@} (\langle \uparrow[], [\![H]\!] \rangle :: \langle \mathscr{V}_{fs}, \langle \mathscr{V}_{ret}, \mathscr{V}_{ops} \rangle \rangle :: \mathscr{W})) \rightsquigarrow^{+}$
   $([\![N_{\ell}]\!] \mathbin{@} \langle \mathscr{V}_{fs}, \langle \mathscr{V}_{ret}, \mathscr{V}_{ops} \rangle \rangle :: \mathscr{W})$
   $[[\![V]\!]/p, \lambda y\, k.\mathbf{let}\ \langle fs, \langle h^{\mathrm{ret}}, h^{\mathrm{ops}} \rangle \rangle :: k' = k\ \mathbf{in}$
   $[\![\mathbf{return}\ y]\!] \mathbin{@} ([\![\mathscr{E}]\!] \mathbin{@} (\langle \uparrow[], [\![H]\!] \rangle :: \langle \uparrow s, \langle \uparrow h^{\mathrm{ret}}, \uparrow h^{\mathrm{ops}} \rangle \rangle :: \uparrow k'))/r],$
2. $[\![\mathbf{do}\ \ell\ V]\!] \mathbin{@} ([\![\mathscr{E}]\!] \mathbin{@} (\langle \uparrow[], [\![H]\!]^{\dagger} \rangle :: \langle \mathscr{V}_{fs}, \langle \mathscr{V}_{ret}, \mathscr{V}_{ops} \rangle \rangle :: \mathscr{W})) \rightsquigarrow^{+}$
   $([\![N_{\ell}]\!] \mathbin{@} \langle \mathscr{V}_{fs}, \langle \mathscr{V}_{ret}, \mathscr{V}_{ops} \rangle \rangle :: \mathscr{W})$
   $[[\![V]\!]/p, \lambda y\, k.\mathbf{let}\ \langle s, \langle h^{\mathrm{ret}}, h^{\mathrm{ops}} \rangle \rangle :: k' = k\ \mathbf{in}$
   $[\![\mathbf{return}\ y]\!] \mathbin{@} ([\![\mathscr{E}]\!] \mathbin{@} (\langle \uparrow s, \langle \uparrow h^{\mathrm{ret}}, \uparrow h^{\mathrm{ops}} \rangle \rangle :: \uparrow k'))/r].$

Proof
By the definition of $[\![-]\!]$ on evaluation contexts we can deduce that

$$[\![\mathscr{E}]\!] \mathbin{@} (\langle \uparrow V, [\![H]\!]^{\delta} \rangle :: \mathscr{W}) = \langle \uparrow V_1, [\![H_1]\!]^{\delta_1} \rangle :: \cdots :: \langle \uparrow (V_n +\!\!+ V), [\![H_n]\!]^{\delta_n} \rangle :: \mathscr{W} \qquad \text{(A 1)}$$

for some dynamic value terms $V_1, \ldots, V_n$, depths $\delta_1, \ldots, \delta_n$, and handlers $H_1, \ldots, H_n$, where $n \geq 1$, $H_n = H$, and $+\!\!+$ is (dynamic) list concatenation.

$$
\begin{array}{ll}
& [\![\mathbf{do}\ \ell\ V]\!] \mathbin{@} ([\![\mathscr{E}]\!] \mathbin{@} (\langle \uparrow[], [\![H]\!] \rangle :: \langle \mathscr{V}_{fs}, \langle \mathscr{V}_{ret}, \mathscr{V}_{ops} \rangle \rangle :: \mathscr{W})) \\
= & \quad \text{(definition of } [\![-]\!]) \\
& (\lambda \langle \theta, \langle \chi^{\mathrm{ret}}, \chi^{\mathrm{ops}} \rangle \rangle :: \kappa'.\downarrow \chi^{\mathrm{ops}} \mathbin{@} \langle \ell, \langle [\![V]\!], \langle \downarrow \theta, \langle \downarrow \chi^{\mathrm{ret}}, \downarrow \chi^{\mathrm{ops}} \rangle \rangle :: [] \rangle \rangle \mathbin{@} \downarrow \kappa) \\
& \quad \mathbin{@} ([\![\mathscr{E}]\!] \mathbin{@} (\langle \uparrow[], [\![H]\!] \rangle :: \langle \mathscr{V}_{fs}, \langle \mathscr{V}_{ret}, \mathscr{V}_{ops} \rangle \rangle :: \mathscr{W})) \\
= & \quad \text{(Equation A 1, above)} \\
& (\lambda \langle \theta, \langle \chi^{\mathrm{ret}}, \chi^{\mathrm{ops}} \rangle \rangle :: \kappa'.\downarrow \chi^{\mathrm{ops}} \mathbin{@} \langle \ell, \langle [\![V]\!], \langle \downarrow \theta, \langle \downarrow \chi^{\mathrm{ret}}, \downarrow \chi^{\mathrm{ops}} \rangle \rangle :: [] \rangle \rangle \mathbin{@} \downarrow \kappa) \\
& \quad \mathbin{@} (\langle \uparrow V_1, [\![H_1]\!]^{\delta_1} \rangle :: \cdots :: \langle \uparrow V_n, [\![H_n]\!]^{\delta_n} \rangle :: \langle \mathscr{V}_{fs}, \langle \mathscr{V}_{ret}, \mathscr{V}_{ops} \rangle \rangle :: \mathscr{W}) \\
= & \quad \text{(static } \beta\text{-conversion)} \\
& [\![H_1^{\mathrm{ops}}]\!]^{\delta_1} \mathbin{@} \langle \ell, \langle [\![V]\!], \langle V_1, [\![H_1]\!]^{\delta_1} \rangle :: [] \rangle \rangle \\
& \quad \mathbin{@} \downarrow (\cdots :: \langle \uparrow V_n, [\![H_n]\!]^{\delta_n} \rangle :: \langle \mathscr{V}_{fs}, \langle \mathscr{V}_{ret}, \mathscr{V}_{ops} \rangle \rangle :: \mathscr{W}) \\
= & \quad \text{(definition of } \downarrow) \\
& [\![H_1^{\mathrm{ops}}]\!]^{\delta_1} \mathbin{@} \langle \ell, \langle [\![V]\!], \langle V_1, [\![H_1]\!]^{\delta_1} \rangle :: [] \rangle \rangle \\
& \quad \mathbin{@} (\cdots :: \langle V_n, [\![H_n]\!]^{\delta_n} \rangle :: \langle \downarrow \mathscr{V}_{fs}, \langle \downarrow \mathscr{V}_{ret}, \downarrow \mathscr{V}_{ops} \rangle \rangle :: \downarrow \mathscr{W}) \\
\rightsquigarrow^{+} & \quad (\ell \notin BL(\mathscr{E}) \text{ and repeated application of Lemma 5})
\end{array}
$$

$$
\begin{array}{ll}
& [\![H_n^{\mathrm{ops}}]\!]^{\delta} \mathbin{@} \langle \ell, \langle [\![V]\!], \langle V_n, [\![H_n]\!]^{\delta_n} \rangle :: \cdots :: \langle V_1, [\![H_1]\!]^{\delta_1} \rangle :: [] \rangle \rangle \\
& \qquad \mathbin{@} (\langle \downarrow \mathscr{V}_{fs}, \langle \downarrow \mathscr{V}_{ret}, \downarrow \mathscr{V}_{ops} \rangle \rangle :: \downarrow \mathscr{W}) \\
\rightsquigarrow^{+} & \quad (H^{\ell} = \{\ell\, p\, r \mapsto N_{\ell}\}) \\
& \quad \mathbf{let}\ r = \mathbf{res}^{\delta}\ (\langle V_n, [\![H_n]\!]^{\delta_n} \rangle :: \cdots :: \langle V_1, [\![H_1]\!]^{\delta_1} \rangle :: [])\ \mathbf{in} \\
& \quad \mathbf{let}\ \langle fs, \langle h^{\mathrm{ret}}, h^{\mathrm{ops}} \rangle \rangle :: k' = \langle \downarrow \mathscr{V}_{fs}, \langle \downarrow \mathscr{V}_{ret}, \downarrow \mathscr{V}_{ops} \rangle \rangle :: \downarrow \mathscr{W}\ \mathbf{in} \\
& \quad ([\![N_{\ell}]\!] \mathbin{\overline{@}} (\langle \uparrow fs, \langle \uparrow h^{\mathrm{ret}}, \uparrow h^{\mathrm{ops}} \rangle \rangle \mathbin{\overline{::}} \uparrow k'))[[\![V]\!]/p] \\
\rightsquigarrow & \quad (\text{U-RES}^{\delta}\text{: there are two cases yielding different } \mathscr{R}\text{, see below}) \\
& \quad \mathbf{let}\ \langle fs, \langle h^{\mathrm{ret}}, h^{\mathrm{ops}} \rangle \rangle :: k' = \langle \downarrow \mathscr{V}_{fs}, \langle \downarrow \mathscr{V}_{ret}, \downarrow \mathscr{V}_{ops} \rangle \rangle :: \downarrow \mathscr{W}\ \mathbf{in} \\
& \quad ([\![N_{\ell}]\!] \mathbin{\overline{@}} (\langle \uparrow fs, \langle \uparrow h^{\mathrm{ret}}, \uparrow h^{\mathrm{ops}} \rangle \rangle \mathbin{\overline{::}} \uparrow k'))[[\![V]\!]/p, \mathscr{R}/r] \\
\rightsquigarrow^{+} & \quad (\text{U-SPLIT}) \\
& \quad ([\![N_{\ell}]\!] \mathbin{\overline{@}} (\langle \mathscr{V}_{fs}, \langle \mathscr{V}_{ret}, \mathscr{V}_{ops} \rangle \rangle \mathbin{\overline{::}} \uparrow \downarrow \mathscr{W}))[[\![V]\!]/p, \mathscr{R}/r] \\
= & \quad (\text{Lemma 4 (Reflect after reify)}) \\
& \quad ([\![N_{\ell}]\!] \mathbin{\overline{@}} (\langle \mathscr{V}_{fs}, \langle \mathscr{V}_{ret}, \mathscr{V}_{ops} \rangle \rangle \mathbin{\overline{::}} \mathscr{W}))[[\![V]\!]/p, \mathscr{R}/r]
\end{array}
$$

To complete the proof, we examine the resumption term $\mathscr{R}$ generated by the reduction of the $\mathbf{let}\ r = \mathbf{res}^{\delta}\ rk\ \mathbf{in}\ N$ construct. There are two cases, depending on whether the handler is deep or shallow. When the handler is deep, we have:

$$
\begin{array}{ll}
\mathscr{R} = & \lambda y k.\mathbf{let}\ \langle fs, \langle h^{\mathrm{ret}}, h^{\mathrm{ops}} \rangle \rangle :: k' = k\ \mathbf{in} \\
& \quad \mathbf{app}\ (\langle V_1, [\![H_1]\!]^{\delta_1} \rangle :: \cdots :: \langle V_n, [\![H_n]\!]^{\delta_n} \rangle :: \langle fs, \langle h^{\mathrm{ret}}, h^{\mathrm{ops}} \rangle \rangle :: k')\ y \\
= & \quad (\text{static } \beta\text{-conversion, and definition of } \downarrow) \\
& \lambda y k.\mathbf{let}\ \langle fs, \langle h^{\mathrm{ret}}, h^{\mathrm{ops}} \rangle \rangle :: k' = k\ \mathbf{in} \\
& \quad (\overline{\lambda}\, \kappa.\mathbf{app}\ (\downarrow \kappa)\ y) \\
& \qquad \mathbin{\overline{@}} ((\langle \uparrow V_1, [\![H_1]\!]^{\delta_1} \rangle \mathbin{\overline{::}} \cdots \mathbin{\overline{::}} \langle \uparrow V_n, [\![H_n]\!]^{\delta_n} \rangle \mathbin{\overline{::}} \langle \uparrow fs, \langle \uparrow h^{\mathrm{ret}}, \uparrow h^{\mathrm{ops}} \rangle \rangle \mathbin{\overline{::}} \uparrow k') \\
= & \quad (\text{definition of } [\![-]\!]) \\
& \lambda y k.\mathbf{let}\ \langle fs, \langle h^{\mathrm{ret}}, h^{\mathrm{ops}} \rangle \rangle :: k' = k\ \mathbf{in} \\
& \quad [\![\mathbf{return}\ y]\!] \mathbin{\overline{@}} ((\langle \uparrow V_1, [\![H_1]\!]^{\delta_1} \rangle \mathbin{\overline{::}} \cdots \mathbin{\overline{::}} \langle \uparrow V_n, [\![H_n]\!]^{\delta_n} \rangle \mathbin{\overline{::}} \langle \uparrow fs, \langle \uparrow h^{\mathrm{ret}}, \uparrow h^{\mathrm{ops}} \rangle \rangle \mathbin{\overline{::}} \uparrow k') \\
= & \quad (\text{Equation A 1}) \\
& \lambda y k.\mathbf{let}\ \langle fs, \langle h^{\mathrm{ret}}, h^{\mathrm{ops}} \rangle \rangle :: k' = k\ \mathbf{in} \\
& \quad [\![\mathbf{return}\ y]\!] \mathbin{\overline{@}} ([\![\mathscr{E}]\!] \mathbin{\overline{@}} (\langle \uparrow [], [\![H_n]\!]^{\delta_n} \rangle \mathbin{\overline{::}} \langle \uparrow fs, \langle \uparrow h^{\mathrm{ret}}, \uparrow h^{\mathrm{ops}} \rangle \rangle \mathbin{\overline{::}} \uparrow k'))
\end{array}
$$

When the handler is shallow, we have:

$$
\begin{array}{ll}
\mathscr{R} = & \lambda y k.\mathbf{let}\ \langle fs, \langle h^{\mathrm{ret}}, h^{\mathrm{ops}} \rangle \rangle :: k' = k\ \mathbf{in} \\
& \quad \mathbf{app}\ (\langle V_1, [\![H_1]\!]^{\delta_1} \rangle :: \cdots :: \langle V_n +\!\!+ fs, \langle h^{\mathrm{ret}}, h^{\mathrm{ops}} \rangle \rangle :: k')\ y \\
= & \quad (\text{static } \beta\text{-conversion, and definition of } \downarrow) \\
& \lambda y k.\mathbf{let}\ \langle fs, \langle h^{\mathrm{ret}}, h^{\mathrm{ops}} \rangle \rangle :: k' = k\ \mathbf{in} \\
& \quad (\overline{\lambda}\, \kappa.\mathbf{app}\ (\downarrow \kappa)\ y) \mathbin{\overline{@}} ((\langle \uparrow V_1, [\![H_1]\!]^{\delta_1} \rangle \mathbin{\overline{::}} \cdots \mathbin{\overline{::}} \langle \uparrow (V_n +\!\!+ fs), \langle \uparrow h^{\mathrm{ret}}, \uparrow h^{\mathrm{ops}} \rangle \rangle \mathbin{\overline{::}} \uparrow k') \\
= & \quad (\text{definition of } [\![-]\!]) \\
& \lambda y k.\mathbf{let}\ \langle fs, \langle h^{\mathrm{ret}}, h^{\mathrm{ops}} \rangle \rangle :: k' = k\ \mathbf{in} \\
& \quad [\![\mathbf{return}\ y]\!] \mathbin{\overline{@}} ((\langle \uparrow V_1, [\![H_1]\!]^{\delta_1} \rangle \mathbin{\overline{::}} \cdots \mathbin{\overline{::}} \langle \uparrow (V_n +\!\!+ fs), \langle \uparrow h^{\mathrm{ret}}, \uparrow h^{\mathrm{ops}} \rangle \rangle \mathbin{\overline{::}} \uparrow k') \\
= & \quad (\text{Equation A 1}) \\
& \lambda y k.\mathbf{let}\ \langle fs, \langle h^{\mathrm{ret}}, h^{\mathrm{ops}} \rangle \rangle :: k' = k\ \mathbf{in} \\
& \quad [\![\mathbf{return}\ y]\!] \mathbin{\overline{@}} ([\![\mathscr{E}]\!] \mathbin{\overline{@}} (\langle \uparrow fs, \langle \uparrow h^{\mathrm{ret}}, \uparrow h^{\mathrm{ops}} \rangle \rangle \mathbin{\overline{::}} \uparrow k'))
\end{array}
$$

□

Theorem 7 (*Simulation*)
If $M \rightsquigarrow N$ then $[\![M]\!]\ \overline{@}\ (\overline{\langle} \mathscr{V}_{fs}, \overline{\langle} \mathscr{V}_{ret}, \mathscr{V}_{ops} \overline{\rangle\rangle} :: \mathscr{W}) \rightsquigarrow^{+} [\![N]\!]\ \overline{@}\ (\overline{\langle} \mathscr{V}_{fs}, \overline{\langle} \mathscr{V}_{ret}, \mathscr{V}_{ops} \overline{\rangle\rangle} :: \mathscr{W})$.

Proof
The proof is by induction on the derivation of the reduction relation ($\rightsquigarrow$).

Case S-App: $(\lambda x^A.M)\, V \rightsquigarrow M[V/x]$.

$$
\begin{array}{ll}
 & [\![(\lambda x^A.M)\, V]\!]\ \overline{@}\ (\overline{\langle} \mathscr{V}_{fs}, \overline{\langle} \mathscr{V}_{ret}, \mathscr{V}_{ops} \overline{\rangle\rangle} :: \mathscr{W}) \\
= & \quad \text{(definition of } [\![-]\!]) \\
 & (\overline{\lambda}\kappa.\, [\![\lambda x^A.M]\!]\ \underline{@}\ [\![V]\!]\ \underline{@}\ \downarrow\kappa)\ \overline{@}\ (\overline{\langle} \mathscr{V}_{fs}, \overline{\langle} \mathscr{V}_{ret}, \mathscr{V}_{ops} \overline{\rangle\rangle} :: \mathscr{W}) \\
= & \quad \text{(static } \beta\text{-conversion)} \\
 & [\![\lambda x^A.M]\!]\ \underline{@}\ [\![V]\!]\ \underline{@}\ \downarrow(\overline{\langle} \mathscr{V}_{fs}, \overline{\langle} \mathscr{V}_{ret}, \mathscr{V}_{ops} \overline{\rangle\rangle} :: \mathscr{W}) \\
= & \quad \text{(definition of } \downarrow) \\
 & [\![\lambda x^A.M]\!]\ \underline{@}\ [\![V]\!]\ \underline{@}\ (\underline{\langle} \downarrow \mathscr{V}_{fs}, \underline{\langle} \downarrow \mathscr{V}_{ret}, \downarrow \mathscr{V}_{ops} \underline{\rangle\rangle} :: \downarrow \mathscr{W}) \\
= & \quad \text{(definition of } [\![-]\!]) \\
 & (\underline{\lambda} x k.\, \mathbf{let}\ \underline{\langle} fs, \underline{\langle} h^{\mathrm{ret}}, h^{\mathrm{ops}} \underline{\rangle\rangle} :: k' = k\ \mathbf{in} [\![M]\!]\ \overline{@}\ (\overline{\langle} \uparrow fs, \overline{\langle} \uparrow h^{\mathrm{ret}}, \uparrow h^{\mathrm{ops}} \overline{\rangle\rangle} :: \uparrow k)) \\
 & \quad \underline{@} [\![V]\!]\ \underline{@}\ (\underline{\langle} \downarrow \mathscr{V}_{fs}, \underline{\langle} \downarrow \mathscr{V}_{ret}, \downarrow \mathscr{V}_{ops} \underline{\rangle\rangle} :: \downarrow \mathscr{W}) \\
\rightsquigarrow^{+} & \quad \text{(dynamic } \beta\text{-reduction and pattern matching, and structure of continuations)} \\
 & [\![M]\!][[\![V]\!]/x]\ \overline{@}\ (\overline{\langle} \mathscr{V}_{fs}, \overline{\langle} \mathscr{V}_{ret}, \mathscr{V}_{ops} \overline{\rangle\rangle} :: \uparrow\downarrow \mathscr{W}) \\
= & \quad \text{(Lemma 2 (Substitution))} \\
 & [\![M[V/x]]\!]\ \overline{@}\ (\overline{\langle} \mathscr{V}_{fs}, \overline{\langle} \mathscr{V}_{ret}, \mathscr{V}_{ops} \overline{\rangle\rangle} :: \uparrow\downarrow \mathscr{W}) \\
= & \quad \text{(Lemma 4 (reflect after reify))} \\
 & [\![M[V/x]]\!]\ \overline{@}\ (\overline{\langle} \mathscr{V}_{fs}, \overline{\langle} \mathscr{V}_{ret}, \mathscr{V}_{ops} \overline{\rangle\rangle} :: \mathscr{W})
\end{array}
$$

Case S-TyApp: $(\Lambda\alpha^K.M)\, T \rightsquigarrow M[T/\alpha]$.

$$
\begin{array}{ll}
 & [\![(\Lambda\alpha^K.M)\, T]\!]\ \overline{@}\ (\overline{\langle} \mathscr{V}_{fs}, \overline{\langle} \mathscr{V}_{ret}, \mathscr{V}_{ops} \overline{\rangle\rangle} :: \mathscr{W}) \\
= & \quad \text{(definition of } [\![-]\!]) \\
 & (\overline{\lambda}\kappa.\, [\![\Lambda\alpha^K.M]\!]\ \underline{@}\ \underline{\langle\rangle}\ \underline{@}\ \downarrow\kappa)\ \overline{@}\ (\overline{\langle} \mathscr{V}_{fs}, \overline{\langle} \mathscr{V}_{ret}, \mathscr{V}_{ops} \overline{\rangle\rangle} :: \mathscr{W}) \\
= & \quad \text{(static } \beta\text{-conversion)} \\
 & [\![\Lambda\alpha^K.M]\!]\ \underline{@}\ [\![V]\!]\ \underline{@}\ \underline{\langle\rangle}\ \underline{@}\ \downarrow(\overline{\langle} \mathscr{V}_{fs}, \overline{\langle} \mathscr{V}_{ret}, \mathscr{V}_{ops} \overline{\rangle\rangle} :: \mathscr{W}) \\
= & \quad \text{(definition of } \downarrow) \\
 & [\![\Lambda\alpha^K.M]\!]\ \underline{@}\ [\![V]\!]\ \underline{@}\ \underline{\langle\rangle}\ \underline{@}\ (\underline{\langle} \downarrow \mathscr{V}_{fs}, \underline{\langle} \downarrow \mathscr{V}_{ret}, \downarrow \mathscr{V}_{ops} \underline{\rangle\rangle} :: \downarrow \mathscr{W}) \\
= & \quad \text{(definition of } [\![-]\!]) \\
 & (\underline{\lambda} x k.\, \mathbf{let}\ \underline{\langle} fs, \underline{\langle} h^{\mathrm{ret}}, h^{\mathrm{ops}} \underline{\rangle\rangle} :: k' = k\ \mathbf{in}\ [\![M]\!]\ \overline{@}\ (\overline{\langle} \uparrow fs, \overline{\langle} \uparrow h^{\mathrm{ret}}, \uparrow h^{\mathrm{ops}} \overline{\rangle\rangle} :: \uparrow k)) \\
 & \quad \underline{@}\underline{\langle\rangle}\ \underline{@}\ (\underline{\langle} \downarrow \mathscr{V}_{fs}, \underline{\langle} \downarrow \mathscr{V}_{ret}, \downarrow \mathscr{V}_{ops} \underline{\rangle\rangle} :: \downarrow \mathscr{W}) \\
\rightsquigarrow^{+} & \quad \text{(dynamic } \beta\text{-reduction and pattern matching, and structure of continuations)} \\
 & [\![M]\!]\ \overline{@}\ (\overline{\langle} \mathscr{V}_{fs}, \overline{\langle} \mathscr{V}_{ret}, \mathscr{V}_{ops} \overline{\rangle\rangle} :: \uparrow\downarrow \mathscr{W}) \\
= & \quad \text{(Lemma 8 (Type Erasure))} \\
 & [\![M[T/\alpha]]\!]\ \overline{@}\ (\overline{\langle} \mathscr{V}_{fs}, \overline{\langle} \mathscr{V}_{ret}, \mathscr{V}_{ops} \overline{\rangle\rangle} :: \uparrow\downarrow \mathscr{W}) \\
= & \quad \text{(Lemma 4 (reflect after reify))} \\
 & [\![M[T/\alpha]]\!]\ \overline{@}\ (\overline{\langle} \mathscr{V}_{fs}, \overline{\langle} \mathscr{V}_{ret}, \mathscr{V}_{ops} \overline{\rangle\rangle} :: \mathscr{W})
\end{array}
$$

Case S-Rec: $(\mathbf{rec}\, g\, x.M)\, V \rightsquigarrow M[(\mathbf{rec}\, g\, x.M)/g, V/x]$. Similar to the previous two cases.

Case S-SPLIT: $\mathbf{let}\ \langle \ell = x; y\rangle = \langle \ell = V; W\rangle\ \mathbf{in}\ N \rightsquigarrow N[V/x, W/y]$.

$$
\begin{array}{ll}
 & [\![\mathbf{let}\ \langle \ell = x; y\rangle = \langle \ell = V; W\rangle\ \mathbf{in}\ N]\!]\ \overline{@}\ (\overline{\langle}\mathscr{V}_{fs}, \overline{\langle}\mathscr{V}_{ret}, \mathscr{V}_{ops}\overline{\rangle\rangle} :: \mathscr{W}) \\
= & \quad \text{(definition of } [\![-]\!]) \\
 & (\overline{\lambda}\kappa.\mathbf{let}\ \underline{\langle}\ell, \underline{\langle} x, y\underline{\rangle\rangle} = \langle \ell, \underline{\langle}[\![V]\!], [\![W]\!]\underline{\rangle}\rangle\ \mathbf{in}\ [\![N]\!]\ \overline{@}\ \kappa)\ \overline{@}\ (\overline{\langle}\mathscr{V}_{fs}, \overline{\langle}\mathscr{V}_{ret}, \mathscr{V}_{ops}\overline{\rangle\rangle} :: \mathscr{W}) \\
= & \quad \text{(static } \beta\text{-conversion)} \\
 & \mathbf{let}\ \underline{\langle}\ell, \underline{\langle} x, y\underline{\rangle\rangle} = \langle \ell, \underline{\langle}[\![V]\!], [\![W]\!]\underline{\rangle}\rangle\ \mathbf{in}\ [\![N]\!]\ \overline{@}\ (\overline{\langle}\mathscr{V}_{fs}, \overline{\langle}\mathscr{V}_{ret}, \mathscr{V}_{ops}\overline{\rangle\rangle} :: \mathscr{W}) \\
\rightsquigarrow^{+} & \quad \text{(U-SPLIT)} \\
 & ([\![N]\!]\ \overline{@}\ (\overline{\langle}\mathscr{V}_{fs}, \overline{\langle}\mathscr{V}_{ret}, \mathscr{V}_{ops}\overline{\rangle\rangle} :: \mathscr{W}))[[\![V]\!]/x, [\![W]\!]/y] \\
= & \quad \text{(Lemma 2 (Substitution))} \\
 & [\![N[V/x, W/y]]\!]\ \overline{@}\ (\overline{\langle}\mathscr{V}_{fs}, \overline{\langle}\mathscr{V}_{ret}, \mathscr{V}_{ops}\overline{\rangle\rangle} :: \mathscr{W})
\end{array}
$$

Case S-CASE$_1$ and S-CASE$_2$: Similar to the previous case.

Case S-LET: $\mathbf{let}\ x \leftarrow \mathbf{return}\ V\ \mathbf{in}\ N \rightsquigarrow N[V/x]$.

$$
\begin{array}{ll}
 & [\![\mathbf{let}\ x \leftarrow \mathbf{return}\ V\ \mathbf{in}\ N]\!]\ \overline{@}\ (\overline{\langle}\mathscr{V}_{fs}, \overline{\langle}\mathscr{V}_{ret}, \mathscr{V}_{ops}\overline{\rangle\rangle} :: \mathscr{W}) \\
= & \quad \text{(definition of } [\![-]\!]) \\
 & (\overline{\lambda}\overline{\langle}\theta, \overline{\langle}\chi^{\mathrm{ret}}, \chi^{\mathrm{ops}}\overline{\rangle\rangle} :: \kappa. \\
 & [\![\mathbf{return}\ V]\!]\ \overline{@}\ (\overline{\langle}\uparrow((\underline{\lambda} x k.\mathbf{let}\ \underline{\langle} fs, \underline{\langle} h^{\mathrm{ret}}, h^{\mathrm{ops}}\underline{\rangle\rangle} :: k' = k\ \mathbf{in} \\
 & \qquad\qquad [\![N]\!]\ \overline{@}\ (\overline{\langle}\uparrow fs, \overline{\langle}\uparrow h^{\mathrm{ret}}, \uparrow h^{\mathrm{ops}}\overline{\rangle\rangle} :: \uparrow k')) :: \downarrow\theta), \overline{\langle}\chi^{\mathrm{ret}}, \chi^{\mathrm{ops}}\overline{\rangle\rangle} :: \kappa)) \\
 & \quad \overline{@}(\overline{\langle}\mathscr{V}_{fs}, \overline{\langle}\mathscr{V}_{ret}, \mathscr{V}_{ops}\overline{\rangle\rangle} :: \mathscr{W}) \\
= & \quad \text{(static } \beta\text{-conversion)} \\
 & [\![\mathbf{return}\ V]\!]\ \overline{@}\ (\overline{\langle}\uparrow((\underline{\lambda} x k.\mathbf{let}\ \underline{\langle} fs, \underline{\langle} h^{\mathrm{ret}}, h^{\mathrm{ops}}\underline{\rangle\rangle} :: k' = k\ \mathbf{in} \\
 & \qquad\qquad [\![N]\!]\ \overline{@}\ (\overline{\langle}\uparrow fs, \overline{\langle}\uparrow h^{\mathrm{ret}}, \uparrow h^{\mathrm{ops}}\overline{\rangle\rangle} :: \uparrow k')) :: \downarrow\mathscr{V}_{fs}), \\
 & \qquad \overline{\langle}\mathscr{V}_{ret}, \mathscr{V}_{ops}\overline{\rangle\rangle} :: \mathscr{W}) \\
= & \quad \text{(definition of } [\![-]\!]) \\
 & (\overline{\lambda}\kappa.\mathbf{app}\ (\downarrow\kappa)\ [\![V]\!])\ \overline{@}\ (\overline{\langle}\uparrow((\underline{\lambda} x k.\mathbf{let}\ \underline{\langle} fs, \underline{\langle} h^{\mathrm{ret}}, h^{\mathrm{ops}}\underline{\rangle\rangle} :: k' = k\ \mathbf{in} \\
 & \qquad\qquad [\![N]\!]\ \overline{@}\ (\overline{\langle}\uparrow fs, \overline{\langle}\uparrow h^{\mathrm{ret}}, \uparrow h^{\mathrm{ops}}\overline{\rangle\rangle} :: \uparrow k')) :: \downarrow\mathscr{V}_{fs}), \\
 & \qquad\qquad \overline{\langle}\mathscr{V}_{ret}, \mathscr{V}_{ops}\overline{\rangle\rangle} :: \mathscr{W}) \\
= & \quad \text{(static } \beta\text{-conversion)} \\
 & \mathbf{app}\ (\downarrow(\overline{\langle}\uparrow((\underline{\lambda} x k.\mathbf{let}\ \underline{\langle} fs, \underline{\langle} h^{\mathrm{ret}}, h^{\mathrm{ops}}\underline{\rangle\rangle} :: k' = k\ \mathbf{in} \\
 & \qquad\qquad [\![N]\!]\ \overline{@}\ (\overline{\langle}\uparrow fs, \overline{\langle}\uparrow h^{\mathrm{ret}}, \uparrow h^{\mathrm{ops}}\overline{\rangle\rangle} :: \uparrow k')) :: \downarrow\mathscr{V}_{fs}), \overline{\langle}\mathscr{V}_{ret}, \mathscr{V}_{ops}\overline{\rangle\rangle} :: \mathscr{W}))\ [\![V]\!] \\
= & \quad \text{(definition of } \downarrow) \\
 & \mathbf{app}\ (\underline{\langle}(\underline{\lambda} x k.\mathbf{let}\ \underline{\langle} fs, \underline{\langle} h^{\mathrm{ret}}, h^{\mathrm{ops}}\underline{\rangle\rangle} :: k' = k\ \mathbf{in} \\
 & \qquad [\![N]\!]\ \overline{@}\ (\overline{\langle}\uparrow fs, \overline{\langle}\uparrow h^{\mathrm{ret}}, \uparrow h^{\mathrm{ops}}\overline{\rangle\rangle} :: \uparrow k')) :: \downarrow\mathscr{V}_{fs}, \underline{\langle}\downarrow\mathscr{V}_{ret}, \downarrow\mathscr{V}_{ops}\underline{\rangle\rangle} :: \downarrow\mathscr{W}))\ [\![V]\!] \\
\rightsquigarrow & \quad \text{(U-KAPPCONS)} \\
 & (\underline{\lambda} x k.\mathbf{let}\ \underline{\langle} fs, \underline{\langle} h^{\mathrm{ret}}, h^{\mathrm{ops}}\underline{\rangle\rangle} :: k' = k\ \mathbf{in} \\
 & \quad [\![N]\!]\ \overline{@}\ (\overline{\langle}\uparrow fs, \overline{\langle}\uparrow h^{\mathrm{ret}}, \uparrow h^{\mathrm{ops}}\overline{\rangle\rangle} :: \uparrow k'))\ \underline{@}\ [\![V]\!]\ \underline{@}\ (\underline{\langle}\downarrow\mathscr{V}_{fs}, \underline{\langle}\downarrow\mathscr{V}_{ret}, \downarrow\mathscr{V}_{ops}\underline{\rangle\rangle} :: \downarrow\mathscr{W}) \\
\rightsquigarrow^{+} & \quad \text{(U-APP, U-SPLIT)} \\
 & [\![N]\!][[\![V]\!]/x]\ \overline{@}\ (\overline{\langle}\mathscr{V}_{fs}, \overline{\langle}\mathscr{V}_{ret}, \mathscr{V}_{ops}\overline{\rangle\rangle} :: \uparrow\downarrow\mathscr{W}) \\
= & \quad \text{(Lemma 4 (reflect after reify))} \\
 & [\![N]\!][[\![V]\!]/x]\ \overline{@}\ (\overline{\langle}\mathscr{V}_{fs}, \overline{\langle}\mathscr{V}_{ret}, \mathscr{V}_{ops}\overline{\rangle\rangle} :: \mathscr{W}) \\
= & \quad \text{(Lemma 2 (substitution))} \\
 & [\![N[V/x]]\!]\ \overline{@}\ (\overline{\langle}\mathscr{V}_{fs}, \overline{\langle}\mathscr{V}_{ret}, \mathscr{V}_{ops}\overline{\rangle\rangle} :: \mathscr{W})
\end{array}
$$

Case S-RET: $\mathbf{handle}^{\delta}\ (\mathbf{return}\ V)\ \mathbf{with}\ H \rightsquigarrow N[V/x]$, where $H^{\mathrm{ret}} = \{\mathbf{return}\ x \mapsto N\}$.

$$
\begin{array}{ll}
 & [\![\mathbf{handle}^{\delta}\ (\mathbf{return}\ V)\ \mathbf{with}\ H]\!]\ \overline{@}\ (\overline{\langle}\mathscr{V}_{fs}, \overline{\langle}\mathscr{V}_{ret}, \mathscr{V}_{ops}\overline{\rangle\rangle} :: \mathscr{W}) \\
= & \quad \text{(definition of } [\![-]\!]) \\
 & (\overline{\lambda}\kappa.[\![\mathbf{return}\ V]\!]\ \overline{@}\ (\overline{\langle}\uparrow\underline{[]}, [\![H]\!]^{\delta}\overline{\rangle} :: \kappa))\ \overline{@}\ (\overline{\langle}\mathscr{V}_{fs}, \overline{\langle}\mathscr{V}_{ret}, \mathscr{V}_{ops}\overline{\rangle\rangle} :: \mathscr{W}) \\
= & \quad \text{(static } \beta\text{-conversion)} \\
 & [\![\mathbf{return}\ V]\!]\ \overline{@}\ (\overline{\langle}\uparrow\underline{[]}, [\![H]\!]^{\delta}\overline{\rangle} :: \overline{\langle}\mathscr{V}_{fs}, \overline{\langle}\mathscr{V}_{ret}, \mathscr{V}_{ops}\overline{\rangle\rangle} :: \mathscr{W}) \\
= & \quad \text{(definition of } [\![-]\!]) \\
 & (\overline{\lambda}\kappa.\mathbf{app}\ (\downarrow\kappa)\ [\![V]\!])\ \overline{@}\ (\overline{\langle}\uparrow\underline{[]}, [\![H]\!]^{\delta}\overline{\rangle} :: \overline{\langle}\mathscr{V}_{fs}, \overline{\langle}\mathscr{V}_{ret}, \mathscr{V}_{ops}\overline{\rangle\rangle} :: \mathscr{W}) \\
= & \quad \text{(static } \beta\text{-conversion)} \\
 & \mathbf{app}\ (\downarrow(\overline{\langle}\uparrow\underline{[]}, [\![H]\!]^{\delta}\overline{\rangle} :: \overline{\langle}\mathscr{V}_{fs}, \overline{\langle}\mathscr{V}_{ret}, \mathscr{V}_{ops}\overline{\rangle\rangle} :: \mathscr{W}))\ [\![V]\!] \\
= & \quad \text{(definition of } [\![H]\!]^{\delta} \text{ and } \downarrow) \\
 & \mathbf{app}\ (\underline{\langle[]}, \underline{\langle}[\![H^{\mathrm{ret}}]\!], [\![H^{\mathrm{ops}}]\!]^{\delta}\underline{\rangle\rangle} :: \underline{\langle}\downarrow\mathscr{V}_{fs}, \underline{\langle}\downarrow\mathscr{V}_{ret}, \downarrow\mathscr{V}_{ops}\underline{\rangle\rangle} :: \downarrow\mathscr{W})\ [\![V]\!] \\
\rightsquigarrow & \quad \text{(U-KAPPNIL)} \\
 & [\![H^{\mathrm{ret}}]\!]\ \underline{@}\ [\![V]\!]\ \underline{@}\ (\underline{\langle}\downarrow\mathscr{V}_{fs}, \underline{\langle}\downarrow\mathscr{V}_{ret}, \downarrow\mathscr{V}_{ops}\underline{\rangle\rangle} :: \downarrow\mathscr{W}) \\
= & \quad \text{(definition of } [\![-]\!]) \\
 & (\underline{\lambda}x\,k.\mathbf{let}\ \underline{\langle}fs, \underline{\langle}h^{\mathrm{ret}}, h^{\mathrm{ops}}\underline{\rangle\rangle} :: k' = k\ \mathbf{in}\ [\![N]\!]\ \overline{@}\ (\overline{\langle}\uparrow fs, \overline{\langle}\uparrow h^{\mathrm{ret}}, \uparrow h^{\mathrm{ops}}\overline{\rangle\rangle} :: \uparrow k')) \\
 & \qquad \underline{@}\ [\![V]\!]\ \underline{@}\ (\underline{\langle}\downarrow\mathscr{V}_{fs}, \underline{\langle}\downarrow\mathscr{V}_{ret}, \downarrow\mathscr{V}_{ops}\underline{\rangle\rangle} :: \downarrow\mathscr{W}) \\
\rightsquigarrow^{+} & \quad \text{(U-APP, U-SPLIT)} \\
 & [\![N]\!][[\![V]\!]/x]\ \overline{@}\ (\overline{\langle}\mathscr{V}_{fs}, \overline{\langle}\mathscr{V}_{ret}, \mathscr{V}_{ops}\overline{\rangle\rangle} :: \uparrow\downarrow\mathscr{W}) \\
= & \quad \text{(Lemma 4 (reflect after reify))} \\
 & [\![N]\!][[\![V]\!]/x]\ \overline{@}\ (\overline{\langle}\mathscr{V}_{fs}, \overline{\langle}\mathscr{V}_{ret}, \mathscr{V}_{ops}\overline{\rangle\rangle} :: \mathscr{W}) \\
= & \quad \text{(Lemma 2 (substitution))} \\
 & [\![N[V/x]]\!]\ \overline{@}\ (\overline{\langle}\mathscr{V}_{fs}, \overline{\langle}\mathscr{V}_{ret}, \mathscr{V}_{ops}\overline{\rangle\rangle} :: \mathscr{W})
\end{array}
$$

Case S-OP: $\mathbf{handle}\ \mathscr{E}[\mathbf{do}\ \ell\ V]\ \mathbf{with}\ H \rightsquigarrow N_{\ell}[V/p, \lambda y.\mathbf{handle}\ \mathscr{E}[\mathbf{return}\ y]\ \mathbf{with}\ H/r]$, where $\ell \notin BL(\mathscr{E})$ and $H^{\ell} = \{\ell\ p\ r \mapsto N_{\ell}\}$.

$$
\begin{array}{ll}
 & [\![\mathbf{handle}\ \mathscr{E}[\mathbf{do}\ \ell\ V]\ \mathbf{with}\ H]\!]\ \overline{@}\ (\overline{\langle}\mathscr{V}_{fs}, \overline{\langle}\mathscr{V}_{ret}, \mathscr{V}_{ops}\overline{\rangle\rangle} :: \mathscr{W}) \\
= & \quad \text{(definition of } [\![-]\!]) \\
 & (\overline{\lambda}\kappa.[\![\mathscr{E}[\mathbf{do}\ \ell\ V]]\!]\ \overline{@}\ (\overline{\langle}\uparrow\underline{[]}, \overline{\langle}\uparrow[\![H^{\mathrm{ret}}]\!], \uparrow[\![H^{\mathrm{ops}}]\!]\overline{\rangle\rangle} :: \kappa))\ \overline{@}\ (\overline{\langle}\mathscr{V}_{fs}, \overline{\langle}\mathscr{V}_{ret}, \mathscr{V}_{ops}\overline{\rangle\rangle} :: \mathscr{W}) \\
= & \quad \text{(static } \beta\text{-conversion)} \\
 & [\![\mathscr{E}[\mathbf{do}\ \ell\ V]]\!]\ \overline{@}\ (\overline{\langle}\uparrow\underline{[]}, \overline{\langle}\uparrow[\![H^{\mathrm{ret}}]\!], \uparrow[\![H^{\mathrm{ops}}]\!]\overline{\rangle\rangle} :: \overline{\langle}\mathscr{V}_{fs}, \overline{\langle}\mathscr{V}_{ret}, \mathscr{V}_{ops}\overline{\rangle\rangle} :: \mathscr{W}) \\
= & \quad \text{(Lemma 3 (Decomposition))} \\
 & [\![\mathbf{do}\ \ell\ V]\!]\ \overline{@}\ ([\![\mathscr{E}]\!]\ \overline{@}\ (\overline{\langle}\uparrow\underline{[]}, \overline{\langle}\uparrow[\![H^{\mathrm{ret}}]\!], \uparrow[\![H^{\mathrm{ops}}]\!]\overline{\rangle\rangle} :: \overline{\langle}\mathscr{V}_{fs}, \overline{\langle}\mathscr{V}_{ret}, \mathscr{V}_{ops}\overline{\rangle\rangle} :: \mathscr{W})) \\
\rightsquigarrow^{+} & \quad \text{(Lemma 6 (Handling))} \\
 & [\![N_{\ell}]\!][[\![V]\!]/p, \underline{\lambda}y\,k.\mathbf{let}\ \underline{\langle}fs, \underline{\langle}h^{\mathrm{ret}}, h^{\mathrm{ops}}\underline{\rangle\rangle} :: k' = k\ \mathbf{in} \\
 & \qquad\qquad [\![\mathbf{return}\ y]\!]\ \overline{@}\ ([\![\mathscr{E}]\!]\ \overline{@}\ (\overline{\langle}\uparrow\underline{[]}, [\![H]\!]\overline{\rangle} :: \overline{\langle}\uparrow fs, \overline{\langle}\uparrow h^{\mathrm{ret}}, \uparrow h^{\mathrm{ops}}\overline{\rangle\rangle} :: \uparrow k'))/r] \\
 & \quad \overline{@}\ (\overline{\langle}\mathscr{V}_{fs}, \overline{\langle}\mathscr{V}_{ret}, \mathscr{V}_{ops}\overline{\rangle\rangle} :: \mathscr{W}) \\
= & \quad \text{(Lemma 3 (Decomposition))} \\
 & [\![N_{\ell}]\!][[\![V]\!]/p, \underline{\lambda}y\,k.\mathbf{let}\ \underline{\langle}fs, \underline{\langle}h^{\mathrm{ret}}, h^{\mathrm{ops}}\underline{\rangle\rangle} :: k' = k\ \mathbf{in} \\
 & \qquad\qquad [\![\mathscr{E}[\mathbf{return}\ y]]\!]\ \overline{@}\ (\overline{\langle}\uparrow\underline{[]}, [\![H]\!]\overline{\rangle} :: \overline{\langle}\uparrow fs, \overline{\langle}\uparrow h^{\mathrm{ret}}, \uparrow h^{\mathrm{ops}}\overline{\rangle\rangle} :: \uparrow k')/r] \\
 & \quad \overline{@}\ (\overline{\langle}\mathscr{V}_{fs}, \overline{\langle}\mathscr{V}_{ret}, \mathscr{V}_{ops}\overline{\rangle\rangle} :: \mathscr{W}) \\
= & \quad \text{(static } \beta\text{-conversion and definition of } [\![-]\!])
\end{array}
$$

$$
\begin{array}{ll}
& [\![N_\ell]\!][[\![V]\!]/p, \underline{\lambda} y\, k.\mathbf{let}\ \underline{\langle} fs, \underline{\langle} h^{\mathrm{ret}}, h^{\mathrm{ops}} \underline{\rangle\rangle} :: k' = k\ \mathbf{in} \\
& \qquad [\![\mathbf{handle}\ \mathscr{E}[\mathbf{return}\ y]\ \mathbf{with}\ H]\!]\ \overline{@}\ (\overline{\langle} \uparrow fs, \overline{\langle} \uparrow h^{\mathrm{ret}}, \uparrow h^{\mathrm{ops}} \overline{\rangle\rangle} :: \uparrow k')/r] \\
& \quad \overline{@}\,(\overline{\langle} \mathscr{V}_{fs}, \overline{\langle} \mathscr{V}_{ret}, \mathscr{V}_{ops} \overline{\rangle\rangle} :: \mathscr{W}) \\
= & \quad \text{(definition of } [\![-]\!]) \\
& [\![N_\ell]\!][[\![V]\!]/p, [\![\lambda y.\mathbf{handle}\ \mathscr{E}[\mathbf{return}\ y]\ \mathbf{with}\ H]\!]/r]\ \overline{@}\ (\overline{\langle} \mathscr{V}_{fs}, \overline{\langle} \mathscr{V}_{ret}, \mathscr{V}_{ops} \overline{\rangle\rangle} :: \mathscr{W}) \\
= & \quad \text{(Lemma 2 (Substitution))} \\
& [\![N_\ell[V/p, \lambda y.\mathbf{handle}\ \mathscr{E}[\mathbf{return}\ y]\ \mathbf{with}\ H/r]]\!]\ \overline{@}\ (\overline{\langle} \mathscr{V}_{fs}, \overline{\langle} \mathscr{V}_{ret}, \mathscr{V}_{ops} \overline{\rangle\rangle} :: \mathscr{W})
\end{array}
$$

Case S-Op†: $\mathbf{handle}^\dagger\ \mathscr{E}[\mathbf{do}\ \ell\ V]\ \mathbf{with}\ H \rightsquigarrow N_\ell[V/p, \lambda y.\mathscr{E}[\mathbf{return}\ y]/r]$, where $\ell \notin BL(\mathscr{E})$ and $H^\ell = \{\ell\ p\ r \mapsto N_\ell\}$.

$$
\begin{array}{ll}
& [\![\mathbf{handle}^\dagger\ \mathscr{E}[\mathbf{do}\ \ell\ V]\ \mathbf{with}\ H]\!]\ \overline{@}\ (\overline{\langle} \mathscr{V}_{fs}, \overline{\langle} \mathscr{V}_{ret}, \mathscr{V}_{ops} \overline{\rangle\rangle} :: \mathscr{W}) \\
= & \quad \text{(definition of } [\![-]\!]) \\
& (\overline{\lambda}\, \kappa.[\![\mathscr{E}[\mathbf{do}\ \ell\ V]]\!]\ \overline{@}\ (\overline{\langle} \uparrow \underline{[]}, \overline{\langle} \uparrow [\![H^{\mathrm{ret}}]\!], \uparrow [\![H^{\mathrm{ops}}]\!]^\dagger \overline{\rangle\rangle} :: \kappa))\ \overline{@}\ (\overline{\langle} \mathscr{V}_{fs}, \overline{\langle} \mathscr{V}_{ret}, \mathscr{V}_{ops} \overline{\rangle\rangle} :: \mathscr{W}) \\
= & \quad \text{(static } \beta\text{-conversion)} \\
& [\![\mathscr{E}[\mathbf{do}\ \ell\ V]]\!]\ \overline{@}\ (\overline{\langle} \uparrow \underline{[]}, \overline{\langle} \uparrow [\![H^{\mathrm{ret}}]\!], \uparrow [\![H^{\mathrm{ops}}]\!]^\dagger \overline{\rangle\rangle} :: \overline{\langle} \mathscr{V}_{fs}, \overline{\langle} \mathscr{V}_{ret}, \mathscr{V}_{ops} \overline{\rangle\rangle} :: \mathscr{W}) \\
= & \quad \text{(Lemma 3 (Decomposition))} \\
& [\![\mathbf{do}\ \ell\ V]\!]\ \overline{@}\ ([\![\mathscr{E}]\!]\ \overline{@}\ (\overline{\langle} \uparrow \underline{[]}, \overline{\langle} \uparrow [\![H^{\mathrm{ret}}]\!], \uparrow [\![H^{\mathrm{ops}}]\!]^\dagger \overline{\rangle\rangle} :: \overline{\langle} \mathscr{V}_{fs}, \overline{\langle} \mathscr{V}_{ret}, \mathscr{V}_{ops} \overline{\rangle\rangle} :: \mathscr{W})) \\
\rightsquigarrow^+ & \quad \text{(Lemma 6 (Handling))} \\
& [\![N_\ell]\!][[\![V]\!]/p, \underline{\lambda} y\, k.\mathbf{let}\ \underline{\langle} fs, \underline{\langle} h^{\mathrm{ret}}, h^{\mathrm{ops}} \underline{\rangle\rangle} :: k' = k\ \mathbf{in} \\
& \qquad [\![\mathbf{return}\ y]\!]\ \overline{@}\ ([\![\mathscr{E}]\!]\ \overline{@}\ (:: \overline{\langle} \uparrow fs, \overline{\langle} \uparrow h^{\mathrm{ret}}, \uparrow h^{\mathrm{ops}} \overline{\rangle\rangle} :: \uparrow k'))/r] \\
& \quad \overline{@}\,(\overline{\langle} \mathscr{V}_{fs}, \overline{\langle} \mathscr{V}_{ret}, \mathscr{V}_{ops} \overline{\rangle\rangle} :: \mathscr{W}) \\
= & \quad \text{(Lemma 3 (Decomposition))} \\
& [\![N_\ell]\!][[\![V]\!]/p, \underline{\lambda} y\, k.\mathbf{let}\ \underline{\langle} fs, \underline{\langle} h^{\mathrm{ret}}, h^{\mathrm{ops}} \underline{\rangle\rangle} :: k' = k\ \mathbf{in} \\
& \qquad [\![\mathscr{E}[\mathbf{return}\ y]]\!]\ \overline{@}\ (\overline{\langle} \uparrow fs, \overline{\langle} \uparrow h^{\mathrm{ret}}, \uparrow h^{\mathrm{ops}} \overline{\rangle\rangle} :: \uparrow k')/r] \\
& \quad \overline{@}\,(\overline{\langle} \mathscr{V}_{fs}, \overline{\langle} \mathscr{V}_{ret}, \mathscr{V}_{ops} \overline{\rangle\rangle} :: \mathscr{W}) \\
= & \quad \text{(definition of } [\![-]\!]) \\
& [\![N_\ell]\!][[\![V]\!]/p, [\![\lambda y.\mathscr{E}[\mathbf{return}\ y]]\!]/r]\ \overline{@}\ (\overline{\langle} \mathscr{V}_{fs}, \overline{\langle} \mathscr{V}_{ret}, \mathscr{V}_{ops} \overline{\rangle\rangle} :: \mathscr{W}) \\
= & \quad \text{(Lemma 2 (Substitution))} \\
& [\![N_\ell[V/p, \lambda y.\mathscr{E}[\mathbf{return}\ y]/r]]\!]\ \overline{@}\ (\overline{\langle} \mathscr{V}_{fs}, \overline{\langle} \mathscr{V}_{ret}, \mathscr{V}_{ops} \overline{\rangle\rangle} :: \mathscr{W})
\end{array}
$$

□

B Proof of Simulation for Parameterised Handlers by Deep Handlers

Theorem 9 (*Simulation of Parameterised Handlers by Deep Handlers*)
If $M \rightsquigarrow N$ then $\mathscr{P}[\![M]\!] \rightsquigarrow^{+}_{\text{cong}} \mathscr{P}[\![N]\!]$.

Proof of Theorem 9
Proof by induction on M. We show only the two interesting cases.

Case $M = \mathbf{handle}^{\ddagger}\ \mathbf{return}\ V\ \mathbf{with}\ (q.\ H)(W) \rightsquigarrow$
$N[V/x, W/q]$, where $H^{\text{ret}} = \{\mathbf{return}\ x \mapsto N\}$.

$$
\begin{array}{ll}
 & \mathscr{P}[\![\mathbf{handle}^{\ddagger}\ \mathbf{return}\ V\ \mathbf{with}\ (q.\ H)(W)]\!] \\
= & \quad \text{(definition of } \mathscr{P}[\![-]\!]) \\
 & (\mathbf{handle}\ \mathbf{return}\ \mathscr{P}[\![V]\!]\ \mathbf{with}\ \mathscr{P}[\![H]\!]_q)\ \mathscr{P}[\![W]\!] \\
\rightsquigarrow & \quad (\text{S-RET with } \mathscr{P}[\![H^{\text{ret}}]\!]_q = \{\mathbf{return}\ x \mapsto \lambda q.\mathscr{P}[\![N]\!]\}) \\
 & (\lambda q.\mathscr{P}[\![N]\!][\mathscr{P}[\![V]\!]/x])\ \mathscr{P}[\![W]\!] \\
\rightsquigarrow & \quad (\text{S-APP}) \\
 & \mathscr{P}[\![N]\!][\mathscr{P}[\![V]\!]/x, \mathscr{P}[\![W]\!]/q] \\
= & \\
 & \mathscr{P}[\![N[V/x, W/q]]\!]
\end{array}
$$

Case $M = \mathbf{handle}^{\ddagger}\ \mathscr{E}[\mathbf{do}\ \ell\ V]\ \mathbf{with}\ (q.\ H)(W) \rightsquigarrow$
$N[V/p, W/q, \lambda\langle x, q'\rangle.\mathbf{handle}^{\ddagger}\ \mathscr{E}[\mathbf{return}\ x]\ \mathbf{with}\ (q.\ H)(q')/r]$, where $H^{\ell} = \{\ell\ p\ r \mapsto N\}$.

$$
\begin{array}{ll}
 & \mathscr{P}[\![\mathbf{handle}^{\ddagger}\ \mathscr{E}[\mathbf{do}\ \ell\ V]\ \mathbf{with}\ (q.\ H)(W)]\!] \\
= & \quad \text{(definition of } \mathscr{P}[\![-]\!]) \\
 & (\mathbf{handle}\ \mathscr{E}[\mathbf{do}\ \ell\ \mathscr{P}[\![V]\!]]\ \mathbf{with}\ \mathscr{P}[\![H]\!]_q)\ \mathscr{P}[\![W]\!] \\
\rightsquigarrow & \quad (\text{S-OP with } \mathscr{P}[\![H^{\ell}]\!]_q = \{\ell\ p\ r \mapsto \lambda q.\mathbf{let}\ r' \leftarrow \lambda\langle x, q'\rangle.r\ x\ q\ \mathbf{in}\ \mathscr{P}[\![N]\!][r'/r]\}) \\
 & ((\lambda q.\mathbf{let}\ r' \leftarrow \lambda\langle x, q'\rangle.r\ x\ q\ \mathbf{in} \\
 & \qquad \mathscr{P}[\![N]\!][r'/r])[\mathscr{P}[\![V]\!]/p, \lambda x.\mathbf{handle}\ \mathscr{E}[\mathbf{return}\ x]\ \mathbf{with}\ \mathscr{P}[\![H]\!]_q)\ \mathscr{P}[\![W]\!] \\
= & \quad \text{(definition of } [-]) \\
 & (\lambda q.\mathbf{let}\ r' \leftarrow \lambda\langle x, q'\rangle.(\lambda x.\mathbf{handle}\ \mathscr{E}[\mathbf{return}\ x]\ \mathbf{with}\ \mathscr{P}[\![H]\!]_q)\ x\ q'\ \mathbf{in} \\
 & \qquad \mathscr{P}[\![N]\!][\mathscr{P}[\![V]\!]/p, r'/r]) \\
\rightsquigarrow & \quad (\text{S-APP}) \\
 & \mathbf{let}\ r' \leftarrow \lambda\langle x, q'\rangle.r\ x\ q'\ \mathbf{in}\ \mathscr{P}[\![N]\!][r'/r, \mathscr{P}[\![V]\!]/p, \mathscr{P}[\![V]\!]/q] \\
\rightsquigarrow & \quad (\text{S-LET}) \\
 & \mathscr{P}[\![N]\!][\mathscr{P}[\![V]\!]/p, \mathscr{P}[\![W]\!]/q, \\
 & \qquad \lambda\langle x, q'\rangle.(\lambda x.\mathbf{handle}\ \mathscr{E}[\mathbf{return}\ x]\ \mathbf{with}\ \mathscr{P}[\![H]\!]_q)\ x\ q'/r] \\
\rightsquigarrow_{\text{cong}} & \quad (\text{S-APP}) \\
 & \mathscr{P}[\![N]\!][\mathscr{P}[\![V]\!]/p, \mathscr{P}[\![W]\!]/q, \lambda\langle x, q'\rangle.(\mathbf{handle}\ \mathscr{E}[\mathbf{return}\ x]\ \mathbf{with}\ \mathscr{P}[\![H]\!]_q)\ q'/r] \\
= & \quad \text{(definition of } \mathscr{P}[\![-]\!]) \\
 & \mathscr{P}[\![N]\!][\mathscr{P}[\![V]\!]/p, \mathscr{P}[\![W]\!]/q, \lambda\langle x, q'\rangle.\mathscr{P}[\![\mathbf{handle}^{\ddagger}\ \mathscr{E}[\mathbf{return}\ x]\ \mathbf{with}\ (q.\ H)(q')]\!]/r] \\
= & \\
 & \mathscr{P}[\![N[V/p, W/q, \lambda\langle x, q'\rangle.\mathbf{handle}^{\ddagger}\ \mathscr{E}[\mathbf{return}\ x]\ \mathbf{with}\ (q.\ H)(q')/r]]\!]
\end{array}
$$

□