# *A general fine-grained reduction theory for effect handlers*

FILIP SIECZKOWSKI
*University of Wrocław, Poland*
*(email:* efes@cs.uni.wroc.pl*)*

MATEUSZ PYZIK
*Wrocław University of Science and Technology, Poland*
*(email:* mateusz.pyzik@pwr.edu.pl*)*

DARIUSZ BIERNACKI
*University of Wrocław, Poland*
*(email:* dabi@cs.uni.wroc.pl*)*

**Abstract**

Effect handlers are a modern and increasingly popular approach to structuring computational effects in functional programming languages. However, while their traditional context-capturing operational semantics is well-suited to implementation tasks, it is less ideal as a reduction theory. We therefore introduce a fine-grained reduction theory for deep effect handlers, inspired by an existing reduction theory for shift0, along with a standard reduction strategy. We relate this strategy to the context-capturing operational semantics via a simulation argument, and show that the reduction theory preserves observational equivalence with respect to the classical semantics of handlers, thus allowing its use as a rewriting theory for handler-equipped programming languages—this rewriting system mostly coincides with previously studied type-based optimisations. In the process, we establish theoretical properties of our reduction theory, including confluence and standardisation theorems, adapting and extending existing techniques. Finally, we demonstrate the utility of our semantics by providing the first normalization-by-evaluation algorithm for effect handlers, expressed both as an abstract machine and as a higher-order normalizer, and we prove its soundness and completeness. Additionally, we establish non-expressibility of the lift operator, found in some effect-handler calculi, by the other constructs.

## 1 Introduction

Effect handlers provide a modern, modular approach to integrating effectful computation in functional programming languages. Based on the pioneering work of (Plotkin and Power 2004) on algebraic effects, and Plotkin and Pretnar's idea of reifying algebras as *handlers* (Plotkin and Pretnar 2013), the technique has seen considerable rise in popularity, with several experimental languages developed (Bauer and Pretnar 2015; Biernacki et al. 2019; Brachthäuser et al. 2020; Convent et al. 2020; Leijen 2017a; Zhang and Myers 2019) and a successful integration within industrial functional languages (Sivaramakrishnan et al. 2021). However, despite its highly semantic and theoretical background, certain aspects of the approach, particularly as they relate to operational semantics, remain somewhat understudied. In this paper, we attempt to reduce this gap by developing a reduction theory for effect handlers, together with some applications.

© 2026 Copyright held by the owner/author(s).
This work is licensed under a Creative Commons Attribution 4.0 International License.

**Journal of Functional Programming**, Volume 36, Article 15 DOI: 10.1017/xxxxx
Submitted July 2025; accepted August 2026

**Motivating example.** Consider the following program fragment, written in a pseudocode that could be interpreted in most programming languages that provide effect handlers:

```
handle
  iter (fn x -> let y = ask () in tell (x + y)) [1, 2, 3]
with
| ask () -> resume 5
| return () -> ()
```

In the above, we assume a standard `iter` function that applies its first argument to the elements of the list in turn (for their side effects), and two *operations*: `ask` and `tell`. These have no interpretation on their own—their behaviour is determined by *handlers* that can be found, dynamically, in their evaluation context. The handler for the `ask` operation is provided: its intended semantics is to resume computation in the place where the operation was encountered with a constant value of `5`. The other operation, on the other hand, is left uninterpreted by this fragment of the code—it may well be unknown at this point if, for instance, the snippet is the body of a function.

To a knowledgeable observer, the intent of this toy example is clear: since the provided handler for `ask` is benign and the environment can never insert a different handler between the operations and the provided one, the program fragment should be equivalent to the following, simpler one:

```
tell 6; tell 7; tell 8
```

Indeed, in an appropriate system (with a static type-and-effect system) we could use known reasoning techniques, such as the logical relation of Biernacki et al. (2018), to establish that the two are observationally equivalent.[1] However, this would be an (admittedly trivial) instance of a sophisticated reasoning technique that is not necessarily easy to automate. Maybe a compiler could transform the first snippet into the second one as an optimisation based simply on specialisation of program fragments?

To consider this question, we must first recall the semantics of effect handlers. There are two known approaches to reducing operations with matching handlers: the first, developed by Kammar et al. (2013), attaches a continuation to an operation when it is found during evaluation, and extends it with pieces of the evaluation context until the matching handler is reached, with the gathered continuation taking the role of the *resumption*, which is passed to the body of the enclosing handler. While this approach reflects the denotational presentation of algebraic effects, it is not optimal as a foundation for a reduction theory. We discuss its relationship with our approach in the following; however, we now focus on the other known approach to computing with handlers, which is an adaptation of context-sensitive reduction semantics for delimited control, due to Leijen (2017a).

In this approach, the gist of the crucial reduction rule is that when we encounter an operation, the rule looks through the evaluation context until a matching handler is found, and only then reifies the intervening part as a resumption. This resumption, together with the argument of the operation are then passed to the body of the handler, as in the previous approach. In the case of our program fragment, the evaluation would look as follows: first, the standard definition of `iter` is unfolded, giving rise to the following snippet:

[1] In fact, it would be easy to prove a more general result that would hold for any input list.

```
handle
  let y = ask () in tell (1 + y);
  iter (fn x -> let y = ask () in tell (x + y)) [2, 3]
with
| ask () -> resume 5
| return () -> ()
```

Now, the operation is in the evaluation position, and its matching handler can be found. Since the argument of the operation is trivial, the only interesting aspect is the resumption that gets captured and passed to the handler, giving us the following result:

```
(fn z ->
  handle
    let y = z in tell (1 + y);
    iter (fn x -> let y = ask () in tell (x + y)) [2, 3]
  with
  | ask () -> resume 5
  | return () -> ()) 5
```

Note that the `handle` expression is retained within the resumption: this is a characteristic feature of *deep* effect handlers, with which we will concern ourselves in this development.

We can now reduce the application of the resumption to its argument, and proceed with evaluation to get the following snippet:

```
handle
  tell 6; iter (fn x -> let y = ask () in tell (x + y)) [2, 3]
with
| ask () -> resume 5
| return () -> ()
```

At this point, however, the operation `tell` is in the evaluation position, and we have no matching handler. No other reduction (save for unfolding `iter` on the tail of the list) can apply, and thus we cannot reach the simpler result.

This example, while clearly artificial, exposes inherent limitations that the standard operational semantics of deep effect handlers imposes on program optimisation through specialisation. The presence of an uninterpreted operation (or, indeed, an uninterpreted variable in the function position of an open program fragment) "freezes" any handlers in its evaluation context, and prevents us from reducing them, thus limiting the possibilities of compiling out known effects into more efficient, pure code. Can we avoid these problems and obtain the simpler program, without any trace of the `ask` operation or its associated handler?

**Fine-grained reduction.** It turns out that if we choose an appropriate reduction theory, such optimisation becomes possible. In fact, one set of rewrite rules that can be used to optimise programs has been proposed by Saleh (2019) and taken up by Karachalias et al. (2021). While their focus is on optimising the general form of calls to recursive, effectful functions wrapped in an effect handler—in the context of our motivating example, calls of the shape `handle iter ... xs with ...`, where `xs` may be a general expression—this task

requires them to consider more local rules for program simplification as well.[2] Independently, Biernacki et al. (2021) proposed a similar set of rules as a *bona fide* reduction theory in the context of delimited-control operators. The core idea behind both sets of rules is to define the reduction of the delimiter (respectively, $ or `handle` construct for the two types of delimited control) through its interactions with other computation formers, notably the `let` expressions. In this approach, rather than form the resumption only when an operation is encountered, a handler would "distribute" through the `let` expression, appropriately adjusting its `return` clause (which determines the result if the computation within the handler terminates with a value). In our example, the reduction (after unfolding the `iter` function) could proceed as follows:

```
handle
  let y = ask () in tell (1 + y)
with
| ask () -> resume 5
| return () ->
  handle
    iter (fn x -> let y = ask () in tell (x + y)) [2, 3]
  with
  | ask () -> resume 5
  | return () -> ()
```

Note how the handler gets duplicated as the iteration over the tail of the list gets incorporated into the `return` clause. This may seem worrisome from the point of view of evaluation (and we address these concerns in Section 2); however, it also unlocks the possibility of further reduction, which was prohibited by the traditional approach. Following on, we get:

```
handle ask ()
with
| ask () -> resume 5
| return y ->
  handle tell (1 + y)
  with
  | ask () -> resume 5
  | return () ->
    handle iter (fn x -> let y = ask () in tell (x + y)) [2, 3]
    with
    | ask () -> resume 5
    | return () -> ()
```

Now, since the handler matches the operation, we can continue—but there is no need to construct the resumption, since there can never be any intervening context. Thus, instead of a "resumption" we simply pass the argument and `return` clause to the body of the handler, obtaining:

[2]The general case they tackle is a proper compiler optimisation that cannot be naturally viewed as part of the reduction theory, and we do not aim to address it in the present paper.

```
handle tell (1 + 5)
with
| ask () -> resume 5
| return () ->
  handle iter (fn x -> let y = ask () in tell (x + y)) [2, 3]
  with
  | ask () -> resume 5
  | return () -> ()
```

This time, the operation within the handler does *not* match the handled operation. Again, there is no need to capture the handler, even though the evaluation of `tell 6` cannot proceed. Instead, we simply sequence the computation of the `return` clause after the control-stuck operation, obtaining:

```
tell 6;
handle iter (fn x -> let y = ask () in tell (x + y)) [2, 3]
with
| ask () -> resume 5
| return () -> ()
```

Now we can continue the reduction process for the tail of the list, which leads to removing all instances of `ask` (and its associated handler) altogether. In this paper, we introduce and study a calculus that makes optimisations like the one described above possible.

**Structure of the paper and contributions.** The remainder of the paper is structured as follows. In Section 2 we develop a core calculus of deep effect handlers, in the style of Piróg et al. (2019). We equip it with two sets of semantics: a context-capturing one, based on capturing parts of the evaluation context as resumptions as outlined above, and a fine-grained one, which draws on the optimisation rules of Saleh (2019) and the reduction theory of Biernacki et al. (2021). We also establish a non-expressibility result for the *lift* construct, which was a feature of some of the systems to date, but whose precise expressive power was not formally studied before. In Section 3 we establish bisimilarity between the traditional semantics and the fine-grained rules, treated as evaluation theories. Thus, we provide a connection between the two sets of semantics of the deep handlers. In Sections 4 and 5 we treat the fine-grained contraction rules as a basis of a *reduction* theory by allowing the reduction rules to be performed in general contexts, and establish key properties of the calculus that ensure it is a sensible reduction theory: the confluence and standardisation results. In this, we follow classic approaches from lambda calculus, and extend them to our setup to account for new challenges posed by the effect handlers. These results allow us to establish, in Section 6, fine-grained equational reasoning as a sound tool to reason about observational equivalence with respect to the traditional semantics, formally justifying Saleh's basic optimisations without recourse to types. Finally, in Section 7 we develop the first normalization-by-evaluation procedure for effect handlers, based on the fine-grained reduction, that takes the form of an abstract machine, and we prove its soundness and completeness with respect to the fine-grained reduction theory. In Section 8 we discuss the most relevant related work, and we conclude in Section 9. Additionally, the article contains Appendix A in which we show an OCaml implementation of the first-order normalizer of Section 7 as well as a higher-order normalizer that has been derived from the first-order

$$e, a \ ::= \ \mathsf{return}\ v \mid \mathsf{let}\ x = a\ \mathsf{in}\ e \mid u\ v \mid \mathsf{do}\ v \mid \mathsf{handle}\ e\ \mathsf{with}\ h, r \mid \mathsf{lift}\ e \quad \text{(expressions)}$$

$$u, v, w, h, r \ ::= \ x \mid \lambda x.e \quad \text{(values)}$$

Fig. 1. The syntax of the handler calculi, shared between the context-capturing and the fine-grained semantics.

one through the functional correspondence between higher-order evaluators and abstract machines (Ager et al. 2003). The implementations take advantage of OCaml's GADTs and of a well-scoped representation of the syntax of the calculus.

**Formalization and implementation.** With the exception of the syntactic abstract machine in Section 2.3 and the expressibility of lift in Section 2.4, all of the theoretical results presented in this paper are formalized in the Rocq interactive theorem prover. Both the Rocq formalization and the OCaml implementation of the normalizers are available at https://doi.org/10.5281/zenodo.19254282.

**Relation to the conference version of this article .** This is a revised and extended version of the work published at ICFP 2023 (Sieczkowski et al. 2023). The proofs in Section 3 were reworked to fit better with the methodology used, which allowed us to significantly streamline and simplify presentation. We have modified the formulation of the syntactic abstract machine (in Section 2.3) and the normalizer (in Section 7) to better highlight the connections between the two and the reduction semantics and enhance the clarity of the presentation. We have also expanded the discussion of methodology and related work in all the technical sections. Finally, in Appendix A we have included a presentation of OCaml implementations of two normalizers based on the theory developed in this article.

## 2 Syntax and semantics

In this section, we introduce two calculi of algebraic effects that share the same syntax but are equipped with different operational semantics. The syntax, presented in Figure 1, is a fine-grain call-by-value $\lambda$-calculus (Levy et al. 2003), extended with algebraic-effect specific constructs—a common setup, used for instance by Hillerström et al. (2020) or Piróg et al. (2019).

More precisely, the syntax is stratified into two categories: values and expressions. Values comprise variables and $\lambda$-abstractions. Note that we do not need to handle recursive functions explicitly, as the calculi are untyped, and thus fixed-point combinators are easily expressible. Expressions represent possibly effectful computations. An expression may return a value, chain computations sequentially with a let-expression, apply a function to an argument (both need to be values), invoke an effect, handle an effectful computation (with $h$ and $r$ representing an effect handler and a return clause, respectively), or *lift* a computation. In contrast to the syntax used in the example in Section 1, the algebraic operations or effects are unnamed; we use the lift operator to allow a single expression to use multiple distinct effects. This is achieved through the lifted computation "skipping" the nearest enclosing handler, which allows us to encode examples such as the one presented in the introduction.

$$
\begin{array}{lll}
K & ::= [\,] \mid K[\mathsf{let}\, x = [\,] \,\mathsf{in}\, e] \mid K[\mathsf{lift}\ [\,]] \mid K[\mathsf{handle}\ [\,] \,\mathsf{with}\, h, r] & \text{(evaluation contexts)} \\
J_n & ::= {}_{(n=0)}\ [\,] \mid \mathsf{let}\, x = J_n \,\mathsf{in}\, e \mid {}_{(n=m+1)}\ \mathsf{lift}\ J_m \mid \mathsf{handle}\, J_{n+1} \,\mathsf{with}\, h, r & (n\text{-free contexts}) \\
R & ::= \mathsf{handle}\, J_0 \,\mathsf{with}\, h, r & \text{(resumptions)}
\end{array}
$$

$$
\begin{array}{llll}
(\beta.\mathsf{v}) & (\lambda x.e)\, v & \mapsto_{\mathsf{cc}} & e[v/x] \\
(\mathsf{let.v}) & \mathsf{let}\, x = \mathsf{return}\, v \,\mathsf{in}\, e & \mapsto_{\mathsf{cc}} & e[v/x] \\
(\mathsf{lift.v}) & \mathsf{lift}\ \mathsf{return}\, v & \mapsto_{\mathsf{cc}} & \mathsf{return}\, v \\
(\mathsf{handle.v}) & \mathsf{handle}\, \mathsf{return}\, v \,\mathsf{with}\, h, r & \mapsto_{\mathsf{cc}} & r\, v \\
(\mathsf{handle.c}) & \mathsf{handle}\, J_0[\mathsf{do}\, v] \,\mathsf{with}\, h, r & \mapsto_{\mathsf{cc}} & \mathsf{let}\, f = h\, v \\
 & & & \mathsf{in}\, f\, (\lambda x.\mathsf{handle}\, J_0[\mathsf{return}\, x] \,\mathsf{with}\, h, r)
\end{array}
$$

$$
\frac{e_1 \mapsto_{\mathsf{cc}} e_2}{K[e_1] \to_{\mathsf{cc}} K[e_2]}
$$

Fig. 2. The traditional, context-capturing operational semantics of $\lambda_{\mathsf{cc}}$.

### 2.1 *Context-capturing calculus*

The first calculus we consider is called $\lambda_{\mathsf{cc}}$: it is an untyped version of a calculus of deep handlers considered by Piróg et al. (2019). The operational semantics of $\lambda_{\mathsf{cc}}$ is presented in Figure 2—it is given as a standard reduction semantics, where a redex can be contracted in any evaluation context. The contraction comprises the standard rules ($\beta$.v) and (let.v), as well as rules that define the interaction of the handler and lift constructs with values and operations. This is simple in the case of values: handling a value (handle.v) consists in passing it to the return value (as no effects need to be handled), whereas lifting a value (lift.v) simply returns it (as no effects need to be lifted).

The most instrumental contraction rule of the calculus is (handle.c), which expresses the handling of an operation by a matching handler. In order to formalize the notion of a matching context, we introduce "free" evaluation contexts, $J$, which are indexed by their freeness level, i.e., the number of handlers surrounding the context hole that the context would need to neutralize any lift operators that surround the context hole to skip handlers. The definition of this indexed family is presented in the style of an attribute grammar: the nonterminal is indexed with a parameter, $n$, which in this case describes how "free" the context is. This parameter can change in the productions (e.g., in the production for handle), but it can also be *constrained* by an equation, which enforces that a given production can only produce contexts at some index (e.g., the [ ] production builds a 0-free context). In essence, this is a compact notation for indexed families of types of the kind one might define in Rocq or Agda; we will use this notation throughout the paper. Whenever the index of an element of such an indexed family is constrained in any way or useful as a clarification, we include it; however, by convention, we drop it when it may be arbitrary.

For our contexts, we can see that in the grammar of $J_n$, lift increases the index (proceeding inside-out), whereas handle decreases it. Thus, balanced contexts are exactly 0-free contexts $J_0$, and the handler surrounding such a context is the one that matches an operation performed within the context. The rule (handle.c) then contracts such a redex by passing the value given to the operation, $v$, to the handler $h$, and passing the captured and reified *resumption*

$$
\begin{array}{rcll}
E_k & ::= & {}_{(k=\mathsf{T})}\ [\,] \mid {}_{(k=\mathsf{T})}\ E_{\mathsf{T}}[\mathsf{let}\, x = [\,] \,\mathsf{in}\, e] \mid {}_{(k=\mathsf{T})}\ E_{\mathsf{T}}[\mathsf{lift}\ [\,]] \mid & \\
 & & {}_{(k=\mathsf{H})}\ E_{k'}[\mathsf{handle}\ [\,] \,\mathsf{with}\, h, r] & \text{(evaluation contexts)} \\
\mathrm{IVal} \ni i & ::= & \mathsf{return}\, v \mid \mathsf{let}\, x = e_1 \,\mathsf{in}\, e_2 \mid \mathsf{do}\, v \mid \mathsf{lift}\ e & \text{(intermediate values)} \\
T_n & ::= & {}_{(n=0)}\ [\,] \mid \mathsf{let}\, x = T_n \,\mathsf{in}\, e \mid {}_{(n=m+1)}\ \mathsf{lift}\ T_m & \text{(resumption contexts)}
\end{array}
$$

$$
\begin{array}{lrcl}
(\beta.\mathsf{v}) & (\lambda x.e)\ v & \mapsto^{k}_{\mathsf{fg}} & e[v/x] \\
(\mathsf{let.v}) & \mathsf{let}\, x = \mathsf{return}\, v \,\mathsf{in}\, e & \mapsto^{\mathsf{T}}_{\mathsf{fg}} & e[v/x] \\
(\mathsf{lift.v}) & \mathsf{lift}\ \mathsf{return}\, v & \mapsto^{\mathsf{T}}_{\mathsf{fg}} & \mathsf{return}\, v \\
(\mathsf{handle.v}) & \mathsf{handle}\, \mathsf{return}\, v \,\mathsf{with}\, h, r & \mapsto^{k}_{\mathsf{fg}} & r\ v \\
(\mathsf{handle.do}) & \mathsf{handle}\, \mathsf{do}\, v \,\mathsf{with}\, h, r & \mapsto^{k}_{\mathsf{fg}} & \mathsf{let}\, f = h\, v \,\mathsf{in}\, f\, r \\
(\mathsf{handle.lift}) & \mathsf{handle}\, \mathsf{lift}\ e \,\mathsf{with}\, h, r & \mapsto^{k}_{\mathsf{fg}} & \mathsf{let}\, x = e \,\mathsf{in}\, r\, x \\
(\mathsf{handle.let}) & \mathsf{handle}\, \mathsf{let}\, x = a \,\mathsf{in}\, e \,\mathsf{with}\, h, r & \mapsto^{k}_{\mathsf{fg}} & \mathsf{handle}\, a \\
 & & & \mathsf{with}\, h, (\lambda\, x\, .\, \mathsf{handle}\, e \,\mathsf{with}\, h, r)
\end{array}
$$

$$
\frac{e_1 \mapsto^{k}_{\mathsf{fg}} e_2}{E_k[e_1] \to_{\mathsf{fg}} E_k[e_2]} \qquad \frac{e_1 \mapsto_{\mathsf{fg}} e_2}{C[e_1] \longrightarrow_{\lambda_{\mathsf{fg}}} C[e_2]}
$$

Fig. 3. The fine-grained operational semantics and reduction theory of $\lambda_{\mathsf{fg}}$. Note that the kind of evaluation context $E_k$ has to match the contraction, while the general reduction may use any contraction in any (possibly binding) context $C$.

(delimited continuation) to the result of the application—this needs to be expanded using a let-binder due to the fine-grained syntax. The resumption comprises the 0-free context together with the surrounding handle expression (we introduce a separate syntactic category $R$ for resumptions that we are going to take advantage of in Section 3); this context is reified as a value by plugging it with a variable bound by a lambda-abstraction on the outside. Capturing the context together with the enclosing handler is characteristic of deep handlers (as opposed to shallow handlers that capture a handler-free resumption (Hillerström et al. 2020)).

The evaluation relation ($\to_{\mathsf{cc}}$) is then defined as a contextual closure of contraction with respect to the entire set of evaluation contexts $K$ that subsume the $n$-free contexts $J_n$ (as usual, $K[e]$ stands for $K$ plugged with $e$).

### 2.2 *Fine-grained calculus*

The second calculus, $\lambda_{\mathsf{fg}}$, builds on the twin lineage of Saleh's optimization rules and Biernacki et al.'s reduction theory for delimited control to avoid explicit capture of contexts and facilitate optimization; it is presented in Figure 3. Similar to the context-capturing semantics, it is formulated as a reduction semantics. However, it incorporates a *hybrid* evaluation strategy (Biernacka et al. 2017; García-Pérez and Nogueira 2014), meaning that the available redexes and contraction rules are dictated by the *kind* of the evaluation context in which they occur. We distinguish two of these kinds, T and H, with the former standing for "top-level" contexts that do not contain handlers, while the latter, for "handling" contexts that

contain at least one handler. The definition of the evaluation context matches this intuition: we can only extend a top-level evaluation context with a let-binding or a lift (and the resulting context is of kind $\mathsf{T}$), while any context can be extended with a handler (giving the result kind $\mathsf{H}$).

Like the evaluation contexts themselves, contractions are also annotated with a kind: these determine in what evaluation context a given contraction may take place. The contraction of applications ($\beta.\mathsf{v}$), let-expressions ($\mathsf{let.v}$) and the lifting of values ($\mathsf{lift.v}$) remain as in the context-capturing semantics, although, in order to guarantee a deterministic operational semantics based on the contraction rules, the latter two may only be used in a top-level context, as evidenced by the $\mathsf{T}$ ascription on the rules. On the other hand, both ($\beta.\mathsf{v}$) and the rules that describe the action of handlers may be used in any context.

The remaining four contractions describe the action of handlers on four kinds of *intermediate values*. These are expressions that are neither redexes, nor can be decomposed further within a handler ($\mathsf{H}$-kinded) context—and thus, from the perspective of reduction of an $\mathsf{H}$-kinded context *behave* like values. They comprise proper values, operations, and let- and lift-expressions. The action of handlers on values ($\mathsf{handle.v}$) remains as before: we simply pass the value to the return clause. The interaction with operations ($\mathsf{handle.do}$) can be much simpler than in the context-capturing case, since we only need to consider an operation *directly* within a handler, with no evaluation context separating the two. This allows us to rewrite $\mathsf{handle\ do}\ v\ \mathsf{with}\ h, r$ as $h\ v\ r$—which, due to the highly stylized presentation of our calculus, we take as syntactic sugar for a let-expression $\mathsf{let}\ f = h\ v\ \mathsf{in}\ f\ r$. Note that $r$ is passed to $h$ as the second argument, which means that at this stage it represents the resumption: in particular, there is no need to duplicate the handler at this point or reify a context by creating a fresh lambda-abstraction. The action of handlers on the lift-expressions ($\mathsf{handle.lift}$) is also straightforward due to the lack of intervening context: the expression within the lift has no access to the handler (as any operations within ought to skip over that handler). Thus, the handler can be dropped, contracting $\mathsf{handle\ lift}\ e\ \mathsf{with}\ h, r$ to $r\ e$, which we write $\mathsf{let}\ x = e\ \mathsf{in}\ r\ x$ in the fine-grained calculus. Finally, we consider the interaction of handlers with let-expressions ($\mathsf{handle.let}$), which is the most complex contraction rule of the fine-grained semantics, and the one that enforces its *deep* treatment of handlers. It states that a handler can be *distributed* to the subcomputations of a let-expression $\mathsf{let}\ x = a\ \mathsf{in}\ e$, by appropriately updating the return clause $r$ of the handler with $e$ enclosed with a copy of the handler the reduction started with. This enforces the intuition about deep handlers that both operations in $a$ and in $e$ should be handled in the same way by explicitly duplicating the handler. Effectively, each use of the ($\mathsf{handle.let}$) rule closes the distance between an effect call or a lift operation and its handler—until they face each other directly and can be eliminated with either the ($\mathsf{handle.do}$) or the ($\mathsf{handle.lift}$) rule.

Besides the operational semantics, which can be used to evaluate terms, we introduce a general reduction relation, written $\longrightarrow_{\lambda_{\mathsf{fg}}}$. This relation arises as the closure of both kinds of contractions under arbitrary, possibly binding, contexts $C$. Thus, it allows us to perform all the evaluation contractions, both in evaluation positions and under binders—but it also allows us to use $\mathsf{T}$-kinded contractions (such as ($\mathsf{let.v}$)) under handlers, i.e., in contexts that would otherwise be $\mathsf{H}$-kinded. This reduction relation corresponds to Saleh's optimization rules; we study this relation and its connection to the fine-grained evaluation in Sections 4 and 5.

In addition to evaluation contexts, which are indexed by kinds, as mentioned before, we also define *resumption contexts*, $T_n$, indexed with natural numbers that determine how many times the lift operation occurs within them. These are isomorphic to the top-level evaluation contexts, $E_{\mathsf{T}}$, but they also mirror the $n$-free contexts of the capturing semantics, $J_n$ in some ways. They are, however, much more restricted due to the nature of the fine-grained reduction. These will be used in Section 3 to relate the two semantics to each other, but play no direct role in the operational semantics.

**Discussion.** With the semantics defined above, we discuss some of the features of our fine-grained semantics, as well as their relation to prior work and possibilities that this framework may unlock. Firstly, both (handle.do) and (handle.let) correspond directly to rewrite rules given by (Saleh 2019) (respectively, rules With-Handled-Op and With-Do in Figure 3.2); (handle.lift) plays the same role as his With-Pure, but in a purely syntax-directed, untyped setting, which makes it more applicable as a basis for a reduction theory. Thus, one could consider our semantics an elevation of a variant of these rules from the position of admissible equations/optimisations in a language to the position of rules that define the semantics.

One consequence of such a treatment that is worth noting is that our architecture of the reduction system immediately precludes any direct extension to shallow handlers (Hillerström and Lindley 2018), whose semantics assumes that the handler is never copied to the resumption. In our setting, the rule (handle.let) eagerly does exactly that, while a shallow handler can only be used in *one* of the two branches of a let-expression.

The reduction rules we propose resemble other reduction theories for calculi of delimited control that are based on piece-by-piece consumption of evaluation contexts (Felleisen 1988; Pretnar 2015). However, in those works the control operator or the operation aggregates the elementary contexts on the way to its surrounding delimiter or handler. In contrast, our calculus consumes the context in the opposite direction, i.e, from the handler to the operation, and we take advantage of the return clause of the handler to store the elementary contexts guarded by (possibly superfluous) handlers. While such semantics will likely be able to reduce programs such as the example in the introduction, it is not clear that they would make for a clean reduction theory as the operations would tend to accumulate all the enclosing contexts as their continuations, giving the resulting programs something of a continuation-passing look.

As mentioned before, our fine-grained semantics is an example of a hybrid evaluation strategy. These strategies are typically used in the semantic specification of full normalization in $\lambda$-calculi (Biernacka et al. 2017; García-Pérez and Nogueira 2014). In contrast, here we use a hybrid nature of the semantics to distinguish between contractions allowed within handlers and the (larger) set of contractions allowed without.

The fact that the fine-grained semantics duplicates handlers very aggressively may not be the first choice for an implementation of an evaluator for a calculus of algebraic effects. It is, however, an interesting alternative to the context-capturing semantics that is more powerful when it comes to program rewriting and optimization, as illustrated with the motivating example in Section 1, and as a foundation for a fully-normalizing procedure (developed in Section 7). The discrepancy between the two semantics in the case where open expressions are reduced is illustrated in Figure 4: note how both the context-capturing and fine-grained semantics reach an open-stuck term, but the captured resumption is much simpler in the fine-grained case. This simplification may be recovered using the reduction relation ($\longrightarrow_{\lambda_{\mathsf{fg}}}$,

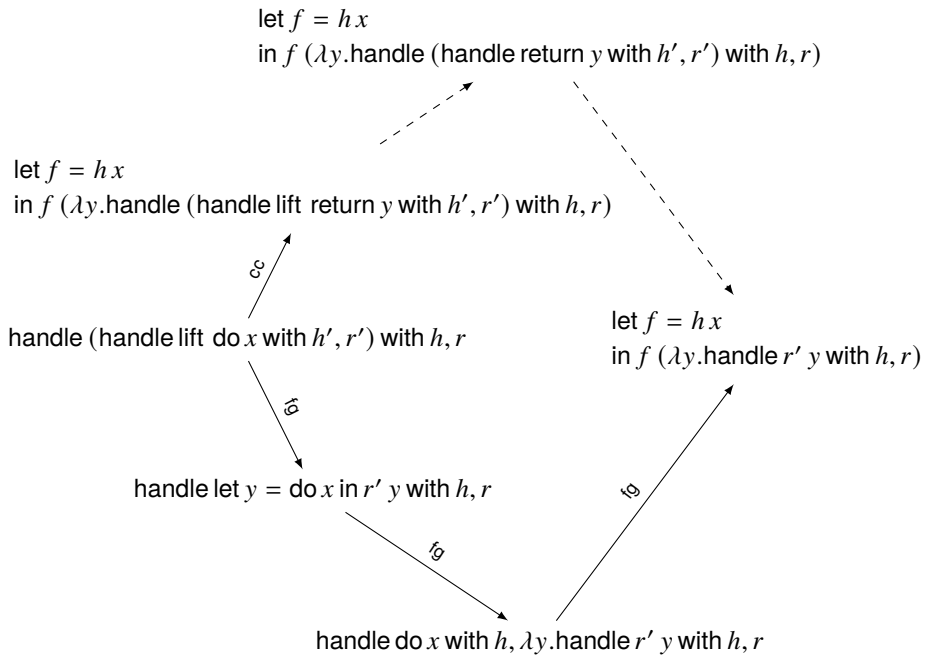


Fig. 4. Comparison of evaluation in the context-capturing and fine-grained semantics. Fine-grained semantics needs three evaluation steps to arrive at a handler call (lower path, annotated with fg), while a single nonlocal evaluation step is sufficient (upper path, annotated with cc). There is a mismatch between the captured resumptions that can be mitigated by two general reduction steps (dashed arrows, fine-grained reductions).

presented as dashed arrows in the figure). The intuition that the resumption captured by the context-capturing semantics may be reduced (in a particular way) to reach the resumption of the fine-grained semantics will be an important insight in the proof that the two semantics converge when evaluating closed terms, developed in Section 3.

Finally, note that the fine-grained semantics allows us to consider whether the rule (lift.v), which is virtually unavoidable in the context-capturing semantics, is necessary or even desirable in fine-grained evaluation. It is never used in "well-handlered" programs, where every lift operation will be tackled by a handler: in particular, this would always be the case in a type-and-effect safe setting. This means that in a non-type-safe setting one could decide that *any* lifted expression in a top-level evaluation context is control-stuck (and thus, that top-level contexts are only formed using let-expressions). We do not pursue the study of this variant semantics further in this paper for a simple reason: we want to establish bisimilarity of our two semantics, which would not hold if we abandoned the rule (lift.v). However, exploring this choice could be an interesting direction for future work.

In the following subsection we present an additional view on the evaluation relation in the fine-grained calculus, offered by a syntactic abstract machine. The abstract machine and the simulation theorems developed in the following section shed light on the inner workings of the new semantics and explain the architecture of the new calculus in greater detail.

$$
\begin{array}{rcl}
\langle \mathsf{handle}\ e\ \mathsf{with}\ h, r \mid E \rangle_{\mathrm{e}} & \triangleright & \langle e \mid E[\mathsf{handle}\ [\,]\ \mathsf{with}\ h, r] \rangle_{\mathrm{e}} \\
\langle v_1\ v_2 \mid E \rangle_{\mathrm{e}} & \triangleright & \langle v_1 \mid v_2 \mid E \rangle_{\mathrm{a}} \\
\langle i \mid E \rangle_{\mathrm{e}} & \triangleright & \langle E \mid i \rangle_{\mathrm{k}} \\
\langle \lambda x.e \mid v \mid E \rangle_{\mathrm{a}} & \triangleright & \langle e[v/x] \mid E \rangle_{\mathrm{e}} \\
\langle E[\mathsf{handle}\ [\,]\ \mathsf{with}\ h, r] \mid i \rangle_{\mathrm{k}} & \triangleright & \langle i \mid h \mid r \mid E \rangle_{\mathrm{h}} \\
\langle E_{\mathsf{T}} \mid i \rangle_{\mathrm{k}} & \triangleright & \langle i \mid E_{\mathsf{T}} \rangle_{\mathrm{t}} \\
\langle \mathsf{return}\ v \mid h \mid r \mid E \rangle_{\mathrm{h}} & \triangleright & \langle r \mid v \mid E \rangle_{\mathrm{a}} \\
\langle \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \mid h \mid r \mid E \rangle_{\mathrm{h}} & \triangleright & \langle e_1 \mid E[\mathsf{handle}\ [\,]\ \mathsf{with}\ h, \lambda x.\mathsf{handle}\ e_2\ \mathsf{with}\ h, r] \rangle_{\mathrm{e}} \\
\langle \mathsf{do}\ v \mid h \mid r \mid E \rangle_{\mathrm{h}} & \triangleright & \langle E \mid \mathsf{let}\ x = h\ v\ \mathsf{in}\ x\ r \rangle_{\mathrm{k}} \\
\langle \mathsf{lift}\ e \mid h \mid r \mid E \rangle_{\mathrm{h}} & \triangleright & \langle E \mid \mathsf{let}\ x = e\ \mathsf{in}\ r\ x \rangle_{\mathrm{k}} \\
\langle \mathsf{return}\ v \mid E_{\mathsf{T}} \rangle_{\mathrm{t}} & \triangleright & \langle E_{\mathsf{T}} \mid v \rangle_{\mathrm{v}} \\
\langle \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \mid E_{\mathsf{T}} \rangle_{\mathrm{t}} & \triangleright & \langle e_1 \mid E_{\mathsf{T}}[\mathsf{let}\ x = [\,]\ \mathsf{in}\ e_2] \rangle_{\mathrm{e}} \\
\langle \mathsf{lift}\ e \mid E_{\mathsf{T}} \rangle_{\mathrm{t}} & \triangleright & \langle e \mid E_{\mathsf{T}}[\mathsf{lift}[\,]] \rangle_{\mathrm{e}} \\
\langle E_{\mathsf{T}}[\mathsf{let}\ x = [\,]\ \mathsf{in}\ e] \mid v \rangle_{\mathrm{v}} & \triangleright & \langle e[v/x] \mid E_{\mathsf{T}} \rangle_{\mathrm{e}} \\
\langle E_{\mathsf{T}}[\mathsf{lift}\ [\,]] \mid v \rangle_{\mathrm{v}} & \triangleright & \langle E_{\mathsf{T}} \mid v \rangle_{\mathrm{v}}
\end{array}
$$

Fig. 5. An abstract machine for fine-grained operational semantics. The machine starts in the configuration $\langle e \mid [\,] \rangle_{\mathrm{e}}$. The final configurations are: $\langle [\,] \mid v \rangle_{\mathrm{v}}$, representing a computed value $v$, $\langle \mathsf{do}\ v \mid E_{\mathsf{T}} \rangle_{\mathrm{t}}$, representing a control-stuck term $E_{\mathsf{T}}[\mathsf{do}\ v]$, and $\langle x \mid v \mid E \rangle_{\mathrm{a}}$, representing an open stuck term $E[x\ v]$.

### *2.3 A syntactic abstract machine*

In this section we present a *syntactic* abstract machine for the fine-grained calculus that is constructed in the spirit of the CK machine (Felleisen et al. 2009). The main goal here is to show how in the new semantics the expressions are decomposed depending on the presence of a handler. A more realistic abstract machine, equipped with environments and other semantic components can be found in Section 7.

The syntactic abstract machine in Figure 5 corresponds through Danvy and Nielsen's refocusing method (Biernacka et al. 2017; Danvy and Nielsen 2004) to a naive implementation of the fine-grained reduction semantics presented above. In such a naive implementation computations are organized into a decompose-contract-recompose loop. However, when a calculus under consideration enjoys the unique-decomposition property, i.e., each non-value expression can be uniquely decomposed into a (potential) redex and an evaluation context, the loop can be optimised to avoid the unnecessary term recomposition after a redex has been contracted, leading to a transition system, typically called an abstract machine, in which term decomposition restarts from the contraction site. In our case, as already mentioned, the reduction semantics enjoys the unique-decomposition property and therefore it lends itself to refocusing, taking into account the fact that it is also hybrid (Biernacka et al. 2017)—the intermediate values $i$, as defined in Figure 3, are interpreted differently depending on whether there is a handler they can interact with or not.

The syntactic abstract machine operates in six different modes:

- in $\langle e \mid E \rangle_{\mathrm{e}}$, based on the structure of $e$, the machine either installs a handler on $E$, or it identifies a potential $(\beta.\mathsf{v})$-redex that will be taken care of through a dedicated mode, or it encounters an intermediate value $i$ and delegates its processing to another mode;
- in $\langle v_1 \mid v_2 \mid E \rangle_{\mathrm{a}}$ the machine contracts a redex according to the rule $(\beta.\mathsf{v})$ or it gets stuck if $v_1$ is a variable;
- in $\langle E \mid i \rangle_{\mathrm{k}}$ the machine inspects the context $E$, finding a redex if it contains a handler (i.e., when it is of kind $\mathsf{H}$) or continuing to process $i$ in a context is of kind $\mathsf{T}$;
- in $\langle i \mid h \mid r \mid E \rangle_{\mathrm{h}}$ the machine contracts the redex according to one of the four contraction rules: (handle.v), (handle.let), (handle.do), (handle.lift);
- in $\langle i \mid E_{\mathsf{T}} \rangle_{\mathrm{t}}$ the machine inspects $i$ and either decomposes it further (let and lift expressions), continues to process $E_{\mathsf{T}}$ if a value is reached, or gets stuck when it encounters an unhandled operation (i.e., a do-expression);
- in $\langle E_{\mathsf{T}} \mid v \rangle_{\mathrm{v}}$ the machine contracts a top-level redex according to one of the two remaining contraction rules (let.v) and (lift.v), or it halts with a computed value $v$, if $E_{\mathsf{T}}$ is empty.

The structure of the machine presented here is slightly different from the one presented in (Sieczkowski et al. 2023), where the evaluation context was divided into two: an inner one representing a possibly empty stack of handlers and an outer one representing the top-level context $E_{\mathsf{T}}$, i.e., the rest of the context beyond the outermost handler. An abstract machine with such a context structure was first defined for a calculus of delimited control (Materzok and Biernacki 2011). This modification is purely cosmetic, but we find the machine presented in this work more concise and more evidently corresponding to the fine-grained reduction semantics. In both cases the machine uses a stack of handlers rather than a stack of expressions or partial continuations coupled with their handlers, which is one of the most notable differences with the more traditional abstract-machine semantics for handlers, e.g., in (Biernacki et al. 2019; Hillerström et al. 2020).

It is worth noting that the hybrid nature of the fine-grained semantics manifests itself in the abstract machine expressly: the intermediate values are interpreted differently in the configurations $\langle i \mid h \mid r \mid E \rangle_{\mathrm{h}}$ and $\langle i \mid E_{\mathsf{T}} \rangle_{\mathrm{t}}$.

Regarding the correctness of the abstract machine, it follows from the correctness of refocusing for hybrid reduction strategies (Biernacka et al. 2017):

Theorem 2.1. *For any expression $e$,*

(1) $e \to_{\mathsf{fg}}^{*} v$ *if and only if* $\langle e \mid [\,] \rangle_{\mathrm{e}} \triangleright^{*} \langle [\,] \mid v \rangle_{\mathrm{v}}$
(2) $e \to_{\mathsf{fg}}^{*} E_{\mathsf{T}}[\mathsf{do}\ v]$ *if and only if* $\langle e \mid [\,] \rangle_{\mathrm{e}} \triangleright^{*} \langle \mathsf{do}\ v \mid E_{\mathsf{T}} \rangle_{\mathrm{t}}$
(3) $e \to_{\mathsf{fg}}^{*} E[x\ v]$ *if and only if* $\langle e \mid [\,] \rangle_{\mathrm{e}} \triangleright^{*} \langle x \mid v \mid E \rangle_{\mathrm{a}}$.

### *2.4 The expressive power of lift*

The lift operator (Leijen 2017a) is a construct present in several calculi of effect handlers in the literature (Biernacki et al. 2018, 2019; Piróg et al. 2019), but its expressive power does not appear to have been formally investigated. In this section we show that lift is *not* eliminable, in the sense of Felleisen (1991), in the untyped calculi of deep handlers considered in this work. We work with $\lambda_{\mathsf{cc}}$, but as we prove in Section 3, the two operational semantics are equivalent and the same reasoning can be carried out in $\lambda_{\mathsf{fg}}$. This result is new and it justifies

the inclusion of lift in the calculus we consider as a construct that increases its expressive power.[3]

More concretely, following Felleisen's work on the expressive power of programming languages (Felleisen 1991), we prove that there does not exist a map $\phi$ from $\lambda_{\mathsf{cc}}$ to $\lambda^{-}_{\mathsf{cc}}$, a restricted version of $\lambda_{\mathsf{cc}}$ where lift has been removed, such that:

- $\phi$ preserves program-ness, i.e., it maps closed expressions to closed expressions
- $\phi$ is a homomorphism on $\lambda^{-}_{\mathsf{cc}}$
- for any program $e$ in $\lambda_{\mathsf{cc}}$, we have that $e$ terminates if and only if $\phi(e)$ terminates.

To this end, we first observe that $\lambda^{-}_{\mathsf{cc}}$ is a conservative restriction of $\lambda_{\mathsf{cc}}$, as required by Felleisen's framework, i.e.:

- the term constructors of $\lambda^{-}_{\mathsf{cc}}$ are a subset of the term constructors of $\lambda_{\mathsf{cc}}$
- the phrases (expressions) of $\lambda^{-}_{\mathsf{cc}}$ are a subset of the phrases of $\lambda_{\mathsf{cc}}$
- the programs (closed expressions) of $\lambda^{-}_{\mathsf{cc}}$ are a subset of the programs of $\lambda_{\mathsf{cc}}$
- the semantics of $\lambda^{-}_{\mathsf{cc}}$ is a restriction of the semantics of $\lambda_{\mathsf{cc}}$, i.e., the semantics of a program in $\lambda^{-}_{\mathsf{cc}}$ is given by the rules ($\beta$.v), (handle.v), (let.v), and (handle.c), where, in the absence of the lift operator, the evaluation contexts $J_0$ are exactly the non-capturing contexts, i.e., contexts where the hole is not guarded by a handler.

Next, let us consider the following program $e$ in $\lambda_{\mathsf{cc}}$

$$\mathsf{handle}\,(\mathsf{handle}\,(\mathsf{lift}\ (\mathsf{do}\,v))\,\mathsf{with}\,\lambda x\,.\,\lambda k\,.\,\Omega, \lambda x\,.\,\Omega)\,\mathsf{with}\,\lambda x\,.\,\lambda k\,.\,x, \lambda x\,.\,x$$

where $v$ is any closed value, and $\Omega$ is a closed non-terminating computation. Applying the reduction rules of $\lambda_{\mathsf{cc}}$, we can see that $e$ evaluates to $v$. Indeed, the inner, diverging, handler is skipped by the lift operator, and the outer handler handles the operation by discarding the resumption and returning the value passed to the operation.

Then, if $\phi$ is a map from $\lambda_{\mathsf{cc}}$ to $\lambda^{-}_{\mathsf{cc}}$ that preserves program-ness and is homomorphic on $\lambda^{-}_{\mathsf{cc}}$, the image of $e$ through $\phi$ is the following program $e'$

$$\mathsf{handle}\,(\mathsf{handle}\,(\phi(\mathsf{lift}\ (\mathsf{do}\,v)))\,\mathsf{with}\,\lambda x\,.\,\lambda k\,.\,\Omega, \lambda x\,.\,\Omega)\,\mathsf{with}\,\lambda x\,.\,\lambda k\,.\,x, \lambda x\,.\,x$$

Now, the (closed) expression $\phi(\mathsf{lift}\ (\mathsf{do}\,v))$ which is in the evaluation position in $e'$ can either diverge, or it can evaluate to a value $u$, or it can evaluate to a control-stuck term of the form $J_n[\mathsf{do}\,u]$ for some non-capturing context $J_n$ and a value $u$. In each case, program $e'$ diverges in $\lambda^{-}_{\mathsf{cc}}$. Therefore, there does not exist a translation from $\lambda_{\mathsf{cc}}$ to $\lambda^{-}_{\mathsf{cc}}$ that makes it possible to locally express (let alone macro-express[4]) lift in terms of algebraic operations and deep handlers.

## 3 Simulation of fine-grained and context-capturing evaluation

In this section we study the relation of the two calculi defined above as evaluation theories. We establish a bisimilarity between the two semantics, which guarantees that whenever an expression evaluates to some value under one semantics it evaluates to a (related) value under the other semantics. This means, in particular, that it is impossible for any expression to

[3]We mean the expressive power in a very precise sense, namely as defined by Felleisen. We only claim that lift cannot be *locally* translated away. Other, *global* translations, e.g., passing the level of freeness from operations to handlers are entirely possible.

[4]For macro-expressibility, the translation function $\phi$ is additionally required to be defined compositionally on the expression being eliminated, in this case, the lift expression (Felleisen 1991).

compute to a value under one semantics while diverging or getting stuck under the other. To achieve this result we follow the methodology developed by Biernacki et al. (2025), extending it to account for the nature of the differences between the two semantics.

As is usually the case with simulation proofs, we need to address the fact that the two programs may diverge in the process of evaluation by introducing a relation between the context-capturing and fine-grained computations. To this end we introduce a relation, written $e \sim e'$, which states that the program $e$ (notionally on the context-capturing side) is related to the program $e'$ (notionally on the fine-grained side). This allows us to state the main simulation theorems.

Lemma 3.1 (Simulation of context-capturing evaluation). *If* $e_1 \to_{\mathsf{cc}}^* e_1'$ *and* $e_1 \sim e_2$, *then* $e_1' \sim e_2$.

Lemma 3.2 (Simulation of fine-grained evaluation). *If* $e_2 \to_{\mathsf{fg}}^* e_2'$ *and* $e_1 \sim e_2$, *then there exists* $e_1'$ *such that* $e_1' \sim e_2'$ *and* $e_1 \to_{\mathsf{cc}}^* e_1'$.

Lemma 3.1 is stronger than the standard analogue of Lemma 3.2 because the simulation relation explicitly incorporates reduction steps on the fine-grained side.

Moreover, the simulation relation is reflexive, and is defined in such a way that if one of the arguments is a normal form (i.e., a value, a control-stuck expression or an open-stuck expression), the other argument can reduce to a matching shape. Together, this means that we can consider the two semantics to define the same partial function.

### 3.1 The similarity relation

In the definition of the similarity relation, we follow our work on simulation between call-by-value and call-by-name evaluation strategies in lambda calculus (Biernacki et al. 2025). The approach is to define congruence relations on our syntactic categories, expressions and values, and extend them with rules that account for discrepancies that may arise between the two semantics. The core part of the relation is presented in Figure 6. As advertised before, it turns out that in our case there is only one source of such a discrepancy: the resumptions captured by a handler may differ quite significantly. Thus, the relation on expressions, $\sim_{\mathrm{e}}$, contains only congruence rules, while the relation on values, $\sim_{\mathrm{v}}$, comprises the two congruence rules and a rule that relates resumptions.

On the context-capturing side (left), a resumption is always of the form $\lambda x.R[x]$, which the rule enforces, while on the fine-grained side (right) any value may be a resumption, since the semantics simply uses the return clause of the handler in this role. The rule then uses an auxiliary relation, $\sim_{\mathrm{h}}$, to relate the captured context to the value and some fine-grained handler, which is no longer present in the resumption.

Thus, the relation $\sim_{\mathrm{h}}$ relates a resumption (i.e., a 0-free context $J$ wrapped within a handler $h, r$) to a pair of a handler $h'$ and a value. The sole rule of its definition posits that the fine-grained handler is related to the handler part of the captured resumption (i.e., $h \sim_{\mathrm{v}} h'$), and that there is a 0-indexed fine-grained resumption $T$ that is related to the 0-free context $J$, and a value $r'$ related to the return clause of the handler $r$. Then, the captured resumption is related to the result of *propagating* the handler $h', r'$ across the resumption $T$. This function is also presented in Figure 6, and amounts to exhaustively applying contraction rules (handle.let) and (handle.lift) at the outside of the resumption. This leaves us needing

$$\mathsf{handle}([\,], h, r) = r$$
$$\mathsf{handle}(\mathsf{let}\, x = T \,\mathsf{in}\, e, h, r) = \mathsf{handle}(T, h, \lambda x.\mathsf{handle}\, e \,\mathsf{with}\, h, r)$$
$$\mathsf{handle}(\mathsf{lift}\; T, h, r) = \mathsf{let}\, x = T \,\mathsf{in}\, r\, x$$

$$\frac{v \sim_{\mathrm{v}} v'}{\mathsf{return}\, v \sim_{\mathrm{e}} \mathsf{return}\, v'} \qquad \frac{a \sim_{\mathrm{e}} a' \qquad e \sim_{\mathrm{e}} e'}{\mathsf{let}\, x = a \,\mathsf{in}\, e \sim_{\mathrm{e}} \mathsf{let}\, x = a' \,\mathsf{in}\, e'} \qquad \frac{e \sim_{\mathrm{e}} e'}{\mathsf{lift}\; e \sim_{\mathrm{e}} \mathsf{lift}\; e'}$$

$$\frac{u \sim_{\mathrm{v}} u' \qquad v \sim_{\mathrm{v}} v'}{u\, v \sim_{\mathrm{e}} u'\, v'} \qquad \frac{v \sim_{\mathrm{v}} v'}{\mathsf{do}\; v \sim_{\mathrm{e}} \mathsf{do}\; v'} \qquad \frac{e \sim_{\mathrm{e}} e' \qquad h \sim_{\mathrm{v}} h' \qquad r \sim_{\mathrm{v}} r'}{\mathsf{handle}\, e \,\mathsf{with}\, h, r \sim_{\mathrm{e}} \mathsf{handle}\, e' \,\mathsf{with}\, h', r'}$$

$$\frac{}{x \sim_{\mathrm{v}} x} \qquad \frac{e \sim_{\mathrm{e}} e'}{\lambda x.e \sim_{\mathrm{v}} \lambda x.e'} \qquad \frac{R \sim_{\mathrm{h}} -, r'}{\lambda x.R[\mathsf{return}\, x] \sim_{\mathrm{v}} r'}$$

$$\frac{J_0 \sim_{\mathrm{R}} T_0 \qquad h \sim_{\mathrm{v}} h' \qquad r \sim_{\mathrm{v}} r'}{\mathsf{handle}\, J_0 \,\mathsf{with}\, h, r \sim_{\mathrm{h}} h', \mathsf{handle}(T_0, h', r')} \qquad \frac{}{[\,] \sim_{\mathrm{R}} [\,]} \qquad \frac{J \sim_{\mathrm{R}} T}{\mathsf{lift}\; J \sim_{\mathrm{R}} \mathsf{lift}\; T}$$

$$\frac{J \sim_{\mathrm{R}} T \qquad e \sim_{\mathrm{e}} e'}{\mathsf{let}\, x = J \,\mathsf{in}\, e \sim_{\mathrm{R}} \mathsf{let}\, x = T \,\mathsf{in}\, e'} \qquad \frac{R \sim_{\mathrm{h}} -, r' \qquad J \sim_{\mathrm{R}} T'}{R[\mathsf{lift}\; J] \sim_{\mathrm{R}} \mathsf{let}\, x = T' \,\mathsf{in}\, r'\, x}$$

Fig. 6. The action of a handler on a resumption and the main part of the similarity relation. The four relations ($\sim_{\mathrm{e}}$, $\sim_{\mathrm{v}}$, $\sim_{\mathrm{h}}$ and $\sim_{\mathrm{R}}$) are mutually inductively defined.

to connect $n$-free contexts on the context-capturing side to $n$-indexed resumptions on the fine-grained side.

The definition of that relation, written $\sim_{\mathrm{R}}$, proceeds by following the shared structure of $n$-free contexts and $n$-indexed resumptions: this accounts for empty contexts, let and lift expressions that are not matched to a handler. To account for the handlers, which may occur on the context-capturing side but not on the fine-grained one, we notice that any handler in an $n$-free contexts needs to be matched to a lift expression, with a 0-free context between them. Thus, we can treat that final kind of an $n$-free context as a resumption with a lift and another $n$-free context plugged inside. This allows us to reuse the relation between resumptions and values, $\sim_{\mathrm{h}}$, to account for the outer part of the context, and the relation between the contexts for the inner part, wrapping the matching resumption in a let expression on the fine-grained side to account for the outer, lifted resumption. Conceptually, one can consider this an encoding of a fact that on the fine-grained side there *used to be* a context that matched the context-capturing resumption precisely, but that it was reduced down to a value, which is all that remains of it.

These four mutually inductively defined relations account for all the discrepancies that can be encountered within the values of our two semantics; however, *during* the process of evaluation, some additional differences may occur. This is in line with the observations of Biernacki et al. (2025) when working with small-step semantics. To this end, we introduce additional relations linking redexes, evaluation contexts and decompositions of the two semantics, as well as the top-level similarity definition. These are presented in Figure 7.

$$\frac{J_n \sim_{\mathrm{R}} T_n}{J_n \sim_{\mathrm{E}} T_n} \qquad \frac{R \sim_{\mathrm{h}} h, r \qquad K \sim_{\mathrm{E}} E}{K[R] \sim_{\mathrm{E}} E[\mathsf{handle}\ [\,]\ \mathsf{with}\ h, r]}$$

$$\frac{u \sim_{\mathrm{v}} u' \qquad v \sim_{\mathrm{v}} v'}{u\ v \sim_{\mathrm{r}}^{k} u'\ v'} \qquad \frac{v \sim_{\mathrm{v}} v' \qquad e \sim_{\mathrm{e}} e'}{\mathsf{let}\, x = \mathsf{return}\, v\, \mathsf{in}\, e \sim_{\mathrm{r}}^{\mathsf{T}} \mathsf{let}\, x = \mathsf{return}\, v'\, \mathsf{in}\, e'}$$

$$\frac{v \sim_{\mathrm{v}} v'}{\mathsf{lift}\ \mathsf{return}\, v \sim_{\mathrm{r}}^{\mathsf{T}} \mathsf{lift}\ \mathsf{return}\, v'} \qquad \frac{v \sim_{\mathrm{v}} v' \qquad h \sim_{\mathrm{v}} h' \qquad r \sim_{\mathrm{v}} r'}{\mathsf{handle}\ \mathsf{return}\, v\ \mathsf{with}\, h, r \sim_{\mathrm{r}}^{k} \mathsf{handle}\ \mathsf{return}\, v'\ \mathsf{with}\, h', r'}$$

$$\frac{v \sim_{\mathrm{v}} v' \qquad h \sim_{\mathrm{v}} h' \qquad r \sim_{\mathrm{v}} r' \qquad J_0 \sim_{\mathrm{R}} T_0}{\mathsf{handle}\, J_0[\mathsf{do}\ v]\ \mathsf{with}\, h, r \sim_{\mathrm{r}}^{k} \mathsf{handle}\, \mathsf{do}\ v'\ \mathsf{with}\, h', \mathrm{handle}(T_0, h', r')}$$

$$\frac{v \sim_{\mathrm{v}} v' \qquad R \sim_{\mathrm{h}} h, r}{R[\mathsf{do}\ v] \sim_{\mathrm{r}}^{k} \mathsf{handle}\, \mathsf{do}\ v'\ \mathsf{with}\, h, r}\ (\mathrm{ALT})$$

$$\frac{v \sim_{\mathrm{v}} v'}{\mathsf{return}\, v \sim_{\mathrm{d}} \mathsf{return}\, v'} \qquad \frac{v \sim_{\mathrm{v}} v' \qquad J_n \sim_{\mathrm{R}} T_n}{J_n[\mathsf{do}\ v] \sim_{\mathrm{d}} T_n[\mathsf{do}\ v']} \qquad \frac{e \sim_{\mathrm{r}}^{k} e' \qquad K \sim_{\mathrm{E}} E_k}{K[e] \sim_{\mathrm{d}} E_k[e']}$$

$$\frac{v \sim_{\mathrm{v}} v' \qquad R \sim_{\mathrm{h}} -, u \qquad K \sim_{\mathrm{E}} E}{K[R[\mathsf{return}\, v]] \sim_{\mathrm{d}} E[u\ v']} \qquad \frac{v \sim_{\mathrm{v}} v' \qquad R \sim_{\mathrm{h}} -, u \qquad K \sim_{\mathrm{E}} E_{\mathsf{T}}}{K[R[\mathsf{lift}\ \mathsf{return}\, v]] \sim_{\mathrm{d}} E_{\mathsf{T}}[\mathsf{let}\, x = \mathsf{return}\, v'\, \mathsf{in}\, u\, x]}$$

$$\frac{e_2 \to_{\mathsf{fg}}^{*} e_2' \qquad e_1 \sim_{\mathrm{d}} e_2'}{e_1 \sim e_2}$$

Fig. 7. The remaining part of the similarity relation. We have relations on evaluation contexts ($\sim_{\mathrm{E}}$), redexes ($\sim_{\mathrm{r}}^{k}$, indexed with the kind of fine-grained context where the redex is contractible), decompositions ($\sim_{\mathrm{d}}$) and the top-level relation ($\sim$).

The first of our relations, written $\sim_{\mathrm{E}}$, links context-capturing evaluation contexts to their fine-grained counterparts. This is achieved by considering two cases: a context-capturing evaluation context is either an $n$-free context for some $n$, or it has the innermost resumption (a handler wrapped around a 0-free context) within an evaluation context. In the former case we can use the relation $\sim_{\mathrm{R}}$ linking the $n$-free context to a fine-grained resumption (which can itself be viewed as a $\mathsf{T}$-indexed fine-grained evaluation context). In the latter case, we can relate our context to an evaluation context with a handle construct inside, provided the handler matches our resumption (via the $\sim_{\mathrm{h}}$ relation), with a recursive call for the remaining parts of the evaluation context.

The second relation connects *redexes*, i.e., left-hand-sides of contraction rules in both semantics. Since some of the fine-grained contractions may only occur in $\mathsf{T}$-indexed contexts, we index the relation on redexes as well, writing $\sim_{\mathrm{r}}^{k}$ for a relation on redexes indexed with kind $k$. As most of the context-capturing contraction rules are mirrored in the fine-grained semantics, the bulk of the relation is structural, although do note that we relate any applications of related functions to related arguments—including matching free variables in

the function positions. The only interesting rule concerns the redex of the rule (handle.c), i.e., an operation within a matching resumption. We relate this redex to a (handle.do) redex on the fine-grained side, utilising our relation on resumptions, $\sim_{\mathrm{h}}$, to relate the resumption on the context-capturing side to the handler and return clause on the fine-grained side. Intuitively, this means that on the fine-grained side there *used to be* a context that matched the context-capturing 0-free context that mediates between operation and the handler, but that this context was reduced by the (handle.let) reductions (as used by the handle function) to the current return clause of the fine-grained handler.

The final complex relation of Figure 7 is the relation on *decompositions*, written $\sim_{\mathrm{d}}$. Both semantics have three kinds of possible decompositions: a complete program is either a value, a *control-stuck* program (i.e., a do operation without a matching handler) or a redex within a matching evaluation context (since it causes no difficulties, we treat open-stuck programs together with actual redexes). The first two rules encompass the former two cases: related values are related decompositions, and operations with related arguments in related non-handling contexts are related. The latter three rules all concern redexes in contexts: the first of these is the standard case, where the two redexes and evaluation contexts match directly. The latter two, on the other hand, account for the differences between the two semantics. Firstly, we need to account for the context-capturing side *calling* a reified resumption: this plugs the captured resumption with a value, leading to a number of possible redexes, while on the fine-grained side there is no natural reduction to perform. Thus, we allow any resumption plugged with a value to relate to an application of the matching return clause to a matching value. Conversely, a resumption plugged with a lift $v$ redex and without any further handlers may not match a lift operation on the fine-grained side, but a let expression where the resumption has been reduced by the handle function, with the let expression produced by the (handle.lift) reduction. Finally, the top-level relation $\sim$ is defined by allowing the fine-grained side any reductions to "catch up" with the decomposition of the context-capturing side of the relation.

### *3.2 Proving the simulations*

With the similarity relation defined, we can focus on proving that each semantics can simulate the other one. We begin with establishing some immediate properties of the structural relations, including reflexivity and closure under related substitutions. The former will be necessary to establish the starting point of the simulation, while the latter allows us to account for $\beta$-reductions.

Lemma 3.3 (Reflexivity of similarity). *The similarity relations $\sim_{\mathrm{e}}$ and $\sim_{\mathrm{v}}$ are reflexive.*

Proof. Reflexivity of both relations is immediate by mutual induction on the structure of expressions and values. □

Lemma 3.4. *The similarity relations $\sim_{\mathrm{e}}$, $\sim_{\mathrm{v}}$, $\sim_{\mathrm{h}}$ and $\sim_{\mathrm{R}}$ are closed under $\sim_{\mathrm{v}}$-related substitutions, i.e., if $\rho$ and $\rho'$ are simultaneous substitutions such that $\rho(x) \sim_{\mathrm{v}} \rho'(x)$ for each variable in their shared domain and $e \sim_{\mathrm{e}} e'$ (respectively, $v \sim_{\mathrm{v}} v'$, etc.), then $\rho^*(e) \sim_{\mathrm{e}} \rho'^*(e')$ (respectively, $\rho^*(v) \sim_{\mathrm{v}} \rho'^*(v')$, etc. )*

Proof. The proof follows immediately via mutual induction over the structure of the similarity relations. □

With these preliminaries out of the way, we turn to properties more directly tied to the simulation, beginning with tying the structural similarity relation to the top-level similarity.

Lemma 3.5. *If $e \sim_{\mathrm{e}} e'$ and $K \sim_{\mathrm{E}} E$ then $K[e] \sim E[e']$.*

Proof. The proof proceeds by induction on the structural relatedness of $e$ and $e'$, with inversion on the relatedness of the evaluation contexts when intermediate values are encountered. In cases where the context-capturing side reaches a decomposition it is relatively straightforward to show that so does the fine-grained side. Where both sides can extend an evaluation context (either where a handle expression is encountered, or because the kind of $E'$ is T) the proof follows by the induction hypothesis with appropriately extended contexts. Finally, where $E'$ starts with a handler and a let or lift expression is encountered, the fine-grained side needs to perform some computation (using (handle.let) or (handle.lift) rules), which it is allowed to by the definition of similarity. Thus, we can reach a state with the appropriate sub-expression in an evaluation position and the two evaluation contexts (extended on the context-capturing side and transformed by the reduction on the fine-grained side) remain related, allowing us to finish by induction hypothesis. □

This is the crucial lemma that ensures that any decomposition on the context-capturing side can be caught up with by the fine-grained side of the simulation. It is used heavily in both directions of the proof, and gives us reflexivity of the main similarity relation as a simple corollary (since empty contexts are related).

Corollary 3.6. *The similarity relation $\sim$ is reflexive.*

**From context-capturing to fine-grained semantics.** We now reach the part of the proof that is specific to the context-capturing semantics. We begin by proving that we can reduce the context-capturing side of two related decompositions.

Lemma 3.7. *If $e_1 \sim_{\mathrm{d}} e_2$ and $e_1 \rightarrow_{\mathsf{cc}} e_1'$, then $e_1' \sim e_2$.*

Proof. We proceed by case analysis on the similarity and reduction relations. The interesting case is when the contraction happens within a resumption plugged with a value, which is related to an application within a context. In that case we need to consider how the context captured by the resumption was formed. If it was empty, the reduction on the context-capturing side is (handle.v), and the fine-grained side already matches the reduct (as two matching applications). If, on the other hand, the context was non-empty, we can consider its innermost constructor to learn about the structure of the value in the function position on the fine-grained side. This can then reduce (as allowed by the definition of similarity), and reach a matching state. The remaining cases simply proceed in lock-step. □

With the lemma above proved, it is a simple consequence that similarity is closed under reduction on the left-hand side, i.e., Lemma 3.1.

Proof of Lemma 3.1. We proceed first by induction on the sequence of context-capturing steps and within, by virtue of similarity being closed under backwards evaluation on the fine-grained side. The latter is immediate from its definition. □

Now we only need to account for the normal forms of the context-capturing semantics, i.e., values, control-stuck programs and open-stuck programs. The connection is established by the following lemma.

Lemma 3.8. *Context-capturing normal forms are only related as decompositions to matching normal forms:*

- *if* $\mathsf{return}\ v \sim_{\mathrm{d}} e$*, then there exists* $v'$ *such that* $e = \mathsf{return}\ v'$ *and* $v \sim_{\mathrm{v}} v'$*;*
- *if* $J_n[\mathsf{do}\ v] \sim_{\mathrm{d}} e$*, then there exist* $T_n$ *and* $v'$ *such that* $e = T_n[\mathsf{do}\ v']$*,* $v \sim_{\mathrm{v}} v'$ *and* $J_n \sim_{\mathrm{R}} T_n$*;*
- *if* $K[x\ v] \sim_{\mathrm{d}} e$*, then there exist* $E$ *and* $v'$ *such that* $e = E[x\ v']$*,* $v \sim_{\mathrm{v}} v'$ *and* $K \sim_{\mathrm{E}} E$*.*

Proof. This is an easy corollary of the uniqueness of decomposition. □

Thus, as a corollary, we obtain our simulation theorem.

Theorem 3.9. *If a program* $e$ *evaluates to a normal form* $e'$ *using the context-capturing semantics, i.e.,* $e \to_{\mathsf{cc}}^{*} e' \not\to_{\mathsf{cc}}$*, then* $e$ *evaluates to a normal form of the same kind using the fine-grained semantics, i.e., there exists* $e''$ *such that* $e \to_{\mathsf{fg}}^{*} e'' \not\to_{\mathsf{fg}}$*. In particular, if* $e' = \mathsf{return}\ v'$ *for some value* $v'$*, then* $e'' = \mathsf{return}\ v''$ *for some value* $v''$*.*

**From fine-grained to context-capturing semantics.** We now consider the reverse direction of the simulation. While the broad structure of the proof remains the same, there are some new complications. The most important of these is that due to the $\eta$-like expansion that occurs when the context-capturing semantics reifies a captured resumption, a ($\beta$.v) redex on the fine-grained side may require a large number of steps on the context-capturing side. This occurs due to a chain of nested resumptions, where a return clause of the handler in the first resumption is itself a reified resumption—since this pattern can continue indefinitely, we need an inductive argument for why they may always reduce.

Lemma 3.10. *The following propositions hold:*

- *if* $u \sim_{\mathrm{v}} u'$*,* $v \sim_{\mathrm{v}} v'$*,* $K \sim_{\mathrm{E}} E$ *and* $E[u'\ v'] \to_{\mathsf{fg}} e'$*, then there exists* $e$ *such that* $K[u\ v] \to_{\mathsf{cc}}^{*} e$ *and* $e \sim e'$*;*
- *if* $R \sim_{\mathrm{h}} -, r'$*,* $v \sim_{\mathrm{v}} v'$*,* $K \sim_{\mathrm{E}} E$ *and* $E[r'\ v'] \to_{\mathsf{fg}} e'$*, then there exists* $e$ *such that* $K[R[\mathsf{return}\ v]] \to_{\mathsf{cc}}^{*} e$ *and* $e \sim e'$*.*

Proof. By mutual induction on the structure of the relations $\sim_{\mathrm{v}}$ and $\sim_{\mathrm{h}}$. In the former case, either the two related values are lambda-abstractions with related bodies, in which case the context-capturing semantics can simply reduce and match the fine-grained result, or it is a reified resumption, in which case it can reduce to plug the resumption with its argument and use the induction hypothesis for the second part of the lemma.

In the second part of the lemma we need to consider the context captured by the resumption. If it is non-empty we have a non-trivial redex, but we also learn the shape of the fine-grained value in the function position. Thus, we can reduce on the context-capturing side and match the fine-grained reduct. If the captured context was empty, on the other hand, the context-capturing side proceeds by the rule (handle.v) to reach an application, at which point we can use the inductive hypothesis for the first part of the lemma. This recursive structure highlights the shape of the chain of resumptions whose handlers' return clauses are themselves reified resumptions, as described above. □

With the main technical hurdle dispatched, we continue in a pattern mirroring the other direction of the simulation, albeit more explicit, since the simulation relation is not closed under backwards context-capturing evaluation.

Lemma 3.11. *If $e_1 \sim_{\mathrm{d}} e_2$ and $e_2 \to_{\mathsf{fg}} e_2'$, then there exists $e_1'$ such that $e_1 \to_{\mathsf{cc}}^* e_1'$ and $e_1' \sim e_2'$.*

Proof. As in the other direction of the simulation, the proof proceeds by cases on the similarity as decompositions and evaluation. The only nontrivial case is the reduction of the application, which is encountered twice and handled via the two cases of the previous lemma. □

We can now obtain Lemma 3.2 by a simple induction, using the fact that fine-grained evaluation is deterministic. The latter property follows trivially from uniqueness of decomposition.

As before we now need to consider the normal forms, this time on the fine-grained side of the similarity relation. The cases for values and control-stuck programs are analogous to the other direction of the simulation; the open-stuck programs, however, may need further reduction on the context-capturing side, due to the resumption nesting described above. Thus, we get the following lemma:

Lemma 3.12. *Fine-grained normal forms are only related as decompositions to matching normal forms (up to additional evaluation for open-stuck programs):*

- *if $e \sim_{\mathrm{d}} \mathsf{return}\, v'$, then there exists $v$ such that $e = \mathsf{return}\, v$ and $v \sim_{\mathrm{v}} v'$;*
- *if $e \sim_{\mathrm{d}} R_n[\mathsf{do}\ v']$, then there exist $J_n$ and $v$ such that $e = J_n[\mathsf{do}\ v]$, $v \sim_{\mathrm{v}} v'$ and $J_n \sim_{\mathrm{R}} R_n$;*
- *if $e \sim_{\mathrm{d}} E[x\, v]$, then there exist $E'$ and $v'$ such that $e \to_{\mathsf{cc}}^* E'[x\, v']$, $v \sim_{\mathrm{v}} v'$ and $E \sim_{\mathrm{E}} E'$.*

Proof. The former two cases are an easy corollary of the unique decomposition property. The latter requires inductive reasoning analogous to Lemma 3.10. □

Again we obtain the simulation theorem as a simple corollary.

Theorem 3.13. *If a program $e$ evaluates to a normal form $e'$ using the context-capturing semantics, i.e., $e \to_{\mathsf{fg}}^* e' \not\to_{\mathsf{fg}}$, then $e$ evaluates to a normal form of the same kind using the fine-grained semantics, i.e., there exists $e''$ such that $e \to_{\mathsf{cc}}^* e'' \not\to_{\mathsf{cc}}$. In particular, if $e' = \mathsf{return}\, v'$ for some value $v'$, then $e'' = \mathsf{return}\, v''$ for some value $v''$.*

Thus, we have established that the two formats of semantics are equivalent as evaluation functions and precisely characterised the possible differences in the shape of the values and expressions that the two semantics produce. With our choice of semantics thus justified with respect to the commonly used format, we proceed to study the fine-grained contractions as a reduction theory.

## 4 Confluence of multi-step fine-grained reduction

In this section we present a proof sketch of one of the most fundamental properties of the reduction relation $\longrightarrow_{\lambda_{\mathsf{fg}}}$ one expects, namely the confluence of the reduction relation or the Church-Rosser theorem (Barendregt 1985), expressed as the following diamond property for $\longrightarrow_{\lambda_{\mathsf{fg}}}^*$:

Theorem 4.1 (Confluence). *If $e \longrightarrow_{\lambda_{\mathsf{fg}}}^* e_1$ and $e \longrightarrow_{\lambda_{\mathsf{fg}}}^* e_2$, then there exists a term $e'$, such that $e_1 \longrightarrow_{\lambda_{\mathsf{fg}}}^* e'$ and $e_2 \longrightarrow_{\lambda_{\mathsf{fg}}}^* e'$.*

$$\frac{}{x \Rrightarrow_{\mathrm{v}} x}\ \textsc{s-var} \qquad \frac{e \Rrightarrow_{\mathrm{e}} e'}{\lambda x.e \Rrightarrow_{\mathrm{v}} \lambda x.e'}\ \textsc{s-lam}$$

$$\frac{v \Rrightarrow_{\mathrm{v}} v'}{\mathsf{return}\ v \Rrightarrow_{\mathrm{e}} \mathsf{return}\ v'}\ \textsc{s-ret}$$

$$\frac{}{\mathsf{let}\, x = e_1\, \mathsf{in}\, e_2 \Rightarrow \mathsf{let}\, x = e_1\, \mathsf{in}\, e_2}\ \textsc{nl-let} \qquad \frac{}{\mathsf{let}\, x = \mathsf{return}\, v_1\, \mathsf{in}\, e_2 \Rightarrow e_2[v_1/x]}\ \textsc{nl-red}$$

$$\frac{e_1 \Rrightarrow_{\mathrm{e}} e_1' \quad e_2 \Rrightarrow_{\mathrm{e}} e_2' \quad \mathsf{let}\, x = e_1'\, \mathsf{in}\, e_2' \Rightarrow e}{\mathsf{let}\, x = e_1\, \mathsf{in}\, e_2 \Rrightarrow_{\mathrm{e}} e}\ \textsc{s-let}$$

$$\frac{}{v_1\, v_2 \Rightarrow v_1\, v_2}\ \textsc{na-app} \qquad \frac{}{(\lambda x.e)\, v_2 \Rightarrow e[v_2/x]}\ \textsc{na-red}$$

$$\frac{v_1 \Rrightarrow_{\mathrm{v}} v_1' \quad v_2 \Rrightarrow_{\mathrm{v}} v_2' \quad v_1'\, v_2' \Rightarrow e}{v_1\, v_2 \Rrightarrow_{\mathrm{v}} e}\ \textsc{s-app}$$

Fig. 8. Definition of the super-parallel reduction $\Rrightarrow_{\mathrm{e}}$ and $\Rrightarrow_{\mathrm{v}}$ for the pure fragment of the calculus.

We prove this theorem using a technique of *super-parallel reductions* and *complete super developments* (Aczel 1978; McKinna 2022; McKinna and Pollack 1999; Takahashi 1995; van Raamsdonk 1993) that is a simplification of the classic proof method by *parallel reductions* due to Tait and Martin-Löf (Barendregt 1985). The idea is to define a super-parallel reduction relation $\Rrightarrow_{\mathrm{e}}$ on expressions (along with a super-parallel reduction relation $\Rrightarrow_{\mathrm{v}}$ on values, with analogous properties), such that $\longrightarrow^*_{\lambda_{\mathsf{fg}}}$ and $\Rrightarrow^*_{\mathrm{e}}$ coincide, and therefore, to establish the confluence of $\longrightarrow_{\lambda_{\mathsf{fg}}}$, it suffices to show that $\Rrightarrow_{\mathrm{e}}$ satisfies the diamond property, which we do using Takahashi's triangles (Takahashi 1995).

### *4.1 Super-parallel reductions*

In Figure 8, we present the rules defining $\Rrightarrow_{\mathrm{e}}$ and $\Rrightarrow_{\mathrm{v}}$ for the pure fragment of the calculus. The idea is to contract, in parallel, all, some or none of the redexes present in an expression or a value. To this end, for each construct that can be eliminated by one or more reduction rules, we introduce an auxiliary relation $\Rightarrow$ that non-deterministically applies the reduction rules or leaves the term intact, i.e., $\Rightarrow$ is a reflexive closure of the contraction relation.

The remaining redexes are treated along similar lines. In Figure 9 we present the rules for the effectful fragment of the calculus. Two rules stand out: NH-RET and NH-LIFT. As we explain in the next section, these two rules require additional reductions, as prescribed by the premises of the rules, in order to make it possible to resolve critical pairs present in expressions of the form $\mathsf{handle}\,(\mathsf{let}\, x = \mathsf{return}\, v\, \mathsf{in}\, e)\, \mathsf{with}\, h, r$ and $\mathsf{handle}\,(\mathsf{lift}\ (\mathsf{return}\, v))\, \mathsf{with}\, h, r$, when proving the diamond property of $\Rrightarrow_{\mathrm{e}}$.

The following lemma gathers some of the useful properties of $\Rrightarrow_{\mathrm{e}}$ ($\Rrightarrow_{\mathrm{v}}$ satisfies the analogous properties).

$$\frac{v \Rrightarrow_{\mathrm{v}} v'}{\mathsf{do}\, v \Rrightarrow_{\mathrm{e}} \mathsf{do}\, v'}\ \textsc{s-do}$$

$$\frac{}{\mathsf{lift}\ e \Rightarrow \mathsf{lift}\ e}\ \textsc{nl-lift} \qquad \frac{}{\mathsf{lift}\ (\mathsf{return}\, v) \Rightarrow \mathsf{return}\, v}\ \textsc{nl-red}$$

$$\frac{e \Rrightarrow_{\mathrm{e}} e' \quad \mathsf{lift}\ e' \Rightarrow e''}{\mathsf{lift}\ e \Rrightarrow_{\mathrm{e}} e''}\ \textsc{s-lift}$$

$$\frac{}{\mathsf{handle}\, e\, \mathsf{with}\, h, r \Rightarrow \mathsf{handle}\, e\, \mathsf{with}\, h, r}\ \textsc{nh-handle}$$

$$\frac{r\, v \Rightarrow e}{\mathsf{handle}\ (\mathsf{return}\, v)\ \mathsf{with}\, h, r \Rightarrow e}\ \textsc{nh-ret}$$

$$\frac{x \notin \mathrm{FV}(h, r)}{\mathsf{handle}\ (\mathsf{let}\, x = a\, \mathsf{in}\, e)\ \mathsf{with}\, h, r \Rightarrow \mathsf{handle}\, a\, \mathsf{with}\, h, \lambda x.\mathsf{handle}\, e\, \mathsf{with}\, h, r}\ \textsc{nh-let}$$

$$\frac{r\, x \Rightarrow e_1 \quad \mathsf{let}\, x = e\, \mathsf{in}\, e_1 \Rightarrow e_2 \quad x \notin \mathrm{FV}(r)}{\mathsf{handle}\ (\mathsf{lift}\ e)\ \mathsf{with}\, h, r \Rightarrow e_2}\ \textsc{nh-lift}$$

$$\frac{x \notin \mathrm{FV}(r)}{\mathsf{handle}\ (\mathsf{do}\, v)\ \mathsf{with}\, h, r \Rightarrow \mathsf{let}\, x = h\, v\, \mathsf{in}\, x\, r}\ \textsc{nh-let}$$

$$\frac{e_1 \Rrightarrow_{\mathrm{e}} e_2 \quad h_1 \Rrightarrow_{v} h_2 \quad r_1 \Rrightarrow_{v} r_2 \quad \mathsf{handle}\, e_2\, \mathsf{with}\, h_2, r_2 \Rightarrow e}{\mathsf{handle}\, e_1\, \mathsf{with}\, h_1, r_1 \Rrightarrow_{\mathrm{e}} e}\ \textsc{s-handle}$$

Fig. 9. Definition of the super-parallel reduction $\Rrightarrow_{\mathrm{e}}$ for the effectful fragment of the calculus.

Lemma 4.2. *For any expressions $e$ and $e'$, and values $v$ and $v'$, we have:*

(1) $e \Rrightarrow_{\mathrm{e}} e$.
(2) *If* $e \longrightarrow_{\lambda_{\mathrm{fg}}} e'$, *then* $e \Rrightarrow_{\mathrm{e}} e'$.
(3) *If* $e \Rrightarrow_{\mathrm{e}} e'$, *then* $e \longrightarrow^{*}_{\lambda_{\mathrm{fg}}} e'$.
(4) *If* $e \Rrightarrow_{\mathrm{e}} e'$ *and* $v \Rrightarrow_{\mathrm{v}} v'$, *then* $e[v/x] \Rrightarrow_{\mathrm{e}} e'[v'/x]$ *for any variable* $x$.

This lemma, in each case, is proved by straightforward induction (mutually inductively with the corresponding statements for $\Rrightarrow_{\mathrm{v}}$). It implies that $\longrightarrow^{*}_{\lambda_{\mathrm{fg}}}$ and $\Rrightarrow^{*}_{\mathrm{e}}$ coincide, and it establishes the substitutivity property of $\Rrightarrow_{\mathrm{e}}$ that plays an important role in the proof of the diamond property for $\Rrightarrow_{\mathrm{e}}$.

### *4.2 Complete super developments and the triangle lemma*

As for the proof of the diamond property of $\Rrightarrow_{\mathrm{e}}$, we take advantage of the notion of a complete super development of an expression (value), which is a *deterministic* relation $\Rrightarrow^{\mathrm{c}}_{\mathrm{e}}$ ($\Rrightarrow^{\mathrm{c}}_{\mathrm{v}}$) that simultaneously and eagerly contracts redexes existing in a term as well as those introduced by the rules such as NL-RED. More precisely, $\Rrightarrow^{\mathrm{c}}_{\mathrm{e}}$ is a subrelation of $\Rrightarrow_{\mathrm{e}}$, obtained from $\Rrightarrow_{\mathrm{e}}$

by restricting all of the non-reducing non-deterministic $\Rightarrow$-rules NL-LET, NA-APP, NL-LIFT, and NH-HANDLE, so that these rules are not allowed to introduce a top-level redex—such a redex has to be contracted. For example, in NL-LET, $e_1$ is not allowed to have the form $\mathsf{return}\, v$—such a let expression has to be reduced by NL-RED. This subrelation of $\Rightarrow$ is noted $\Rightarrow^{\mathrm{c}}$.

Lemma 4.3. *For any expressions $e$ and $e'$, we have:*

(1) *If $e \Rightarrow^{c} e'$, then $e \Rightarrow e'$.*
(2) *If $e \Rrightarrow^{\mathrm{c}}_{\mathrm{e}} e'$, then $e \Rrightarrow_{\mathrm{e}} e'$.*
(3) *$\Rightarrow^{\mathrm{c}}$ and $\Rrightarrow^{\mathrm{c}}_{\mathrm{e}}$ are deterministic.*
(4) *$\Rrightarrow^{\mathrm{c}}_{\mathrm{e}}$ is total.*

Since the relation $\Rrightarrow^{\mathrm{c}}_{\mathrm{e}}$ contracts at least as many redexes as $\Rrightarrow_{\mathrm{e}}$, we are in a position to prove the *triangle lemma* of Takahashi (1995). The following lemmas are the central ingredient in the proof of the triangle lemma (Lemma 4.8), and their proofs determine the actual definition of the relation $\Rightarrow$, and, therefore, of $\Rrightarrow_{\mathrm{e}}$. Below we state one such lemma for each of the constructs that can introduce a redex, i.e., for application, let-expression, lift-expression, and handle-expression, and we present the most instructive fragments of the proof for the handle-construct that deal with the only two possible critical pairs in the calculus and justify the shape of the rules NH-RET and NH-LIFT. (Lemmas 4.4 and 4.5 require the substitutivity property of Lemma 4.2.)

Lemma 4.4. *If $u_1\, v_1 \Rightarrow e$, $u_1 \Rrightarrow_{\mathrm{v}} u_2$, $v_1 \Rrightarrow_{\mathrm{v}} v_2$, and $u_2\, v_2 \Rightarrow^{\mathrm{c}} e'$, then $e \Rrightarrow_{\mathrm{e}} e'$.*

Lemma 4.5. *If $\mathsf{let}\, x = a_1\, \mathsf{in}\, e_1 \Rightarrow e$, $a_1 \Rrightarrow_{\mathrm{e}} a_2$, $e_1 \Rrightarrow_{\mathrm{e}} e_2$, and $\mathsf{let}\, x = a_2\, \mathsf{in}\, e_2 \Rightarrow^{\mathrm{c}} e'$, then $e \Rrightarrow_{\mathrm{e}} e'$.*

Lemma 4.6. *If $\mathsf{lift}\ e_1 \Rightarrow e$, $e_1 \Rrightarrow_{\mathrm{e}} e_2$, and $\mathsf{lift}\ e_2 \Rightarrow^{\mathrm{c}} e'$, then $e \Rrightarrow_{\mathrm{e}} e'$.*

Lemma 4.7. *If $\mathsf{handle}\, e_1\, \mathsf{with}\, h_1, r_1 \Rightarrow e$, $e_1 \Rrightarrow_{\mathrm{e}} e_2$, $h_1 \Rrightarrow_{\mathrm{v}} h_2$, $r_1 \Rrightarrow_{\mathrm{v}} r_2$, and $\mathsf{handle}\, e_2\, \mathsf{with}\, h_2, r_2 \Rightarrow^{\mathrm{c}} e'$, then $e \Rrightarrow_{\mathrm{e}} e'$.*

Proof. The proof is done by case analysis on $\mathsf{handle}\, e_1\, \mathsf{with}\, h_1, r_1 \Rightarrow e$.

The case of the the NH-HANDLE-rule is not interesting and straightforward.

Let us consider the NH-LET-rule. In this case we have $e_1 = \mathsf{let}\, x = e_1'\, \mathsf{in}\, e_1''$, and $e = \mathsf{handle}\, e_1'\, \mathsf{with}\, h_1, \lambda x.\mathsf{handle}\, e_1''\, \mathsf{with}\, h_1, r_1$. Let us consider the case where $\mathsf{let}\, x = e_1'\, \mathsf{in}\, e_1'' \Rrightarrow_{\mathrm{e}} e_2''[v/x]$ with $e_1' \Rrightarrow_{\mathrm{e}} \mathsf{return}\, v$ and $e_1'' \Rrightarrow_{\mathrm{e}} e_2''$ (the other case, when $\mathsf{let}\, x = e_1'\, \mathsf{in}\, e_1'' \Rrightarrow_{\mathrm{e}} \mathsf{let}\, x = e_2'\, \mathsf{in}\, e_2''$ is less interesting). First, let us observe that $e' = \mathsf{handle}\, e_2''[v/x]\, \mathsf{with}\, h_2, r_2$. Second, by rules S-LAM and S-HANDLE, we have:

$$\lambda x.\mathsf{handle}\, e_1''\, \mathsf{with}\, h_1, r_1 \Rrightarrow_{\mathrm{v}} \lambda x.\mathsf{handle}\, e_2''\, \mathsf{with}\, h_2, r_2.$$

Since

$$(\lambda x.\mathsf{handle}\, e_1''\, \mathsf{with}\, h_2, r_2)\, v \Rightarrow e'$$

it follows that $e \Rrightarrow_{\mathrm{e}} e'$ by the rule S-HANDLE. This part of the proof resolves a critical pair arising from expressions of the form

$$\mathsf{handle}\, \mathsf{let}\, x = \mathsf{return}\, v\, \mathsf{in}\, e\, \mathsf{with}\, h, r$$

and hinges on the premise of the NH-RET-rule that allows us to reduce the application of the continuation $r$ to the value $v$ in the last step above, which makes it possible for $\Rrightarrow_{\mathrm{e}}$ to catch up with $\Rrightarrow^{\mathrm{c}}_{\mathrm{e}}$.

The other critical pair occurs when $e_1 = \mathsf{lift}\ e_1'$ and the rule NH-LIFT is used. In that case, $r_1\, x \Rightarrow e_r$ and $\mathsf{let}\, x = e_1'\, \mathsf{in}\, e_r \Rightarrow e$. Let us consider the case where $\mathsf{lift}\ e_1' \Rightarrow_{\mathrm{e}} \mathsf{return}\ v$ with $e_1' \Rightarrow_{\mathrm{e}} \mathsf{return}\ v$ (the other case, when $\mathsf{lift}\ e_1' \Rightarrow_{\mathrm{e}} \mathsf{lift}\ e_1''$ with $e_1' \Rightarrow_{\mathrm{e}} e_1''$, is similar and slightly less interesting). By the definition of $\Rightarrow^{\mathrm{c}}$, we get that $r_2\, v \Rightarrow^{\mathrm{c}} e'$. Since $\Rightarrow^{\mathrm{c}}$ is deterministic as well as total on applications and substitutive in the argument position (if $r\, x \Rightarrow^{\mathrm{c}} e$, then $r\, v \Rightarrow^{\mathrm{c}} e[v/x]$, assuming $x \notin \mathrm{FV}(r)$), it follows that $e' = e''[v/x]$, where $r_2\, x \Rightarrow^{\mathrm{c}} e''$. By using Lemma 4.4, we get $e_r \Rightarrow_{\mathrm{e}} e''$, which makes it possible to apply Lemma 4.5 to what we have learnt, namely that $\mathsf{let}\, x = e_1'\, \mathsf{in}\, e_r \Rightarrow e$, $e_1' \Rightarrow_{\mathrm{e}} \mathsf{return}\, v$, $e_r \Rightarrow_{\mathrm{e}} e''$, and $\mathsf{let}\, x = \mathsf{return}\, v\, \mathsf{in}\, e'' \Rightarrow e''[v/x]$, and conclude that $e \Rightarrow_{\mathrm{e}} e'$. This case resolves a critical pair arising from the expressions of the form

$$\mathsf{handle}\ (\mathsf{lift}\ \ (\mathsf{return}\ v))\ \mathsf{with}\ h, r$$

that, by reducing the inner redex first, $\Rightarrow_{\mathrm{e}}^{\mathrm{c}}$ rewrites into $e'$ such that $r'\, v' \Rightarrow^{\mathrm{c}} e'$ (where $v \Rightarrow_{\mathrm{v}}^{\mathrm{c}} v'$ and $r \Rightarrow_{\mathrm{v}}^{\mathrm{c}} r'$). However, $\Rightarrow_{\mathrm{e}}$ can choose the outer redex, which leads to a let-expression

$$\mathsf{let}\, x = \mathsf{return}\, v\, \mathsf{in}\, e''$$

that is immediately reduced by $\Rightarrow$ to $e''[v/x]$, which can only catch up with $e'$, if $r''\, x \Rightarrow e''$ (where $r \Rightarrow_{\mathrm{v}} r''$). This is the reason why the NH-LIFT-rule requires the two premises.

The remaining cases, given by the rules NH-RET and NH-DO, are proved along the same lines, but since they are not concerned with critical pairs, they are less telling. □

The triangle lemma follows from Lemmas 4.4-4.7:

Lemma 4.8. *If $e \Rightarrow_{\mathrm{e}} e'$ and $e \Rightarrow_{\mathrm{e}}^{\mathrm{c}} e^{\bullet}$, then $e' \Rightarrow_{\mathrm{e}} e^{\bullet}$.*

Proof. The proof is done by induction on $e \Rightarrow_{\mathrm{e}} e'$ (induction on $e \Rightarrow_{\mathrm{e}}^{\mathrm{c}} e^{\bullet}$ works as well). We show the proof only for the S-HANDLE-rule, as a representative case.

Let us assume that $e = \mathsf{handle}\, e_0\, \mathsf{with}\, h_0, r_0$ and $e \Rightarrow_{\mathrm{e}} e'$. Then, for some $e_1$, $h_1$, and $r_1$, we have $e_0 \Rightarrow_{\mathrm{e}} e_1$, $h_0 \Rightarrow_{\mathrm{v}} h_1$, $r_0 \Rightarrow_{\mathrm{v}} r_1$, and $\mathsf{handle}\, e_1\, \mathsf{with}\, h_1, r_1 \Rightarrow e'$. By $e \Rightarrow_{\mathrm{e}}^{\mathrm{c}} e^{\bullet}$, we then also have $e_0 \Rightarrow_{\mathrm{e}}^{\mathrm{c}} e_0^{\bullet}$, $h_0 \Rightarrow_{\mathrm{v}}^{\mathrm{c}} h_0^{\bullet}$, $r_0 \Rightarrow_{\mathrm{v}}^{\mathrm{c}} r_0^{\bullet}$, and $\mathsf{handle}\, e_0^{\bullet}\, \mathsf{with}\, h_0^{\bullet}, r_0^{\bullet} \Rightarrow^{\mathrm{c}} e^{\bullet}$, for some $e_0^{\bullet}$, $h_0^{\bullet}$, and $r_0^{\bullet}$. By the induction hypothesis, we learn that $e_1 \Rightarrow_{\mathrm{e}} e_0^{\bullet}$, $h_1 \Rightarrow_{\mathrm{v}} h_0^{\bullet}$, and $r_1 \Rightarrow_{\mathrm{v}} r_0^{\bullet}$. We therefore reduce the proof in this case to the task of showing $e' \Rightarrow_{\mathrm{e}} e^{\bullet}$ from $\mathsf{handle}\, e_1\, \mathsf{with}\, h_1, r_1 \Rightarrow e'$, $e_1 \Rightarrow_{\mathrm{e}} e_0^{\bullet}$, $h_1 \Rightarrow_{\mathrm{v}} h_0^{\bullet}$, $r_1 \Rightarrow_{\mathrm{v}} r_0^{\bullet}$, and $\mathsf{handle}\, e_0^{\bullet}\, \mathsf{with}\, h_0^{\bullet}, r_0^{\bullet} \Rightarrow^{\mathrm{c}} e^{\bullet}$, which is exactly what Lemma 4.7 states.

The remaining cases are treated analogously, using the appropriate lemma for each construct (Lemmas 4.4, 4.5, and 4.6). □

As a corollary, we show that $\Rightarrow_{\mathrm{e}}$ satisfies the diamond property, which implies Theorem 4.1.

Lemma 4.9. *If $e \Rightarrow_{\mathrm{e}} e_1$ and $\Rightarrow_{\mathrm{e}} e_2$, then there exists $e^{\bullet}$ such that $e_1 \Rightarrow_{\mathrm{e}} e^{\bullet}$ and $e_2 \Rightarrow_{\mathrm{e}} e^{\bullet}$.*

Proof. Using the facts that $\Rightarrow_{\mathrm{e}}^{\mathrm{c}}$ is functional, i.e., deterministic and total (Lemma 4.2), if $e \Rightarrow_{\mathrm{e}} e_1$ and $e \Rightarrow_{\mathrm{e}} e_2$, we can construct two triangles given by Lemma 4.8, and compose them into the desired diamond, such that $e_1 \Rightarrow_{\mathrm{e}} e^{\bullet}$ and $e_2 \Rightarrow_{\mathrm{e}} e^{\bullet}$, where $e^{\bullet}$ is the unique term such that $e \Rightarrow_{\mathrm{e}}^{\mathrm{c}} e^{\bullet}$ (where $e^{\bullet}$ does not depend on $e_1$ or $e_2$). □

As a comment on the developments of this section, we would like to argue that the fact that our definition of $\Rightarrow_{\mathrm{e}}$ is built in the spirit of *super*-parallel reductions (McKinna 2022; van Raamsdonk 1993), rather than of the more familiar parallel reductions (Takahashi 1995) is essential for the proof to go through. The latter reduces all the redexes that are already

present in a term under consideration and works well, e.g., when there are no critical pairs in the calculus, as is the case, e.g., in the $\lambda$-calculus with $\beta$-reduction. In contrast, super parallel reductions are allowed to contract more redexes, i.e., also some redexes that are created after the subterms of a given term have been super-parallel reduced. This extra eagerness of the super-parallel reductions seems critical for the triangle lemma to hold in our calculus. In particular, in a parallel reduction relation we would have the following rules:

$$\frac{v_1 \Rrightarrow_{\mathrm{v}} v_1' \quad e_2 \Rrightarrow_{\mathrm{e}} e_2'}{\mathsf{let}\, x = \mathsf{return}\, v_1 \,\mathsf{in}\, e_2 \Rrightarrow_{\mathrm{e}} e_2'[v_1'/x]}$$

and

$$\frac{e_1 \Rrightarrow_{\mathrm{e}} e_1' \quad e_2 \Rrightarrow_{\mathrm{e}} e_2' \quad h \Rrightarrow_{\mathrm{v}} h' \quad r \Rrightarrow_{\mathrm{v}} r'}{\mathsf{handle}\,\mathsf{let}\, x = e_1 \,\mathsf{in}\, e_2 \,\mathsf{with}\, h, r \Rrightarrow_{\mathrm{e}} \mathsf{handle}\, e_1' \,\mathsf{with}\, h', (\lambda x.\mathsf{handle}\, e_2' \,\mathsf{with}\, h', r')}$$

If we now look at an expression $e = \mathsf{handle}\,(\mathsf{let}\, x = \mathsf{return}\, v \,\mathsf{in}\, e_2)\,\mathsf{with}\, h, r$, we observe that $e \Rrightarrow_{\mathrm{e}} e_1' = \mathsf{handle}\, e_2'[v'/x]\,\mathsf{with}\, h', r'$—we are allowed to reduce the inner redex, but also $e \Rrightarrow_{\mathrm{e}}^{\mathrm{c}} e_2' = \mathsf{handle}\,\mathsf{return}\, v''\,\mathsf{with}\, h'', (\lambda x.\mathsf{handle}\, e_2''\,\mathsf{with}\, h'', r'')$—the outer redex has to be reduced by $\Rrightarrow_{\mathrm{e}}^{\mathrm{c}}$, as dictated by the above rule for the handle-construct. In this case, $e_1' \not\Rrightarrow_{\mathrm{e}} e_2'$, in general.

We, therefore, conjecture that our calculus requires some form of super parallel reductions and complete super developments for the confluence proof based on parallel reductions to go through. Moreover, the actual definition of $\Rrightarrow_{\mathrm{e}}$ is not a trivial generalization of the one for the pure $\lambda$-calculus, but it is quite subtle in the choice of which redexes have to be contracted and which ones can stay intact (witness, e.g., the potential redex $h\, v$ generated in the NH-LET-rule that does not have to be reduced eagerly).

## 5 Standardisation theorem

The standardisation theorem for $\beta$-reduction in the $\lambda$-calculus, in its classical form (Mitschke 1979), states that any reduction sequence that reaches a head-normal form can be separated into a head-reduction sequence that reaches a head-normal form, followed by a number of *internal* reductions. As noted by McKinna and Pollack (1999), this can be adapted to the more practically useful notion of weak-head reductions, and since the important consequences follow already from Mitschke's semi-standardisation, or factorisation, lemma, this is the problem we focus on in this section.

To prove standardisation, we adapt the methodology developed by Takahashi (1995) and refined by McKinna and Pollack (McKinna 2022; McKinna and Pollack 1999). The crux of the idea is to use *internal parallel* reductions to describe the internal (i.e., non-weak-head) reductions of the standard sequence. In its refined form, the *standard* reduction sequences can then be defined as a composition of weak-head reduction sequences with internal-parallel reductions,[5] and semi-standardisation simply states that any reduction sequence can be mimicked by a standard one.

While the methodology is very refined in the case of pure lambda calculus, the fact that our notion of evaluation (with which we want the weak-head reduction to coincide) is a hybrid strategy introduces new challenges. This is because the *top* contractions ((let.v) and (lift.v)),

[5] As a matter of fact, in the refined proof method one can use internal reductions that are sequential rather than parallel (McKinna 2022), but the definition and the proof become less concise.

$$\frac{a \twoheadrightarrow^{\mathrm{w}} \mathsf{return}\, v \quad e[v/x] \twoheadrightarrow^{\mathrm{w}} e'}{\mathsf{let}\, x = a\, \mathsf{in}\, e \twoheadrightarrow^{\mathrm{f}} e'}\ \textsc{f-let-ret} \qquad \frac{a \twoheadrightarrow^{\mathrm{w}} a'}{\mathsf{let}\, x = a\, \mathsf{in}\, e \twoheadrightarrow^{\mathrm{f}} \mathsf{let}\, x = a'\, \mathsf{in}\, e}\ \textsc{f-let}$$

$$\frac{e \twoheadrightarrow^{\mathrm{w}} \mathsf{return}\, v}{\mathsf{lift}\ e \twoheadrightarrow^{\mathrm{f}} v}\ \textsc{f-lift-ret} \qquad \frac{e \twoheadrightarrow^{\mathrm{w}} e'}{\mathsf{lift}\ e \twoheadrightarrow^{\mathrm{f}} \mathsf{lift}\ e'}\ \textsc{f-lift} \qquad \frac{}{\mathsf{return}\, v \twoheadrightarrow^{\mathrm{f}} \mathsf{return}\, v}\ \textsc{f-ret}$$

$$\frac{}{u\, v \twoheadrightarrow^{\mathrm{f}} u\, v}\ \textsc{f-app} \qquad \frac{}{\mathsf{do}\, v \twoheadrightarrow^{\mathrm{f}} \mathsf{do}\, v}\ \textsc{f-do}$$

$$\frac{}{\mathsf{handle}\, e\, \mathsf{with}\, h, r \twoheadrightarrow^{\mathrm{f}} \mathsf{handle}\, e\, \mathsf{with}\, h, r}\ \textsc{f-handle} \qquad \frac{e_1 \mapsto_{\mathrm{i}}^{*} e_2 \quad e_2 \twoheadrightarrow^{\mathrm{f}} e_3}{e_1 \twoheadrightarrow^{\mathrm{w}} e_3}\ \textsc{w}$$

$$\frac{}{(\lambda x.e)\, v \mapsto_{\mathrm{i}} e[v/x]}\ \textsc{h-app} \qquad \frac{}{\mathsf{handle}\, \mathsf{return}\, v\, \mathsf{with}\, h, r \mapsto_{\mathrm{i}} r\, v}\ \textsc{h-ret}$$

$$\frac{}{\mathsf{handle}\, \mathsf{do}\, v\, \mathsf{with}\, h, r \mapsto_{\mathrm{i}} \mathsf{let}\, f = h\, v\, \mathsf{in}\, f\, r}\ \textsc{h-do}$$

$$\frac{}{\mathsf{handle}\, \mathsf{lift}\ e\, \mathsf{with}\, h, r \mapsto_{\mathrm{i}} \mathsf{let}\, x = e\, \mathsf{in}\, r\, x}\ \textsc{h-lift}$$

$$\frac{}{\mathsf{handle}\, \mathsf{let}\, x = a\, \mathsf{in}\, e\, \mathsf{with}\, h, r \mapsto_{\mathrm{i}} \mathsf{handle}\, a\, \mathsf{with}\, h, (\lambda x\, .\, \mathsf{handle}\, e\, \mathsf{with}\, h, r)}\ \textsc{h-let}$$

$$\frac{e \mapsto_{\mathrm{i}} e'}{\mathsf{handle}\, e\, \mathsf{with}\, h, r \mapsto_{\mathrm{i}} \mathsf{handle}\, e'\, \mathsf{with}\, h, r}\ \textsc{h-cong}$$

$$\frac{}{x \twoheadrightarrow_{v}^{\mathrm{i}} x}\ \textsc{i-var} \qquad \frac{e_1 \twoheadrightarrow^{\mathrm{s}} e_2}{\lambda x.e_1 \twoheadrightarrow_{v}^{\mathrm{i}} \lambda x.e_2}\ \textsc{i-lam} \qquad \frac{v_1 \twoheadrightarrow_{v}^{\mathrm{i}} v_2}{\mathsf{return}\, v_1 \twoheadrightarrow_{e}^{\mathrm{i}} \mathsf{return}\, v_2}\ \textsc{i-ret}$$

$$\frac{a_1 \twoheadrightarrow_{e}^{\mathrm{i}} a_2 \quad e_1 \twoheadrightarrow^{\mathrm{s}} e_2}{\mathsf{let}\, x = a_1\, \mathsf{in}\, e_1 \twoheadrightarrow_{e}^{\mathrm{i}} \mathsf{let}\, x = a_2\, \mathsf{in}\, e_2}\ \textsc{i-let} \qquad \frac{e_1 \twoheadrightarrow_{e}^{\mathrm{i}} e_2}{\mathsf{lift}\ e_1 \twoheadrightarrow_{e}^{\mathrm{i}} \mathsf{lift}\ e_2}\ \textsc{i-lift}$$

$$\frac{v_1 \twoheadrightarrow_{v}^{\mathrm{i}} v_2}{\mathsf{do}\, v_1 \twoheadrightarrow_{e}^{\mathrm{i}} \mathsf{do}\, v_2}\ \textsc{i-do} \qquad \frac{u_1 \twoheadrightarrow_{v}^{\mathrm{i}} u_2 \quad v_1 \twoheadrightarrow_{v}^{\mathrm{i}} v_2}{u_1\, v_1 \twoheadrightarrow_{e}^{\mathrm{i}} u_2\, v_2}\ \textsc{i-app}$$

$$\frac{e_1 \twoheadrightarrow^{\mathrm{f}} e_2 \quad e_2 \twoheadrightarrow_{e}^{\mathrm{i}} e_3 \quad h_1 \twoheadrightarrow_{v}^{\mathrm{i}} h_2 \quad r_1 \twoheadrightarrow_{v}^{\mathrm{i}} r_2}{\mathsf{handle}\, e_1\, \mathsf{with}\, h_1, r_1 \twoheadrightarrow_{e}^{\mathrm{i}} \mathsf{handle}\, e_3\, \mathsf{with}\, h_2, r_2}\ \textsc{i-handle} \qquad \frac{e_1 \twoheadrightarrow^{\mathrm{w}} e_2 \quad e_2 \twoheadrightarrow_{e}^{\mathrm{i}} e_3}{e_1 \twoheadrightarrow^{\mathrm{s}} e_3}\ \textsc{s}$$

Fig. 10. Standard reduction ($\twoheadrightarrow^{\mathrm{s}}$), internal reduction ($\twoheadrightarrow_{e}^{\mathrm{i}}$)/($\twoheadrightarrow_{v}^{\mathrm{i}}$), inner reduction ($\mapsto_{\mathrm{i}}$), frozen weak head reduction ($\twoheadrightarrow^{\mathrm{f}}$), and weak head reduction ($\twoheadrightarrow^{\mathrm{w}}$).

while part of the weak-head reduction in appropriate contexts, become *frozen* if performed within a handle expression—and both let and lift constructs cease to act as evaluation contexts within handle. Thus, we lose a clear, structural distinction between evaluation positions,

where any weak-head reductions remain weak-head, and the remaining positions, where no weak-head reduction is possible.

These considerations led us to the definitions of weak-head, frozen, internal-parallel and standard reductions presented in Figure 10. To obtain a well-behaved definition of frozen reductions we repeat McKinna's observation that standard sequences can be *defined* as composition of weak-head and internal reductions and adapt it as a definition of weak-head sequences: these are now formed as a composition of *inner* reductions ($\mapsto_{\mathsf{i}}^{*}$), which are always allowed within a handle expression, and the frozen reductions. The latter are defined to be reflexive and encompass weak-head reduction sequences under let, lift, and those including the two *top* contractions. It is easily shown that this notion of weak-head reduction correponds to the notion of fine-grained evaluation ($\rightarrow_{\mathsf{fg}}^{*}$) defined in Section 2.

This leaves us with the definition of standard and internal-parallel reductions. These follow the recent presentation by McKinna (2022), adapted to the fine-grained call-by-value setting. Thus, we get the notions of internal-parallel reductions for both expressions ($\twoheadrightarrow_{e}^{\mathsf{i}}$) and values ($\twoheadrightarrow_{v}^{\mathsf{i}}$), with the definitions following the structure of terms and allowing:

- *internal* reductions for any value sub-terms;
- *standard* reductions for any expression sub-terms outside of the evaluation position;
- *internal* reductions for expression sub-terms in evaluation position of let and lift;
- *frozen* reductions followed by *internal* reductions for the evaluation position of handle.

The last case accounts for the hybrid nature of the weak-head reduction, as discussed above. Finally, the standard sequences are defined as composition of weak-head and internal reductions.

The crucial step in the methodology we follow is establishing transitivity of the relations defined as above. For this, it suffices to know that standard reductions *absorb* contractions and weak-head reductions from the right (cf. Lemma 5.5 for finer detail). This is the heart of the argument that reductions can be appropriately reordered, and is always somewhat involved, as internal reductions can become unblocked through the commutation of the weak-head reduction in front of them. Most of these cases are handled in a standard way: what remains are the frozen reductions in the evaluation position of handle, which can also get unblocked by the following reduction. This leads us to the following four lemmas, corresponding to the four contractions of handle.

Lemma 5.1 (Handling of values). *The following implications hold:*

(1) *If* $e \twoheadrightarrow^{\mathrm{f}} \mathsf{return}\, v$, *then* $\mathsf{handle}\, e\, \mathsf{with}\, h, r \twoheadrightarrow^{\mathrm{w}} r\, v$.
(2) *If* $e \twoheadrightarrow^{\mathrm{w}} \mathsf{return}\, v$, *then* $\mathsf{handle}\, e\, \mathsf{with}\, h, r \twoheadrightarrow^{\mathrm{w}} r\, v$.

Lemma 5.2 (Handling of effects). *The following implications hold:*

(1) *If* $e \twoheadrightarrow^{\mathrm{f}} \mathsf{do}\, v$, *then* $\mathsf{handle}\, e\, \mathsf{with}\, h, r \twoheadrightarrow^{\mathrm{w}} \mathsf{let}\, x = h\, v\, \mathsf{in}\, x\, r$.
(2) *If* $e \twoheadrightarrow^{\mathrm{w}} \mathsf{do}\, v$, *then* $\mathsf{handle}\, e\, \mathsf{with}\, h, r \twoheadrightarrow^{\mathrm{w}} \mathsf{let}\, x = h\, v\, \mathsf{in}\, x\, r$.

Lemma 5.3 (Handling of lifts). *The following implications hold:*

(1) *If* $e \twoheadrightarrow^{\mathrm{f}} \mathsf{lift}\, e'$, *then* $\mathsf{handle}\, e\, \mathsf{with}\, h, r \twoheadrightarrow^{\mathrm{w}} \mathsf{let}\, x = e'\, \mathsf{in}\, r\, x$.
(2) *If* $e \twoheadrightarrow^{\mathrm{w}} \mathsf{lift}\, e'$, *then* $\mathsf{handle}\, e\, \mathsf{with}\, h, r \twoheadrightarrow^{\mathrm{w}} \mathsf{let}\, x = e'\, \mathsf{in}\, r\, x$.

Lemma 5.4 (Handling of let-expressions). *The following implications hold:*

(1) *If* $e \twoheadrightarrow^{\mathrm{f}} \mathsf{let}\, x = a \,\mathsf{in}\, e'$, *then there exists* $a'$ *such that* $\mathsf{handle}\, e \,\mathsf{with}\, h, r \twoheadrightarrow^{\mathrm{w}} \mathsf{handle}\, a' \,\mathsf{with}\, h, \lambda x.\mathsf{handle}\, e' \,\mathsf{with}\, h, r$ *and* $a' \twoheadrightarrow^{\mathrm{f}} a$.
(2) *If* $e \twoheadrightarrow^{\mathrm{w}} \mathsf{let}\, x = a \,\mathsf{in}\, e'$, *then there exists* $a'$ *such that* $\mathsf{handle}\, e \,\mathsf{with}\, h, r \twoheadrightarrow^{\mathrm{w}} \mathsf{handle}\, a' \,\mathsf{with}\, h, \lambda x.\mathsf{handle}\, e' \,\mathsf{with}\, h, r$ *and* $a' \twoheadrightarrow^{\mathrm{f}} a$.

In the first three cases, these simply state that a frozen reduction under handle followed by an appropriate contraction can be simulated with a weak-head reduction sequence. In the final one, however, we separate the original reduction sequence into two parts: a reduction sequence that exposes a let-expression, and any frozen reductions within its evaluation position. While the former sequence can be followed with a weak-head reduction, the latter become internal to the resulting handler when absorbed as part of the standard reduction. Note that these lemmas correspond to some of the critical pairs in the reduction theory, which we needed to consider in the previous section: in a sense, they provide a resolution of these critical pairs in concrete cases that arise from evaluation. All four of these lemmas follow by routine mutual induction on the structure of the weak-head and frozen derivations, suggesting that our definitions fit well with the methodology we used.

Having handled the unfreezing of reductions frozen within the handle expression, we can now return to establishing transitivity of internal and standard reduction relations. We prove, in turn, that internal reductions on expressions absorb contractions, weak-head and frozen reductions from the right. Most of these give rise to standard reduction sequences, but we establish more precise commutation results for *top* contractions and frozen reductions. These allow us to establish transitivity for internal reductions on expressions, where a lack of precision would only give us standard reduction sequence as a result. While not strictly necessary, this is a good litmus test of our definition of internal reductions for expressions. The following lemma summarizes the transitivity results.

Lemma 5.5 (Recomposition). *For all* $e_1$, $e_2$, $e_3$, $v_1$, $v_2$, *and* $v_3$:

(1) *If* $e_1 \twoheadrightarrow^{\mathrm{i}}_{e} e_2 \mapsto_{c} e_3$, *then* $e_1 \twoheadrightarrow^{\mathrm{s}} e_3$.
(2) *If* $e_1 \twoheadrightarrow^{\mathrm{i}}_{e} e_2 \mapsto_{f} e_3$, *then* $e_1 \twoheadrightarrow^{\mathrm{s}} e_3$.
(3) *If* $e_1 \twoheadrightarrow^{\mathrm{i}}_{e} e_2 \mapsto_{t} e_3$, *then there exists* $e_2'$, *such that* $e_1 \twoheadrightarrow^{\mathrm{f}} e_2' \twoheadrightarrow^{\mathrm{i}}_{e} e_3$.
(4) *If* $e_1 \twoheadrightarrow^{\mathrm{i}}_{e} e_2 \mapsto_{\mathrm{i}} e_3$, *then* $e_1 \twoheadrightarrow^{\mathrm{s}} e_3$.
(5) *If* $e_1 \twoheadrightarrow^{\mathrm{i}}_{e} e_2 \mapsto^{*}_{\mathrm{i}} e_3$, *then* $e_1 \twoheadrightarrow^{\mathrm{s}} e_3$.
(6) *If* $e_1 \twoheadrightarrow^{\mathrm{i}}_{e} e_2 \twoheadrightarrow^{\mathrm{f}} e_3$, *then there exists* $e_2'$, *such that* $e_1 \twoheadrightarrow^{\mathrm{f}} e_2' \twoheadrightarrow^{\mathrm{i}}_{e} e_3$.
(7) *If* $e_1 \twoheadrightarrow^{\mathrm{i}}_{e} e_2 \twoheadrightarrow^{\mathrm{w}} e_3$, *then* $e_1 \twoheadrightarrow^{\mathrm{s}} e_3$.
(8) *If* $e_1 \twoheadrightarrow^{\mathrm{i}}_{e} e_2 \twoheadrightarrow^{\mathrm{i}}_{e} e_3$, *then* $e_1 \twoheadrightarrow^{\mathrm{i}}_{e} e_3$.
(9) *If* $v_1 \twoheadrightarrow^{\mathrm{i}}_{v} v_2 \twoheadrightarrow^{\mathrm{i}}_{v} v_3$, *then* $v_1 \twoheadrightarrow^{\mathrm{i}}_{v} v_3$.
(10) *If* $e_1 \twoheadrightarrow^{\mathrm{s}} e_2 \twoheadrightarrow^{\mathrm{s}} e_3$, *then* $e_1 \twoheadrightarrow^{\mathrm{s}} e_3$.

With transitivity established, the semi-standardisation theorem follows.

Theorem 5.6 (Semi-standardisation). *If* $e \longrightarrow^{*}_{\lambda_{\mathrm{fg}}} e'$, *then* $e \twoheadrightarrow^{\mathrm{s}} e'$.

Proof. Since standard reduction is reflexive and transitive, it remains to show that single-step reduction is a subrelation of standard reduction, which follows by simple induction. □

This is, to the best of our knowledge, the first explicitly stated standardisation result for a theory of effect handlers. Having said that, a careful reading of McLaughlin's proof of the Context Lemma for Ella (McLaughlin 2020, Chapter 6), a core calculus of the Frank programming language (Convent et al. 2020) and thus substantially different in terms of

structuring and handling effects, reveals the structure of weak-head and internal reductions characteristic to standardisation.

## 6 Equational reasoning

The results presented thus far license the fine-grained reduction theory we have developed to be used as a foundation for an equational theory that is sound with respect to the standard context capturing semantics. To formally state this, let us define $=_{\lambda_{\mathsf{fg}}}$ as a symmetric closure of $\longrightarrow^*_{\lambda_{\mathsf{fg}}}$. Let us also define the notion of observational equivalence $\approx_{\mathsf{cc}}$ as follows: $e_1 \approx_{\mathsf{cc}} e_2$ ($e_1$ is observationally equivalent to $e_2$), if for all contexts $C$ such that both $C[e_1]$ and $C[e_2]$ are closed terms, $C[e_1] \to^*_{\mathsf{cc}} \mathsf{return}\, v_1$ for some value $v_1$ if and only if $C[e_2] \to^*_{\mathsf{cc}} \mathsf{return}\, v_2$ for some value $v_2$.[6] Then the following theorem holds.

Theorem 6.1 (Soundness of fine-grained equational reasoning). *If $e_1 =_{\lambda_{\mathsf{fg}}} e_2$, then $e_1 \approx_{\mathsf{cc}} e_2$.*

Proof. Let us assume that $e_1 =_{\lambda_{\mathsf{fg}}} e_2$ and $C[e_1] \to^*_{\mathsf{cc}} v_1$ (the symmetric case is proved analogously). From the former assumption, we know that $C[e_1] =_{\lambda_{\mathsf{fg}}} C[e_2]$. From the latter assumption, we get that $C[e_1] \to^*_{\mathsf{fg}} v_1'$ by the simulation of context capturing evaluation (Theorem 3.1). Therefore, we have that $C[e_2] =_{\lambda_{\mathsf{fg}}} v_1'$, which by the confluence of $\longrightarrow_{\lambda_{\mathsf{fg}}}$ (Theorem 4.1) gives us that $C[e_2] \longrightarrow^*_{\lambda_{\mathsf{fg}}} v_2''$ for some value $v_2''$. Now, applying the standardisation theorem (Theorem 5.6), we deduce that $C[e_2] \to^*_{\mathsf{fg}} v_1'$ for some value $v_1'$. Finally, by the simulation of fine-grained evaluation (Theorem 3.2), we arrive at the conclusion $C[e_2] \to^*_{\mathsf{cc}} v_2$ for some value $v_2$. □

It follows from Theorem 6.1 that the directed reduction theory defined as $\longrightarrow_{\lambda_{\mathsf{fg}}}$ and understood as a program optimization, as in the motivating example in Section 1, is sound.

## 7 Normal forms and normalization

One of the useful consequences of the introduction of a reduction theory for effect handlers is the ability to study its normal forms and normalization processes. There are two traditional applications of such results: firstly, in partial evaluation, or specialisation of programs—such as the example presented in the introduction—and as a means to determine convertibility in dependent type theories. While only the first of these is directly applicable to our current setup (due to its untyped character), we believe that the results presented below may be useful in attempts to marry effects and effect handlers with dependent types.

### *7.1 Normal forms*

We begin by characterising normal forms of our calculus, presented in Figure 11. The normal expressions ($\mu$) consist of either normal values ($\xi$) or *neutral expressions* ($\nu$), with normal values encompassing variables and lambda-abstractions with normalized bodies. The remainder of our normal forms are the neutral expressions: expressions that are either control-stuck, i.e., containing a $\mathsf{do}$ expression unenclosed by a handle, or open stuck, i.e., containing an application of a variable to a (normalized) value in a (normalized) evaluation

[6] It can be shown, straightforwardly adjusting the argument from (Biernacki et al. 2020), that this notion of observational equivalence coincides with the standard one, in which termination, rather than termination to a value, is observed. Moreover, instead of $\approx_{\mathsf{cc}}$, we could introduce a notion of observational equivalence based directly on the fine-grained semantics, since thanks to the simulation theorems of Section 3 the two notions coincide.

$$
\begin{array}{lll}
\mu & ::= & \mathsf{return}\,\xi \mid \nu_{\mathsf{T}} \qquad \text{(normal expressions)} \\
\xi & ::= & x \mid \lambda x.\mu \qquad \text{(normal values)} \\
\nu_k & ::= & {}_{(k=\mathsf{T})}\ \mathsf{do}\ \xi \mid {}_{(k=\mathsf{T})}\ \mathsf{let}\, x = \nu_{\mathsf{T}}\, \mathsf{in}\, \mu \mid {}_{(k=\mathsf{T})}\ \mathsf{lift}\ \nu_{\mathsf{T}} \mid x\,\xi \mid \mathsf{handle}\ \nu_{\mathsf{H}}\ \mathsf{with}\ \xi, \xi' \\
& & \qquad\qquad \text{(neutral expressions)}
\end{array}
$$

Fig. 11. Normal forms for fine-grained reduction.

context. The distinction in the allowable contexts is achieved, by indexing them with their kind (T or H), and preventing the appearance of intermediate values (T-kinded) if there are handlers present (any H-kinded neutral expression, however, is also a T-kinded one). Note that this description is somewhat removed from a traditional separation into normal and neutral forms of terms, where the neutral class encompasses both elimination forms and variables. In our case, variables $x$ and their eliminations $x\,\xi$ belong to two different classes. This is due to their distinct reduction behaviour: the term $\mathsf{let}\, x = y\, v\, \mathsf{in}\, e$ cannot evaluate further (although reductions within $v$ or $e$ may still be allowed), while $\mathsf{let}\, x = \mathsf{return}\, y\, \mathsf{in}\, e$ is a valid redex.[7] We cannot extend the notion of substitutions and allow the first computation to reduce, since a substitution of certain values for $y$ may cause effects—and in such a case the behaviour of a hypothetical term $e[y\, v/x]$ could easily differ from the original! Therefore, the let-bindings, lift and handle expressions that organise the computation need to be preserved if they enclose a neutral expression, but not if they enclose a value: a substitution in a value cannot turn a stuck term into a redex.

We formalize the notion that our definitions characterise normal forms by the following lemmas (note that $\mu$ and $\nu$ trivially inject into expressions, and $\xi$ into values):

Lemma 7.1. *The sets $\mu$ and $\xi$ are precisely normal forms of expressions and values, respectively, with respect to fine-grained reduction $\longrightarrow_{\lambda_{\mathsf{fg}}}$, i.e.,*

(1) *Normal forms ($\mu, \xi, \nu$) do not reduce;*
(2) *Any expression $e$ (respectively, value $v$) either reduces to some expression $e'$ (value $v'$) or is a normal form $\mu = e$ ($\xi = v$).*

### 7.2 Normalization

Since the equational theory of our calculus is given through explicitly directed, confluent reductions, it is natural to compute normal forms of expressions (where they exist, as the calculus is, in general, non-terminating) through a combination of open-term evaluation and readback functions (Grégoire and Leroy 2002). We begin by presenting the normalizer as the graph of a partial function, implementable as an inductively-defined relation that corresponds to our formalization in Rocq, which in fact can be interpreted as an environment-based abstract machine. In Appendix A, in turn, we show how this abstract machine can be implemented in a general-purpose programming language, such as OCaml, and we derive from it, using the functional correspondence (Ager et al. 2003), the corresponding higher-order compositional normalizer. The definition of the first-order normalizer we present in this section follows closely the structure of the syntactic abstract machine of Section 2.3. However, since it

[7] Such considerations are typical of call-by-value calculi in which reduction on open terms is allowed (Accattoli and Guerrieri 2016; Lassen 2005).

$$
\begin{array}{rcll}
\mathsf{SVal} \ni \mathsf{v} & ::= & x \mid \mathsf{f} & \text{(semantic values)} \\
\mathsf{SFun} \ni \mathsf{f} & ::= & [x, e, \rho] \mid \mathsf{v}- \mid -\mathsf{v} \mid \mathsf{H}(\mathsf{f}, \mathsf{v}, \mathsf{v}') & \text{(semantic functions)} \\
\rho \in \mathsf{SEnv} & \triangleq & \mathsf{SVal}^{\mathrm{Var}} & \text{(environments)} \\
\mathsf{SComp} \ni \mathsf{c} & ::= & [e, \rho] \mid \mathsf{v}\,\mathsf{v}' & \text{(closures/semantic computations)} \\
\mathsf{SIVal} \ni \mathsf{i} & ::= & \mathsf{IRet}(\mathsf{v}) \mid \mathsf{ILet}(\mathsf{c}, \mathsf{f}) \mid \mathsf{IDo}(\mathsf{v}) \mid \mathsf{ILift}(\mathsf{c}) & \text{(intermediate values)} \\
\mathsf{SCont} \ni \mathsf{E}_k & ::= & {}_{(k=\mathsf{T})}\,[\,] \mid {}_{(k=\mathsf{T})}\,\mathsf{E}_\mathsf{T}[\mathsf{let}\,\mathsf{f}] \mid {}_{(k=\mathsf{T})}\,\mathsf{E}_\mathsf{T}[\mathsf{lift}] \mid {}_{(k=\mathsf{H})}\,\mathsf{E}[\mathsf{handle}\,\mathsf{v}_h\,\mathsf{v}_r] & \text{(stacks)} \\
\mathsf{SAns} \ni \mathsf{a} & ::= & \mathsf{v} \mid \mathsf{E}_\mathsf{T}[\mathsf{do}\,\mathsf{v}] \mid \mathsf{E}[x\,\mathsf{v}] & \text{(answers)} \\
\mathsf{Conf} \ni \mathsf{C} & ::= & \langle \mathsf{c} \rangle_\mathrm{E} \mid \langle \mathsf{v}_1, \mathsf{v}_2 \rangle_\mathrm{A} \mid \langle \mathsf{f}, \mathsf{v} \rangle_\mathrm{F} \mid \langle \mathsf{i} \rangle_\mathrm{K} \mid \langle \mathsf{i}, \mathsf{v}_h, \mathsf{v}_r \rangle_\mathrm{H} \mid \langle \mathsf{i} \rangle_\mathrm{T} \mid \langle \mathsf{v} \rangle_\mathrm{VT} & \text{(configurations)}
\end{array}
$$

Fig. 12. Syntax instrumented for the environment-based semantics.

utilises environment semantics rather than substitutions—which makes it a more realistic model of normalization, amenable to refunctionalization of the function space comprising the values the normalizer generates and manipulates—it requires an instrumentation with some semantic artifacts that we introduce first.

**Semantic constructs.** The necessary components of an environment-based evaluator comprise semantic values, *functions* (lambda-like values), environments, closures (computations), semantic intermediate values, stacks, and answers; these are presented in Figure 12. Note that syntactic expressions appear only as parts of the closures, and the remainder of the instrumentation is achieved through simple inductive definitions. We represent environments as total functions from variables to semantic values, with $\rho[x \mapsto \mathsf{v}]$ standing for the environment mapping $x$ to $\mathsf{v}$ and otherwise equal to $\rho$.[8] For the sake of uniformity, the evaluation will operate on pairs of stacks and *configurations*, and produce an answer as a result. The correspondence with the syntactic abstract machine is rather direct: configurations E correspond to the configurations e of the syntactic abstract machine, configurations A and F correspond to configurations a (unlike the evaluator of this section, the syntactic abstract machine is assumed to operate on closed expressions, so there is no need for a special treatment of free variables), configurations K correspond to configurations k, and so on. The role of intermediate values is played by semantic intermediate values.

Note that additional semantic functions and computations are present, beyond the expected closures for functions and expressions. These arise as semantic counterparts of the code that reduction rules create, as will become apparent once we see the evaluator. They can also be thought of as semantic continuations representing the syntactic continuations that the syntactic abstract machine encodes as terms in the last three transitions from the h-mode, e.g., in the transition

$$\langle \mathsf{do}\,v \mid h \mid r \mid E \rangle_\mathrm{h} \triangleright \langle E \mid \mathsf{let}\,x = h\,v\,\mathsf{in}\,x\,r \rangle_\mathrm{k}$$

[8]The precise implementation of the environments depends on the representation of variables, which we abstract from in this section.

$$
\begin{aligned}
\lfloor - \rfloor_{\mathrm{v}} &: \mathrm{SVal} \to \mathrm{Val} \\
\lfloor x \rfloor_{\mathrm{v}} &= x \\
\lfloor \mathsf{f} \rfloor_{\mathrm{v}} &= \lambda x.\lfloor \mathsf{f} \rfloor_{\mathrm{f}}^{x}
\end{aligned}
\qquad
\begin{aligned}
\lfloor - \rfloor_{\mathrm{c}} &: \mathrm{SComp} \to \mathrm{Exp} \\
\lfloor [e, \rho] \rfloor_{\mathrm{c}} &= (z \mapsto \lfloor \rho(z) \rfloor_{\mathrm{v}})(e) \\
\lfloor \mathsf{v}\, \mathsf{v}' \rfloor_{\mathrm{c}} &= \lfloor \mathsf{v} \rfloor_{\mathrm{v}}\, \lfloor \mathsf{v}' \rfloor_{\mathrm{v}}
\end{aligned}
$$

$$
\begin{aligned}
\lfloor - \rfloor_{\mathrm{f}}^{-} &: \mathrm{SFun} \times \mathrm{Var} \to \mathrm{Exp} \\
\lfloor [x, e, \rho] \rfloor_{\mathrm{f}}^{y} &= ((z \mapsto \lfloor \rho(z) \rfloor_{\mathrm{v}})[x \mapsto y])(e) \\
\lfloor \mathsf{v} - \rfloor_{\mathrm{f}}^{x} &= \lfloor \mathsf{v} \rfloor_{\mathrm{v}}\, x \\
\lfloor - \mathsf{v} \rfloor_{\mathrm{f}}^{x} &= x\, \lfloor \mathsf{v} \rfloor_{\mathrm{v}} \\
\lfloor \mathsf{H}(\mathsf{f}, \mathsf{v}, \mathsf{v}') \rfloor_{\mathrm{f}}^{x} &= \mathsf{handle}\, \lfloor \mathsf{f} \rfloor_{\mathrm{f}}^{x}\, \mathsf{with} \lfloor \mathsf{v} \rfloor_{\mathrm{v}}, \lfloor \mathsf{v}' \rfloor_{\mathrm{v}}
\end{aligned}
\qquad
\begin{aligned}
\lfloor - \rfloor_{\mathrm{C}} &: \mathrm{Conf} \to \mathrm{Exp} \\
\lfloor \langle \mathsf{c} \rangle_{\mathrm{E}} \rfloor_{\mathrm{C}} &= \lfloor \mathsf{c} \rfloor_{\mathrm{c}} \\
\lfloor \langle \mathsf{v}_1, \mathsf{v}_2 \rangle_{\mathrm{A}} \rfloor_{\mathrm{C}} &= \lfloor \mathsf{v}_1 \rfloor_{\mathrm{v}}\, \lfloor \mathsf{v}_2 \rfloor_{\mathrm{v}} \\
\lfloor \langle \mathsf{f}, \mathsf{v} \rangle_{\mathrm{F}} \rfloor_{\mathrm{C}} &= \lfloor \mathsf{f} \rfloor_{\mathrm{f}}^{x} [\lfloor \mathsf{v} \rfloor_{\mathrm{v}}/x] \\
\lfloor \langle \mathsf{i} \rangle_{\mathrm{K}} \rfloor_{\mathrm{C}} &= \lfloor \mathsf{i} \rfloor_{\mathrm{i}} \\
\lfloor \langle \mathsf{i}, \mathsf{v}_h, \mathsf{v}_r \rangle_{\mathrm{H}} \rfloor_{\mathrm{C}} &= \mathsf{handle}\, \lfloor \mathsf{i} \rfloor_{\mathrm{i}}\, \mathsf{with} \lfloor \mathsf{v}_h \rfloor_{\mathrm{v}}, \lfloor \mathsf{v}_r \rfloor_{\mathrm{v}} \\
\lfloor \langle \mathsf{i} \rangle_{\mathrm{T}} \rfloor_{\mathrm{C}} &= \lfloor \mathsf{i} \rfloor_{\mathrm{i}} \\
\lfloor \langle \mathsf{v} \rangle_{\mathrm{VT}} \rfloor_{\mathrm{C}} &= \lfloor \mathsf{v} \rfloor_{\mathrm{v}}
\end{aligned}
$$

$$
\begin{aligned}
\lfloor - \rfloor_{\mathrm{i}} &: \mathrm{SIVal} \to \mathrm{Exp} \\
\lfloor \mathsf{IRet}(\mathsf{v}) \rfloor_{\mathrm{i}} &= \mathsf{return}\, \lfloor \mathsf{v} \rfloor_{\mathrm{v}} \\
\lfloor \mathsf{ILet}(\mathsf{c}, \mathsf{f}) \rfloor_{\mathrm{i}} &= \mathsf{let}\, x = \lfloor \mathsf{c} \rfloor_{\mathrm{c}}\, \mathsf{in} \lfloor \mathsf{f} \rfloor_{\mathrm{f}}^{x} \\
\lfloor \mathsf{IDo}(\mathsf{v}) \rfloor_{\mathrm{i}} &= \mathsf{do}\, \lfloor \mathsf{v} \rfloor_{\mathrm{v}} \\
\lfloor \mathsf{ILift}(\mathsf{c}) \rfloor_{\mathrm{i}} &= \mathsf{lift}\, \lfloor \mathsf{c} \rfloor_{\mathrm{c}}
\end{aligned}
\qquad
\begin{aligned}
\lfloor - \rfloor_{\mathrm{E}} &: \mathrm{SCont} \to \mathrm{Cont} \\
\lfloor [\,] \rfloor_{\mathrm{E}} &= [\,] \\
\lfloor \mathsf{E}_{\mathsf{T}}[\mathsf{let}\, \mathsf{f}] \rfloor_{\mathrm{E}} &= \lfloor \mathsf{E} \rfloor_{\mathrm{E}} [\mathsf{let}\, x = [\,]\, \mathsf{in} \lfloor \mathsf{f} \rfloor_{\mathrm{f}}^{x}] \\
\lfloor \mathsf{E}_{\mathsf{T}}[\mathsf{lift}] \rfloor_{\mathrm{E}} &= \lfloor \mathsf{E}_{\mathsf{T}} \rfloor_{\mathrm{E}} [\mathsf{lift}\ [\,]] \\
\lfloor \mathsf{E}[\mathsf{handle}\, \mathsf{v}_h\, \mathsf{v}_r] \rfloor_{\mathrm{E}} &= \\
&\quad \lfloor \mathsf{E} \rfloor_{\mathrm{E}} [\mathsf{handle}\, [\,]\, \mathsf{with} \lfloor \mathsf{v}_h \rfloor_{\mathrm{v}}, \lfloor \mathsf{v}_r \rfloor_{\mathrm{v}}]
\end{aligned}
$$

$$
\begin{aligned}
\lfloor - \rfloor_{\mathrm{a}} &: \mathrm{SAns} \to \mathrm{Exp} \\
\lfloor \mathsf{v} \rfloor_{\mathrm{a}} &= \mathsf{return}\, \lfloor \mathsf{v} \rfloor_{\mathrm{v}} \\
\lfloor \mathsf{E}_{\mathsf{T}}[\mathsf{do}\, \mathsf{v}] \rfloor_{\mathrm{a}} &= \lfloor \mathsf{E}_{\mathsf{T}} \rfloor_{\mathrm{E}} [\mathsf{do}\, \lfloor \mathsf{v} \rfloor_{\mathrm{v}}] \\
\lfloor \mathsf{E}[x\, \mathsf{v}] \rfloor_{\mathrm{a}} &= \lfloor \mathsf{E} \rfloor_{\mathrm{E}} [x\, \lfloor \mathsf{v} \rfloor_{\mathrm{v}}]
\end{aligned}
$$

Fig. 13. Erasure of semantic components of the evaluator.

the right hand side encodes a computation $h\, v$ with a local continuation $x\, r$, where $x$ is the result of the computation $h\, v$. In the semantic evaluator this local continuation is represented as $-\mathsf{v}_r$, where $\mathsf{v}_r$ represents $r$. Similarly, the other two h-transitions in the syntactic abstract machine introduce local continuations which are represented as $\mathsf{v}-$ and $\mathsf{H}(\mathsf{f}, \mathsf{v}, \mathsf{v}')$, respectively.

Figure 13 we present erasure functions for semantic values, functions, computations, semantic intermediate values, and configurations that map these into values or expressions, as appropriate. These can be naturally extended to the stacks (which share the structure with evaluation contexts) and answers. The erasure functions will be useful in the following, but, hopefully, they also provide a better intuition for the interpretation of the semantic constructs.

There are three points worth mentioning here. First, in order to translate a local continuation expressed as a functional value $\mathsf{f}$, a fresh variable $x$ denoting the formal parameter of the continuation is generated and the body of the continuation is translated with $x$ as a parameter. This requires value closures $[y, e, \rho]$ to have their bound variable $y$ renamed to $x$—a consequence of placing both the expressible values (value closures) and local continuations within the same data type. Second, translating closures $[e, \rho]$ (and $[y, e, \rho]$) away is achieved by constructing a substitution $(z \mapsto \lfloor \rho(z) \rfloor_{\mathrm{v}})$ that maps variables to syntactic values and then mapping this substitution over the body of the closure, written $(z \mapsto \lfloor \rho(z) \rfloor_{\mathrm{v}})(e)$. Third, the

$$\mathsf{V} : \mathrm{Val} \times \mathrm{SEnv} \to \mathrm{SVal} \qquad \mathsf{I} : \mathrm{IVal} \times \mathrm{SEnv} \to \mathrm{SIVal}$$

$$\mathsf{V}(x, \rho) = \rho(x) \qquad \mathsf{I}(\mathsf{return}\, v, \rho) = \mathsf{IRet}(\mathsf{V}(v, \rho))$$

$$\mathsf{V}(\lambda x.e, \rho) = [x, e, \rho] \qquad \mathsf{I}(\mathsf{let}\, x = e_1\, \mathsf{in}\, e_2, \rho) = \mathsf{ILet}([\mathsf{e}_1, \rho], [\mathsf{x}, \mathsf{e}_2, \rho])$$

$$\mathsf{I}(\mathsf{do}\ v, \rho) = \mathsf{IDo}(\mathsf{V}(\mathsf{v}, \rho))$$

$$\mathsf{I}(\mathsf{lift}\ e, \rho) = \mathsf{ILift}([\mathsf{v}, \rho])$$

Fig. 14. Interpretation of values and intermediate values.

$\langle \mathsf{f}, \mathsf{v} \rangle_{\mathrm{F}}$ configurations are translated by applying the translation $\mathsf{f}$ to the translation of $\mathsf{v}$, which reflects the role of F-configurations.

**Evaluation.** With the necessary semantic components defined, we now turn to the definition of the evaluation and read-back functions. Since both of these are partial, we present them via their graphs, as inductively defined predicates that are functional. In this, we follow some of the spirit of Abel's presentation of untyped normalization-by-evaluation via partial applicative structures (Abel 2013), although due to the call-by-value nature of the reduction theory and the presence of control operators the spaces defined above do not form partial applicative structures as such.

The evaluation, presented in Figure 15, is an abstract-machine-style big-step semantics, reminiscent of Harper *et al.*'s account of call/cc in SML (Harper et al. 1993). Moreover, the inductive rules very closely follow the transitions of the syntactic abstract machine, e.g., the rule

$$\frac{\langle \mathsf{ILet}(\mathsf{v}_h\, \mathsf{v}, -\mathsf{v}_r) \rangle_{\mathrm{K}}, \mathsf{E} \searrow \mathsf{a}}{\langle \mathsf{IDo}(\mathsf{v}), \mathsf{v}_h, \mathsf{v}_r \rangle_{\mathrm{H}}, \mathsf{E} \searrow \mathsf{a}}$$

when read bottom up can be viewed as the following transition of an abstract machine

$$\langle \mathsf{IDo}(\mathsf{v}), \mathsf{v}_h, \mathsf{v}_r \rangle_{\mathrm{H}}, \mathsf{E} \triangleright \langle \mathsf{ILet}(\mathsf{v}_h\, \mathsf{v}, -\mathsf{v}_r) \rangle_{\mathrm{K}}, \mathsf{E}$$

which is a counterpart of the corresponding transition in the syntactic abstract machine. In other words, the evaluator of Figure 15 has been obtained from the syntactic abstract machine by introducing the environment, replacing purely syntactic entities with their environment-instrumented counterparts, and replacing machine transitions with inductive rules that mimick the transitions.

The evaluation, therefore, proceeds as in the syntactic abstract machine by inspecting the configuration and—in the case of the K- and VT-configurations—the stack. The eval configuration (denoted by E) inspects the closure, decomposes the expression, taking advantage of two auxiliary interpretation functions $\mathsf{V}$ and $\mathsf{I}$ shown in Figure 14, and chooses either the continuation or application configuration. The apply configuration (A) either recognises that the computation is open stuck, producing an appropriate answer, or continues with the application of the function. The function-application configuration (F) proceeds depending on the shape of the function: if it is a closure, the argument is added to the environment, and the evaluation proceeds in the usual fashion (this step corresponds to $(\beta.\mathsf{v})$ reduction). The two "application" functions apply one of the arguments to the other,

$$\frac{\langle \mathsf{I}(i,\rho)\rangle_{\mathrm{K}},\mathsf{E} \searrow \mathsf{a}}{\langle [i,\rho]\rangle_{\mathrm{E}},\mathsf{E} \searrow \mathsf{a}} \qquad \frac{\langle \mathsf{V}(v_1,\rho),\mathsf{V}(v_2,\rho)\rangle_{\mathrm{A}},\mathsf{E} \searrow \mathsf{a}}{\langle [v_1\ v_2,\rho]\rangle_{\mathrm{E}},\mathsf{E} \searrow \mathsf{a}}$$

$$\frac{\langle [e,\rho]\rangle_{\mathrm{E}},\mathsf{E}[\mathsf{handle}\,\mathsf{V}(h,\rho)\,\mathsf{V}(r,\rho)] \searrow \mathsf{a}}{\langle [(\mathsf{handle}\,e\,\mathsf{with}\,h,r),\rho]\rangle_{\mathrm{E}},\mathsf{E} \searrow \mathsf{a}} \qquad \frac{\langle \mathsf{v}_1,\mathsf{v}_2\rangle_{\mathrm{A}},\mathsf{E} \searrow \mathsf{a}}{\langle \mathsf{v}_1\ \mathsf{v}_2\rangle_{\mathrm{E}},\mathsf{E} \searrow \mathsf{a}}$$

$$\frac{}{\langle x,\mathsf{v}\rangle_{\mathrm{A}},\mathsf{E} \searrow \mathsf{E}[x\ \mathsf{v}]} \qquad \frac{\langle \mathsf{f},\mathsf{v}\rangle_{\mathrm{F}},\mathsf{E} \searrow \mathsf{a}}{\langle \mathsf{f},\mathsf{v}\rangle_{\mathrm{A}},\mathsf{E} \searrow \mathsf{a}}$$

$$\frac{\langle [e,\rho[x \mapsto \mathsf{v}]]\rangle_{\mathrm{E}},\mathsf{E} \searrow \mathsf{a}}{\langle [x,e,\rho],\mathsf{v}\rangle_{\mathrm{F}},\mathsf{E} \searrow \mathsf{a}} \qquad \frac{\langle \mathsf{v}_1,\mathsf{v}_2\rangle_{\mathrm{A}},\mathsf{E} \searrow \mathsf{a}}{\langle -\mathsf{v}_2,\mathsf{v}_1\rangle_{\mathrm{F}},\mathsf{E} \searrow \mathsf{a}} \qquad \frac{\langle \mathsf{f},\mathsf{v}\rangle_{\mathrm{F}},\mathsf{E}[\mathsf{handle}\,\mathsf{v}_1\,\mathsf{v}_2] \searrow \mathsf{a}}{\langle \mathsf{H}(\mathsf{f},\mathsf{v}_1,\mathsf{v}_2),\mathsf{v}\rangle_{\mathrm{F}},\mathsf{E} \searrow \mathsf{a}}$$

$$\frac{\langle \mathsf{v}_1,\mathsf{v}_2\rangle_{\mathrm{A}},\mathsf{E} \searrow \mathsf{a}}{\langle \mathsf{v}_1-,\mathsf{v}_2\rangle_{\mathrm{F}},\mathsf{E} \searrow \mathsf{a}}$$

$$\frac{\langle \mathsf{i},\mathsf{v}_h,\mathsf{v}_r\rangle_{\mathrm{H}},\mathsf{E} \searrow \mathsf{a}}{\langle \mathsf{i}\rangle_{\mathrm{K}},\mathsf{E}[\mathsf{handle}\,\mathsf{v}_h\,\mathsf{v}_r] \searrow \mathsf{a}} \qquad \frac{\langle \mathsf{i}\rangle_{\mathrm{T}},\mathsf{E}_{\mathsf{T}} \searrow \mathsf{a}}{\langle \mathsf{i}\rangle_{\mathrm{K}},\mathsf{E}_{\mathsf{T}} \searrow \mathsf{a}}$$

$$\frac{\langle \mathsf{v}_r,\mathsf{v}\rangle_{\mathrm{A}},\mathsf{E} \searrow \mathsf{a}}{\langle \mathsf{IRet}(\mathsf{v}),\mathsf{v}_h,\mathsf{v}_r\rangle_{\mathrm{H}},\mathsf{E} \searrow \mathsf{a}} \qquad \frac{\langle \mathsf{c}\rangle_{\mathrm{E}},\mathsf{E}[\mathsf{handle}\,\mathsf{v}_h\,\mathsf{H}(\mathsf{f},\mathsf{v}_h,\mathsf{v}_r)] \searrow \mathsf{a}}{\langle \mathsf{ILet}(\mathsf{c},\mathsf{f})\rangle_{\mathrm{H}},\mathsf{E}[\mathsf{handle}\,\mathsf{v}_h\,\mathsf{v}_r] \searrow \mathsf{a}}$$

$$\frac{\langle \mathsf{ILet}(\mathsf{v}_h\,\mathsf{v},-\mathsf{v}_r)\rangle_{\mathrm{K}},\mathsf{E} \searrow \mathsf{a}}{\langle \mathsf{IDo}(\mathsf{v})\rangle_{\mathrm{H}},\mathsf{E}[\mathsf{handle}\,\mathsf{v}_h\,\mathsf{v}_r] \searrow \mathsf{a}} \qquad \frac{\langle \mathsf{ILet}(\mathsf{c},\mathsf{v}_r-)\rangle_{\mathrm{K}},\mathsf{E} \searrow \mathsf{a}}{\langle \mathsf{ILift}(\mathsf{c})\rangle_{\mathrm{H}},\mathsf{E}[\mathsf{handle}\,\mathsf{v}_h\,\mathsf{v}_r] \searrow \mathsf{a}}$$

$$\frac{\langle \mathsf{v}\rangle_{\mathrm{VT}},\mathsf{E}_{\mathsf{T}} \searrow \mathsf{a}}{\langle \mathsf{IRet}(\mathsf{v})\rangle_{\mathrm{T}},\mathsf{E}_{\mathsf{T}} \searrow \mathsf{a}} \qquad \frac{\langle \mathsf{c}\rangle_{\mathrm{E}},\mathsf{E}_{\mathsf{T}}[\mathsf{let}\,\mathsf{f}] \searrow \mathsf{a}}{\langle \mathsf{ILet}(\mathsf{c},\mathsf{f})\rangle_{\mathrm{T}},\mathsf{E}_{\mathsf{T}} \searrow \mathsf{a}} \qquad \frac{}{\langle \mathsf{IDo}(\mathsf{v})\rangle_{\mathrm{T}},\mathsf{E}_{\mathsf{T}} \searrow \mathsf{E}_{\mathsf{T}}[\mathsf{do}\ \mathsf{v}]}$$

$$\frac{\langle \mathsf{c}\rangle_{\mathrm{E}},\mathsf{E}_{\mathsf{T}}[\mathsf{lift}] \searrow \mathsf{a}}{\langle \mathsf{ILift}(\mathsf{c})\rangle_{\mathrm{T}},\mathsf{E}_{\mathsf{T}} \searrow \mathsf{a}}$$

$$\frac{}{\langle \mathsf{v}\rangle_{\mathrm{VT}},[\,] \searrow \mathsf{v}} \qquad \frac{\langle \mathsf{f},\mathsf{v}\rangle_{\mathrm{F}},\mathsf{E} \searrow \mathsf{a}}{\langle \mathsf{v}\rangle_{\mathrm{VT}},\mathsf{E}[\mathsf{let}\,\mathsf{f}] \searrow \mathsf{a}} \qquad \frac{\langle \mathsf{v}\rangle_{\mathrm{VT}},\mathsf{E} \searrow \mathsf{a}}{\langle \mathsf{v}\rangle_{\mathrm{VT}},\mathsf{E}[\mathsf{lift}] \searrow \mathsf{a}}$$

Fig. 15. The evaluation judgment.

according to which one was missing. Finally, the H function reconstructs the larger context within which its first argument can now be applied to the given value.

This leaves us with the continuation configurations. The K-configuration is used to recognize if the given semantic intermediate value i has a handler on the stack or not. If it does, i is "handled" by the H-configuration according to the four transitions from h-configuration in the syntactic abstract machine, which are the analogues of (handle.v), (handle.do), (handle.let) and (handle.lift) reductions. If it does not, the T-configuration is

used to interpret i with a top-level stack $\mathsf{E}_\mathsf{T}$, again, completely in harmony with how transitions from t-configuration in the syntactic abstract machine work. In particular, at this point a do expression is reported as control-stuck and the answer is returned, while let and lift simply extend the stack and return to the eval configuration. Finally, the remaining configuration (VT) represents continuation of the evaluation when in the T-configuration a value has been encountered: this can either lead to the final result of the computation when the stack is empty, or the remaining two contractions: (let.v) or (lift.v).

Note that the relation is obviously functional in its two arguments, thus representing a graph of a partial function. Before turning to the read-back function and the process of full normalization, we remark that the evaluation judgment defined above corresponds with the weak-head reduction defined in Section 5.

Lemma 7.2. *If any configuration* $\mathsf{C}$ *with a stack* $\mathsf{E}$ *evaluates to an answer* $\mathsf{a}$*, i.e.,* $\mathsf{C}, \mathsf{E} \searrow \mathsf{a}$*, then its erasure reduces to the erasure of the answer:* $\lfloor \mathsf{E} \rfloor_\mathsf{E}[\lfloor \mathsf{C} \rfloor_\mathsf{C}] \twoheadrightarrow^\mathrm{w} \lfloor \mathsf{a} \rfloor_\mathsf{a}$.

Lemma 7.3. *For any two configurations* $\mathsf{C}_1, \mathsf{C}_2$ *such that* $\lfloor \mathsf{C}_1 \rfloor_\mathsf{C} \twoheadrightarrow^\mathrm{w} \lfloor \mathsf{C}_2 \rfloor_\mathsf{C}$ *and two stacks* $\mathsf{E}_1, \mathsf{E}_2$ *such that* $\lfloor \mathsf{E}_1 \rfloor_\mathsf{E} = \lfloor \mathsf{E}_2 \rfloor_\mathsf{E}$*, the two evaluations are equivalent as partial functions, i.e.,*

(1) *for any* $\mathsf{a}_1$*, if* $\mathsf{C}_1, \mathsf{E}_1 \searrow \mathsf{a}_1$*, then* $\mathsf{C}_2, \mathsf{E}_2 \searrow \mathsf{a}_2$ *for some* $\mathsf{a}_2$ *such that* $\lfloor \mathsf{a}_1 \rfloor_\mathsf{a} = \lfloor \mathsf{a}_2 \rfloor_\mathsf{a}$*;*
(2) *for any* $\mathsf{a}_2$*, if* $\mathsf{C}_2, \mathsf{E}_2 \searrow \mathsf{a}_2$*, then* $\mathsf{C}_1, \mathsf{E}_1 \searrow \mathsf{a}_1$ *for some* $\mathsf{a}_1$ *such that* $\lfloor \mathsf{a}_1 \rfloor_\mathsf{a} = \lfloor \mathsf{a}_2 \rfloor_\mathsf{a}$.

Since any expression that cannot progress in terms of weak-head reduction is an answer, the latter lemma leads to a simple corollary that any reduction to a weak-head normal form can be computed (up to erasure) by the evaluator ($x \mapsto x$ represents the initial identity environment):

Corollary 7.4. *For any expressions* $e_1, e_2$ *such that* $e_1 \twoheadrightarrow^\mathrm{w} e_2$ *and* $e_2 \not\twoheadrightarrow^\mathrm{w}$*, there exists an answer* $\mathsf{a}$ *such that* $\langle\langle e_1, x \mapsto x \rangle\rangle_\mathrm{E}, [\,] \searrow \mathsf{a}$ *and* $\lfloor \mathsf{a} \rfloor_\mathsf{a} = e_2$.

More important, from the normalization perspective, is the relationship of evaluation to general reduction. Instead of tackling this problem directly, we utilise the standardisation theorem, which helps us order the general reduction sequence. In order to state the lemmas, we first define the notions of reduction for the components of the evaluator by taking $\mathsf{v} \twoheadrightarrow_\mathrm{v} \mathsf{v}' \triangleq \lfloor \mathsf{v} \rfloor_\mathrm{v} \twoheadrightarrow^\mathrm{i} \lfloor \mathsf{v}' \rfloor_\mathrm{v}$ and $\mathsf{f} \twoheadrightarrow_\mathrm{f} \mathsf{f}' \triangleq \lfloor \mathsf{f} \rfloor^x_\mathrm{f} \twoheadrightarrow^\mathrm{s} \lfloor \mathsf{f}' \rfloor^x_\mathrm{f}$, and defining the notions for stacks ($\twoheadrightarrow_\mathrm{E}$) and answers ($\twoheadrightarrow_\mathrm{a}$) as a congruential closure over the constructors of the notions for values and functions. These definitions are straightforward as illustrated by the following two representative rules:

$$\frac{\mathsf{E} \twoheadrightarrow_\mathrm{E} \mathsf{E}' \qquad \mathsf{v} \twoheadrightarrow_\mathrm{v} \mathsf{v}'}{\mathsf{E}[x\,\mathsf{v}] \twoheadrightarrow_\mathrm{a} \mathsf{E}'[x\,\mathsf{v}']} \qquad\qquad \frac{\mathsf{E} \twoheadrightarrow_\mathrm{E} \mathsf{E}' \qquad \mathsf{f} \twoheadrightarrow_\mathrm{f} \mathsf{f}'}{\mathsf{E}[\mathsf{let}\,\mathsf{f}] \twoheadrightarrow_\mathrm{E} \mathsf{E}'[\mathsf{let}\,\mathsf{f}']}$$

This lets us state the properties that relate reduction and evaluation.

Lemma 7.5. *For any configurations* $\mathsf{C}_1, \mathsf{C}_2$ *and stacks* $\mathsf{E}_1, \mathsf{E}_2$ *such that* $\lfloor \mathsf{E}_1 \rfloor_\mathsf{E}[\lfloor \mathsf{C}_1 \rfloor_\mathsf{C}] \twoheadrightarrow^\mathrm{s} \lfloor \mathsf{E}_2 \rfloor_\mathsf{E}[\lfloor \mathsf{C}_2 \rfloor_\mathsf{C}]$*:*

(1) *if* $\mathsf{C}_1, \mathsf{E}_1 \searrow \mathsf{a}_1$ *for some* $\mathsf{a}_1$*, there exists* $\mathsf{a}_2$ *such that* $\mathsf{C}_2, \mathsf{E}_2 \searrow \mathsf{a}_2$ *and* $\mathsf{a}_1 \twoheadrightarrow_\mathrm{a} \mathsf{a}_2$*;*
(2) *if* $\mathsf{C}_2, \mathsf{E}_2 \searrow \mathsf{a}_2$ *for some* $\mathsf{a}_2$*, there exists* $\mathsf{a}_1$ *such that* $\mathsf{C}_1, \mathsf{E}_1 \searrow \mathsf{a}_1$ *and* $\mathsf{a}_1 \twoheadrightarrow_\mathrm{a} \mathsf{a}_2$.

**Read-back.** With the evaluation aspect defined, we now turn to the read-back function, whose graph is defined inductively in Figure 16. It consists of four mutually inductive predicates: reading back answers as normal forms ($\mathsf{RB}_\mathsf{a}$), semantic values as normal values

$$\frac{\mathsf{RB_v}\ \mathsf{v} \searrow \xi}{\mathsf{RB_a}\ \mathsf{v} \searrow \xi} \qquad \frac{\mathsf{RB_v}\ \mathsf{v} \searrow \xi \qquad \mathsf{RB_E}\ \mathsf{E}[x\,\xi] \searrow \nu}{\mathsf{RB_a}\ \mathsf{E}[x\,\mathsf{v}] \searrow \nu}$$

$$\frac{\mathsf{RB_v}\ \mathsf{v} \searrow \xi \qquad \mathsf{RB_E}\ \mathsf{E_T}[\mathsf{do}\ \xi] \searrow \nu}{\mathsf{RB_a}\ \mathsf{E_T}[\mathsf{do}\ \mathsf{v}] \searrow \nu}$$

$$\frac{}{\mathsf{RB_v}\ x \searrow x} \qquad \frac{\mathsf{RB_f}\ \mathsf{f}^x \searrow \mu}{\mathsf{RB_v}\ \mathsf{f} \searrow \lambda x.\mu} \qquad \frac{\langle \mathsf{f}, x\rangle_{\mathsf{F}}, [\,] \searrow \mathsf{a} \qquad \mathsf{RB_a}\ \mathsf{a} \searrow \mu}{\mathsf{RB_f}\ \mathsf{f}^x \searrow \mu}$$

$$\frac{}{\mathsf{RB_E}\ [\,][\nu] \searrow \nu} \qquad \frac{\mathsf{RB_f}\ \mathsf{f}^x \searrow \mu \qquad \mathsf{RB_E}\ \mathsf{E_T}[\mathsf{let}\, x = \nu\, \mathsf{in}\, \mu] \searrow \nu'}{\mathsf{RB_E}\ \mathsf{E_T}[\mathsf{let}\,\mathsf{f}][\nu] \searrow \nu'}$$

$$\frac{\mathsf{RB_E}\ \mathsf{E_T}[\mathsf{lift}\ \nu] \searrow \nu'}{\mathsf{RB_E}\ \mathsf{E_T}[\mathsf{lift}][\nu] \searrow \nu'}$$

$$\frac{\mathsf{RB_v}\ \mathsf{v} \searrow \xi \qquad \mathsf{RB_v}\ \mathsf{v}' \searrow \xi' \qquad \mathsf{RB_E}\ \mathsf{E}[\mathsf{handle}\ \nu_{\mathsf{H}}\ \mathsf{with}\ \xi, \xi'] \searrow \nu'}{\mathsf{RB_E}\ \mathsf{E}[\mathsf{handle}\ \mathsf{v}\ \mathsf{v}'][\nu_{\mathsf{H}}] \searrow \nu'}$$

Fig. 16. The read-back judgment.

($\mathsf{RB_v}$), semantic functions (with a variable name as a second parameter) as normal forms $\mathsf{RB_f}$, and stacks, together with neutral expressions ($\mathsf{RB_E}$). Most of the process proceeds structurally, with all the recursive evaluation in the internal structure delegated to reading back of functions—which, as expected, proceeds by applying the function to the fresh variable (chosen to match the appropriate binder in the resulting normal form). The definition of $\mathsf{RB_E}\ \mathsf{E}[\nu] \searrow \nu'$ is an enriched plug operation that reifies context $\mathsf{E}$ frame by frame and plugs the result with a neutral expression $\nu$ yielding another neutral expression $\nu'$. The invariant here is that $\mathsf{E}$, $\nu$, and $\nu'$ are of the same kind, which we elide to avoid clutter.

As expected, the following properties are rather simple to obtain, given the lemmas about evaluation above.

Lemma 7.6. *The read-back process only performs internal reductions, i.e., for any answer* $\mathsf{a}$ *and normal form* $\mu$*, if* $\mathsf{RB_a}\ \mathsf{a} \searrow \mu$*, then* $\lfloor a \rfloor \twoheadrightarrow^{\mathsf{i}} \mu$.

Lemma 7.7. *The read-back process is closed under reductions of answers, i.e., for any answers* $\mathsf{a}_1, \mathsf{a}_2$ *such that* $\mathsf{a}_1 \twoheadrightarrow_{\mathsf{a}} \mathsf{a}_2$

(1) *if* $\mathsf{RB_a}\ \mathsf{a}_1 \searrow \mu$ *for some* $\mu$*, then* $\mathsf{RB_a}\ \mathsf{a}_2 \searrow \mu$*;*
(2) *if* $\mathsf{RB_a}\ \mathsf{a}_2 \searrow \mu$ *for some* $\mu$*, then* $\mathsf{RB_a}\ \mathsf{a}_1 \searrow \mu$.

The proofs proceed by induction on the structure of the derivation of read-back, mutually with analogous lemmas for values, functions and contexts, and utilise appropriate lemmas about evaluation in the case for functions.

Together with the standardisation theorem and the lemmas about behaviour of evaluation with respect to standard reduction, we obtain soundness and completeness of the normalization process.

Theorem 7.8 (Soundness and Completeness of NbE). *The (partial) normalization function, defined as*

$$\mathrm{norm}(e) \searrow \mu \iff \exists \mathsf{a}.\ \langle\langle e, x \mapsto x\rangle\rangle_{\mathrm{E}}, [\,] \searrow \mathsf{a} \wedge \mathsf{RB}_{\mathsf{a}}\ a \searrow \mu$$

*is sound and complete, i.e.,*

(1) *for any* $e$, $\mu$, *if* $\mathrm{norm}(e) \searrow \mu$ *then* $e \longrightarrow^{*}_{\lambda_{\mathsf{fg}}} \mu$;
(2) *for any* $e_1, e_2$ *if* $e_1 =_{\lambda_{\mathsf{fg}}} e_2$ *and* $\mathrm{norm}(e_1) \searrow \mu$ *for some* $\mu$, *then* $\mathrm{norm}(e_2) \searrow \mu$.

Since $\mathrm{norm}(\mu) \searrow \mu$, for any normal form $\mu$, Theorem 7.8 implies that if $e \longrightarrow^{*}_{\lambda_{\mathsf{fg}}} \mu$ or, more generally, $e =_{\lambda_{\mathsf{fg}}} \mu$, then the normalization procedure will compute the normal form of $e$, i.e., $\mathrm{norm}(e) \searrow \mu$.

The partial normalizer that we obtained matches our expectations outlined in the introduction. Indeed, if we extended our language (and the normalizer) with lists, the result of normalization of the example from the introduction is the sequence of operations that we expected to obtain.

## 8 Related work

### 8.1 Reduction semantics for effect handlers

Operational semantics of effect handlers is typically presented as a small-step reduction relation that either takes advantage of explicitly represented evaluation contexts and captures resumptions at once, or it builds the resumption piece-by-piece with a helper construct that holds a prefix of the context to be captured in search of the matching handler. In each case, the resumption is built inside-out.

The former, more global approach can be found, e.g., in Leijen's work on Koka (Leijen 2017b), in Hillerström et al.'s work on effect handlers in the context of the Links programming language (Hillerström et al. 2020), and in a series of papers coauthored by the first and third authors revolving around the Helium programming language (Biernacki et al. 2018, 2019; Piróg et al. 2019). Such semantics is an adaptation of a classical context-sensitive reduction semantics for delimited control introduced by Felleisen et al. (Biernacka et al. 2005a) and directly implementable on a CEK-like abstract machine (Biernacka et al. 2005b; Dybvig et al. 2007; Felleisen and Friedman 1987).

The latter, more local appoach can be found, e.g., in Bauer and Pretnar's calculi modelling the Eff programming language (Bauer and Pretnar 2015; Kammar et al. 2013; Pretnar 2015) or in Schuster et al.'s foundations of the Effekt programming language, based on the notion of capability (Schuster et al. 2022), where the continuation is captured by unwinding the stack of a CEK-like abstract machine. Again, such semantics is an adaptation of a classical operational, based on local reduction rules, semantics for calculi with control operators (Felleisen et al. 1986), first applied to delimited control by Felleisen (1988).

In principle, it is relatively easy to see the correspondence between the global and local semantics of a calculus and to define one given the other one. In both cases performing the effect operation initiates the resumption capture. The fine-grained operational semantics of Section 2 is based on an entirely different principle. Here, the handler plays the leading role and the semantics of the program hinges on the reduction rules that describe the interaction of the handler with other constructs. As a consequence, the resumption is built outside-in. Altogether, this new design makes it non-trivial to establish the correctness of the semantics but, at the same time, opens new possibilities for applications. Moreover, it does not require

the runtime system to search or unwind the stack or to keep track of the freeness level for contexts, otherwise critical in the presence of lift (Biernacki et al. 2018, 2019; Piróg et al. 2019). This form of operational semantics is already present in the authors' work on the control operator shift0 (Biernacki et al. 2021).

### *8.2 Rewriting rules for handlers as code optimisation*

Pretnar (2015) presents a set of basic equivalences for labeled handlers that are sound with respect to observational equivalence in their calculus. In his PhD dissertation (Saleh 2019), Saleh goes further and defines a set of optimisation rules for handlers that is divided into the simplification rules and handler-reduction rules (earlier works in this line include (Saleh and Schrijvers 2016) and (Pretnar et al. 2017)). The former serve as administrative rules to expose handlers, whereas the latter aim at eliminating or distributing handlers. Three of them coincide with our rules (handle.v), (handle.do), and (handle.let), one is dedicated to a treatment of recursive functions, which we do not need to handle explicitly, and the last one deals with pure computations. The latter rule takes advantage of a type-and-effect system – we instead work in the untyped setting. These optimisation rules were then used by Karachalias et al. (2021) in their study of efficient compilation of effect handlers, again, heavily relying on a type-and-effect system (with subtyping).

None of the above works treats the optimisation rules as a reduction theory proper. In particular, no properties such as confluence or standardisation have been established before for the reduction rules that are the topic of the present work. Moreover, Saleh and Karachalias et al. only prove soundness of their reduction rules with respect to contextual equivalence, whereas we establish both their soundness and completeness with respect to the standard operational semantics.

### *8.3 Confluence and standardisation proofs*

Our proofs of confluence and standardisation theorems follow the most recent Agda formalization of these results for the pure $\lambda$-calculus by McKinna, presented in his unpublished lecture notes (McKinna 2022). The confluence proof is based on the technique of super-parallel reductions and complete super developments (Aczel 1978; McKinna 2022; McKinna and Pollack 1999; Takahashi 1995; van Raamsdonk 1993), which is a simplification of the classic proof method by parallel reductions due to Tait and Martin-Löf (Barendregt 1985). The proof of the standardisation theorem (Mitschke 1979), in turn, is based on the methodology introduced by Takahashi (1995) and refined by McKinna and Pollack (McKinna 2022; McKinna and Pollack 1999). McLaughlin's proof of the Context Lemma for Ella (McLaughlin 2020, Chapter 6), a core calculus of the Frank programming language (Convent et al. 2020), takes advantage of some of the techniques characteristic to such standardisation proofs.

### *8.4 Normalization by evaluation for algebraic effects*

The only result that appears to directly take up the subject of a fully normalizing procedure for a calculus with algebraic effects is due to Ahman and Staton (2013). The normalization-by-evaluation algorithm presented in that work is rather distant from ours: it relies on a (simple) type system and the grammar of terms does not include effect handlers, just algebraic operations to be interpreted at the meta level, in a residualizing monad.

Normalization by evaluation for the untyped lambda calculus has been studied in (Abel 2013; Aehlig and Joachimski 2004; Filinski and Rohde 2005). We followed Abel's work that relies on partial applicative structures and big-step operational semantics (Harper et al. 1993; Kahn 1987). These normalizers are in the functional correspondence (Ager et al. 2003) with fully normalizing abstract machines (Biernacka et al. 2020, 2022; García-Pérez and Nogueira 2014). A read-back function was first presented as a key ingredient of a compiled fully normalizing abstract machine in (Grégoire and Leroy 2002).

## 9 Conclusion and future work

We have introduced an operational semantics for deep effect handlers based on a notion of fine-grained reduction that describes the interaction of control delimiters (handlers) with other computation formers. Based on this notion of contraction, we have developed a reduction theory, with classic properties such as confluence and standardisation—some of the earliest such results for reduction theories of effect handlers and, to the best of our knowledge, the first explicitly stated. Due to the complexities of the reduction rules and the hybrid nature of the evaluation, certain technical innovations were introduced to the classic techniques used in these proofs. We believe that they may generalise to other theories that exhibit similar complexities.

To connect our semantics to the traditional, context-capturing notion of evaluation we have established a bisimilarity between the two; thus, since our reductions are more local and finer-grained, we provide a more powerful equational theory that is nonetheless compatible with the traditional evaluators. As a further application of our semantics to partial evaluation, we have developed the first formalized normalization-by-evaluation procedure for an effect handler calculus, which can be used in program specialisation, or to establish convertibility of expressions involving handlers. These developments fill a significant gap in theoretical foundations of programming with effect handlers.

**Formalization.** As noted in the introduction, the bulk of the results presented above has been formalized using the Rocq interactive theorem prover. In order to aid clarity, we have used explicit variable names with a hygienic convention in play throughout the paper, where the formalization uses a functorial, intrinsically well-scoped representation (Sieczkowski and Polesiuk 2024); this, however, is clearly an adequate choice. Except for this point, the presentation follows the formalization closely, although there are cosmetic differences in the presentation of the evaluator in Section 7. We believe that our formalization effort presents additional evidence for scalability of techniques such as super-parallel reductions (in Section 4) or parallel internal reductions (in Section 5) to complex reduction theories: while we had to extend those in certain ways to account for the particular critical pairs or the fact that our weak-head reduction is a hybrid strategy, the fact that they *could* be naturally extended speaks to their robustness.

**Future work.** While significant theoretical properties of our calculus have been established, there are still major directions for future work. Firstly, the impact of the fine-grained semantics on the connection between effect handlers and pure lambda calculus, through a transformation to continuation-passing style, should be explored. Such a development would help ensure that the reduction rules are not just internally coherent, but tie in appropriately with a compilation process to a pure calculus. It might also shed light on whether a variant of the semantics where

lift in evaluation position is always control-stuck is more natural than the current formulation. A separate area for investigation is the interplay between the fine-grained rules and shallow handlers: it is clear that in the current presentation the rules are not easily applicable to shallow handlers (Hillerström and Lindley 2018).[9] If adaptation of the fine-grained approach to shallow handlers proves impossible, characterising and explaining such a failure could provide a valuable insight into the nature of the two varieties of control operators. Finally, our normalization algorithm could be used—in a statically-typed, terminating context—as a basis for convertibility checking, which could help guide integration of effect handlers (with type-and-effect-systems) and dependent types.

## Acknowledgments

We thank the anonymous ICFP 2023 and JFP reviewers for their valuable comments on the presentation of our results and related work. We are also grateful to James McKinna for insightful discussions, particularly regarding the standardization theorem, and to Witold Charatonik for his helpful remarks on the exposition. This work was partially supported by the National Science Centre, Poland, under grant number 2019/33/B/ST6/00289.

## References

Andreas Abel. 2013. Normalization by Evaluation: Dependent Types and Impredicativity. Habilitation thesis.

Beniamino Accattoli and Giulio Guerrieri. 2016. Open Call-by-Value. In *Programming Languages and Systems - 14th Asian Symposium, APLAS 2016, Hanoi, Vietnam, November 21-23, 2016, Proceedings (Lecture Notes in Computer Science, Vol. 10017)*, Atsushi Igarashi (Ed.). 206–226. https://doi.org/10.1007/978-3-319-47958-3_12

Peter Aczel. 1978. *A General Church-Rosser Theorem*. Technical Report. University of Manchester.

Klaus Aehlig and Felix Joachimski. 2004. Operational aspects of untyped Normalisation by Evaluation. *Math. Struct. Comput. Sci.* 14, 4 (2004), 587–611. https://doi.org/10.1017/S096012950400427X

Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. 2003. A functional correspondence between evaluators and abstract machines. In *Proceedings of the 5th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, 27-29 August 2003, Uppsala, Sweden*. ACM, 8–19. https://doi.org/10.1145/888251.888254

Danel Ahman and Sam Staton. 2013. Normalization by Evaluation and Algebraic Effects. In *Proceedings of the Twenty-ninth Conference on the Mathematical Foundations of Programming Semantics, MFPS 2013, New Orleans, LA, USA, June 23-25, 2013 (Electronic Notes in Theoretical Computer Science, Vol. 298)*, Dexter Kozen and Michael W. Mislove (Eds.). Elsevier, 51–69. https://doi.org/10.1016/j.entcs.2013.09.007

Hendrik Pieter Barendregt. 1985. *The lambda calculus - its syntax and semantics*. Studies in logic and the foundations of mathematics, Vol. 103. North-Holland.

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

Malgorzata Biernacka, Dariusz Biernacki, Witold Charatonik, and Tomasz Drab. 2020. An Abstract Machine for Strong Call by Value. In *Programming Languages and Systems - 18th Asian Symposium, APLAS 2020, Fukuoka, Japan, November 30 - December 2, 2020, Proceedings (Lecture Notes in Computer Science, Vol. 12470)*, Bruno C. d. S. Oliveira (Ed.). Springer, 147–166. https://doi.org/10.1007/978-3-030-64437-6_8

Malgorzata Biernacka, Dariusz Biernacki, and Olivier Danvy. 2005a. An Operational Foundation for Delimited Continuations in the CPS Hierarchy. *Log. Methods Comput. Sci.* 1, 2 (2005). https://doi.org/10.2168/LMCS-1(2:5)2005

Malgorzata Biernacka, Dariusz Biernacki, and Olivier Danvy. 2005b. An Operational Foundation for Delimited Continuations in the CPS Hierarchy. *CoRR* abs/cs/0508048 (2005). arXiv:cs/0508048 http://arxiv.org/abs/cs/0508048

[9]The same applies to delimited control operators, where shift0/reset admit fine-grained rules, while the status is unclear for control0/prompt.

Malgorzata Biernacka, Witold Charatonik, and Tomasz Drab. 2022. A simple and efficient implementation of strong call by need by an abstract machine. *Proc. ACM Program. Lang.* 6, ICFP (2022), 109–136. https://doi.org/10.1145/3549822

Malgorzata Biernacka, Witold Charatonik, and Klara Zielinska. 2017. Generalized Refocusing: From Hybrid Strategies to Abstract Machines. In *2nd International Conference on Formal Structures for Computation and Deduction, FSCD 2017, September 3-9, 2017, Oxford, UK (LIPIcs, Vol. 84)*, Dale Miller (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 10:1–10:17. https://doi.org/10.4230/LIPIcs.FSCD.2017.10

Dariusz Biernacki, Sergueï Lenglet, and Piotr Polesiuk. 2020. A Complete Normal-Form Bisimilarity for Algebraic Effects and Handlers. In *5th International Conference on Formal Structures for Computation and Deduction, FSCD 2020, June 29-July 6, 2020, Paris, France (Virtual Conference) (LIPIcs, Vol. 167)*, Zena M. Ariola (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 7:1–7:22. https://doi.org/10.4230/LIPIcs.FSCD.2020.7

Dariusz Biernacki, James McKinna, and Filip Sieczkowski. 2025. Call-by-value and call-by-name: A simple proof of a classic theorem. *J. Funct. Program.* 35 (2025). https://doi.org/10.1017/S0956796825100038

Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. 2018. Handle with care: Relational interpretation of algebraic effects and handlers. *PACMPL* 2, POPL (2018), 8:1–8:30. https://doi.org/10.1145/3158096

Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. 2019. Abstracting algebraic effects. *Proc. ACM Program. Lang.* 3, POPL (2019), 6:1–6:28. https://doi.org/10.1145/3290319

Dariusz Biernacki, Mateusz Pyzik, and Filip Sieczkowski. 2021. Reflecting Stacked Continuations in a Fine-Grained Direct-Style Reduction Theory. In *PPDP 2021: 23rd International Symposium on Principles and Practice of Declarative Programming, Tallinn, Estonia, September 6-8, 2021*, Niccolò Veltri, Nick Benton, and Silvia Ghilezan (Eds.). ACM, 4:1–4:13. https://doi.org/10.1145/3479394.3479399

Richard S. Bird and Ross Paterson. 1999. De Bruijn Notation as a Nested Datatype. *J. Funct. Program.* 9, 1 (1999), 77–91. https://doi.org/10.1017/S0956796899003366

Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. 2020. Effects as Capabilities: Effect Handlers and Lightweight Effect Polymorphism. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 126 (nov 2020), 30 pages. https://doi.org/10.1145/3428194

Lukas Convent, Sam Lindley, Conor McBride, and Craig McLaughlin. 2020. Doo bee doo bee doo. *J. Funct. Program.* 30 (2020), e9. https://doi.org/10.1017/S0956796820000039

Olivier Danvy and Andrzej Filinski. 1990. Abstracting Control. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming, LFP 1990, Nice, France, 27-29 June 1990*, Gilles Kahn (Ed.). ACM, 151–160. https://doi.org/10.1145/91556.91622

Olivier Danvy and Lasse R. Nielsen. 2004. Refocusing in Reduction Semantics. *BRICS Report Series* 11, 26 (Nov. 2004). https://doi.org/10.7146/brics.v11i26.21851

R. Kent Dybvig, Simon L. Peyton Jones, and Amr Sabry. 2007. A monadic framework for delimited continuations. *J. Funct. Program.* 17, 6 (2007), 687–730. https://doi.org/10.1017/S0956796807006259

Matthias Felleisen. 1988. The Theory and Practice of First-Class Prompts. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, San Diego, California, USA, January 10-13, 1988*, Jeanne Ferrante and Peter Mager (Eds.). ACM Press, 180–190. https://doi.org/10.1145/73560.73576

Matthias Felleisen. 1991. On the Expressive Power of Programming Languages. *Sci. Comput. Program.* 17, 1-3 (1991), 35–75. https://doi.org/10.1016/0167-6423(91)90036-W

Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. 2009. *Semantics Engineering with PLT Redex*. MIT Press. http://mitpress.mit.edu/catalog/item/default.asp?ttype=2&tid=11885

Matthias Felleisen and Daniel P. Friedman. 1987. Control operators, the SECD-machine, and the $\lambda$-calculus. In *Formal Description of Programming Concepts - III: Proceedings of the IFIP TC 2/WG 2.2 Working Conference on Formal Description of Programming Concepts - III, Ebberup, Denmark, 25-28 August 1986*, Martin Wirsing (Ed.). North-Holland, 193–222.

Matthias Felleisen, Daniel P. Friedman, Eugene E. Kohlbecker, and Bruce F. Duba. 1986. Reasoning with Continuations. In *Proceedings of the Symposium on Logic in Computer Science (LICS '86), Cambridge, Massachusetts, USA, June 16-18, 1986*. IEEE Computer Society, 131–141.

Andrzej Filinski and Henning Korsholm Rohde. 2005. Denotational aspects of untyped normalization by evaluation. *RAIRO Theor. Informatics Appl.* 39, 3 (2005), 423–453. https://doi.org/10.1051/ita:2005026

Álvaro García-Pérez and Pablo Nogueira. 2014. On the syntactic and functional correspondence between hybrid (or layered) normalisers and abstract machines. *Sci. Comput. Program.* 95 (2014), 176–199. https:

//doi.org/10.1016/j.scico.2014.05.011

Benjamin Grégoire and Xavier Leroy. 2002. A compiled implementation of strong reduction. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP '02), Pittsburgh, Pennsylvania, USA, October 4-6, 2002*, Mitchell Wand and Simon L. Peyton Jones (Eds.). ACM, 235–246. https://doi.org/10.1145/581478.581501

Robert Harper, Bruce F. Duba, and David B. MacQueen. 1993. Typing First-Class Continuations in ML. *J. Funct. Program.* 3, 4 (1993), 465–484. https://doi.org/10.1017/S095679680000085X

Daniel Hillerström and Sam Lindley. 2018. Shallow Effect Handlers. In *Programming Languages and Systems - 16th Asian Symposium, APLAS 2018, Wellington, New Zealand, December 2-6, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 11275)*, Sukyoung Ryu (Ed.). Springer, 415–435. https://doi.org/10.1007/978-3-030-02768-1_22

Daniel Hillerström, Sam Lindley, and Robert Atkey. 2020. Effect handlers via generalised continuations. *J. Funct. Program.* 30 (2020), e5. https://doi.org/10.1017/S0956796820000040

Gilles Kahn. 1987. Natural Semantics. In *STACS 87, 4th Annual Symposium on Theoretical Aspects of Computer Science, Passau, Germany, February 19-21, 1987, Proceedings (Lecture Notes in Computer Science, Vol. 247)*, Franz-Josef Brandenburg, Guy Vidal-Naquet, and Martin Wirsing (Eds.). Springer, 22–39. https://doi.org/10.1007/BFb0039592

Ohad Kammar, Sam Lindley, and Nicolas Oury. 2013. Handlers in action. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, Greg Morrisett and Tarmo Uustalu (Eds.). ACM, 145–158. https://doi.org/10.1145/2500365.2500590

Georgios Karachalias, Filip Koprivec, Matija Pretnar, and Tom Schrijvers. 2021. Efficient compilation of algebraic effect handlers. *Proc. ACM Program. Lang.* 5, OOPSLA (2021), 1–28. https://doi.org/10.1145/3485479

Søren B. Lassen. 2005. Eager Normal Form Bisimulation. In *20th IEEE Symposium on Logic in Computer Science (LICS 2005), 26-29 June 2005, Chicago, IL, USA, Proceedings*. IEEE Computer Society, 345–354. https://doi.org/10.1109/LICS.2005.15

Daan Leijen. 2017a. Type directed compilation of row-typed algebraic effects. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 486–499. https://doi.org/10.1145/3009837.3009872

Daan Leijen. 2017b. Type directed compilation of row-typed algebraic effects. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 486–499. https://doi.org/10.1145/3009837

Paul Blain Levy, John Power, and Hayo Thielecke. 2003. Modelling environments in call-by-value programming languages. *Inf. Comput.* 185, 2 (2003), 182–210. https://doi.org/10.1016/S0890-5401(03)00088-9

Marek Materzok and Dariusz Biernacki. 2011. Subtyping delimited continuations. In *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011*, Manuel M. T. Chakravarty, Zhenjiang Hu, and Olivier Danvy (Eds.). ACM, 81–93. https://doi.org/10.1145/2034773.2034786

James McKinna. 2022. The Lambda Calculus, Formalised: the Church-Rosser and Standardisation Theorems, with Applications. https://www.macs.hw.ac.uk/splv/splv-2022/ Unpublished Lecture Notes from the Scottish Programming Languages and Verification Summer School (SPLV 2022), Edinburgh.

James McKinna and Robert Pollack. 1999. Some Lambda Calculus and Type Theory Formalized. *J. Autom. Reason.* 23, 3-4 (1999), 373–409. https://doi.org/10.1023/A:1006294005493

Craig McLaughlin. 2020. *Relational reasoning for effects and handlers*. Ph. D. Dissertation. University of Edinburgh, UK. https://doi.org/10.7488/era/537

Gerd Mitschke. 1979. The Standardization Theorem for $\lambda$-Calculus. *Math. Log. Q.* 25, 1-2 (1979), 29–31. https://doi.org/10.1002/malq.19790250104

Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. 2019. Typed Equivalence of Effect Handlers and Delimited Control. In *4th International Conference on Formal Structures for Computation and Deduction, FSCD 2019, June 24-30, 2019, Dortmund, Germany (LIPIcs, Vol. 131)*, Herman Geuvers (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 30:1–30:16. https://doi.org/10.4230/LIPIcs.FSCD.2019.30

Gordon D. Plotkin and A. John Power. 2004. Computational Effects and Operations: An Overview. *Electronic Notes in Theoretical Computer Science* 73 (2004), 149–163. https://doi.org/10.1016/j.entcs.2004.08.008

Gordon D. Plotkin and Matija Pretnar. 2013. Handling Algebraic Effects. *Logical Methods in Computer Science* 9, 4:23 (2013), 1–36. https://doi.org/10.2168/LMCS-9(4:23)2013

Matija Pretnar. 2015. An Introduction to Algebraic Effects and Handlers. Invited tutorial paper. In *The 31st Conference on the Mathematical Foundations of Programming Semantics, MFPS 2015, Nijmegen, The Netherlands, June 22-25, 2015 (Electronic Notes in Theoretical Computer Science, Vol. 319)*, Dan R. Ghica (Ed.). Elsevier, 19–35. https://doi.org/10.1016/j.entcs.2015.12.003

Matija Pretnar, Amr Hany Saleh, Axel Faes, and Tom Schrijvers. 2017. *Efficient compilation of algebraic effects and handlers*. Technical Report CW708. KU Leuven, Department of Computer Science.

Amr Hany Saleh. 2019. *Efficient Algebraic Effect Handlers*. Ph. D. Dissertation. Arenberg Doctoral School, Faculty of Engineering Science, KU Leuven, Belgium.

Amr Hany Saleh and Tom Schrijvers. 2016. Efficient algebraic effect handlers for Prolog. *Theory Pract. Log. Program.* 16, 5-6 (2016), 884–898. https://doi.org/10.1017/S147106841600034X

Philipp Schuster, Jonathan Immanuel Brachthäuser, Marius Müller, and Klaus Ostermann. 2022. A typed continuation-passing translation for lexical effect handlers. In *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*, Ranjit Jhala and Isil Dillig (Eds.). ACM, 566–579. https://doi.org/10.1145/3519939.3523710

Filip Sieczkowski and Piotr Polesiuk. 2024. Functorial Syntax for All. *The Tenth International Workshop on Coq for Programming Languages (CoqPL 2024), London, United Kingdom, 2024*.

Filip Sieczkowski, Mateusz Pyzik, and Dariusz Biernacki. 2023. A General Fine-Grained Reduction Theory for Effect Handlers. *Proc. ACM Program. Lang.* 7, ICFP (2023), 511–540. https://doi.org/10.1145/3607848

K. C. Sivaramakrishnan, Stephen Dolan, Leo White, Tom Kelly, Sadiq Jaffer, and Anil Madhavapeddy. 2021. Retrofitting effect handlers onto OCaml. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 206–221. https://doi.org/10.1145/3453483.3454039

Masako Takahashi. 1995. Parallel Reductions in lambda-Calculus. *Inf. Comput.* 118, 1 (1995), 120–127. https://doi.org/10.1006/inco.1995.1057

Femke van Raamsdonk. 1993. Confluence and Superdevelopments. In *Rewriting Techniques and Applications, 5th International Conference, RTA-93, Montreal, Canada, June 16-18, 1993, Proceedings (Lecture Notes in Computer Science, Vol. 690)*, Claude Kirchner (Ed.). Springer, 168–182. https://doi.org/10.1007/978-3-662-21551-7_14

Yizhou Zhang and Andrew C. Myers. 2019. Abstraction-Safe Effect Handlers via Tunneling. *Proc. ACM Program. Lang.* 3, POPL, Article 5 (jan 2019), 29 pages. https://doi.org/10.1145/3290318

```
type 'a value =
  | Var of 'a
  | Lam of ('a option) expr
and 'a expr  =
  | Val of 'a value
  | App of 'a value * 'a value
  | Let of 'a expr * ('a option) expr
  | Lift of 'a expr
  | Do of 'a value
  | Hndl of 'a expr * 'a value * 'a value

type top = |
type hnd = |

type 'a nf =
  | NV of 'a nv
  | NC of ('a, top) nc
and 'a nv =
  | NVar of 'a
  | NLam of 'a option nf
and ('a, _) nc =
  | NDo : 'a nv -> ('a, top) nc
  | NApp : 'a * 'a nv -> ('a, 'b) nc
  | NLift : ('a, top) nc -> ('a, top) nc
  | NLet  : ('a, top) nc * 'a option nf -> ('a, top) nc
  | NHndl : ('a, hnd) nc * 'a nv * 'a nv -> ('a, 'b) nc
```

Fig. 17. The syntax of the calculus and the normal forms as OCaml datatypes.

## A Implementations of the normalizer

In this section, we discuss the implementation of the normalization algorithm of Section 7 as OCaml functions, rather than the inductively defined relation.

The syntax of the calculus, presented in Figure 17, is defined in a functorial style, with the type argument that determines the type of free variables (Bird and Paterson 1999). This allows us to ensure that terms of the calculus are well-scoped (and any operations on them preserve scope appropriately), and is in line with the Rocq formalization. However, in contrast to substitution-based approaches, we do not need to provide the syntactic renaming and substitution operations on the programs: the programs are static inputs to the evaluator. We also provide the syntax of the normal forms, following closely the presentation in Figure 11. In particular, we keep track of whether the handler is present: the empty type `top` corresponds to the index T, while another empty type, `hnd`, corresponds to the index H (and thus, to the normal form occurring under a handle construct).

```
type 'a sval =
  | SVar of 'a
  | SFun of 'a sfun
and 'a sfun =
  | SCl : 'b option expr * ('b, 'a) env -> 'a sfun
  | SAppL of 'a sval
  | SAppR of 'a sval
  | SHRet of 'a sfun * 'a sval * 'a sval
and ('a, 'b) env = 'a -> 'b sval

type 'a sclo =
  | CPair : 'a expr * ('a, 'b) env -> 'b sclo
  | CApp of 'a sval * 'a sval

type 'a sint =
  | IVal  of 'a sval
  | IDo   of 'a sval
  | ILet  of 'a sclo * 'a sfun
  | ILift of 'a sclo

type ('a, _) stack =
  | EEmp : ('a, top) stack
  | ELet : ('a, top) stack * ('a sfun) -> ('a, top) stack
  | ELift : ('a, top) stack -> ('a, top) stack
  | EHndl : ('a, 'b) stack * 'a sval * 'a sval -> ('a, hnd) stack

type 'a ans =
  | AVal of 'a sval
  | ADo  of 'a sval * ('a, top) stack
  | AApp : 'a * 'a sval * ('a, 'b) stack -> 'a ans
```

Fig. 18. Evaluator datatypes for the machine-style presentation.

### *A.1 Machine-style implementation*

We begin by describing an implementation with a first-order[10] representation of semantic functions as closures, following closely the machine presented in Section 7.2. The definitions of semantic constructs are presented in Figure 18, and follow closely the ones of Figure 12. The `sval` and `sfun` types are equipped with natural renaming functions, which allow them to be "shifted" to introduce a fresh variable. While not necessary in the evaluator itself, these are needed for the read-back functions, when going under binders. The data type `sint` represents the intermediate values. Note also that, like the normal forms, the stacks are indexed with `top` or `hnd`.

[10]Or near-enough: the environments are represented as functions due to the choice of well-scoped representation of expressions.

```
let rec eval : type a b s. a expr -> (a, b) env -> (b, s) stack -> b ans =
  fun e r s -> match e with
  | Val v          -> cont s (IVal (mkV v r))
  | Let (a, e)     -> cont s (ILet (CPair (a, r), SCl (e, r)))
  | Lift e         -> cont s (ILift (CPair (e, r)))
  | Do  v          -> cont s (IDo (mkV v r))
  | App (u, v)     -> apply (mkV u r) (mkV v r) s
  | Hndl (e, h, c) -> eval e r (EHndl (s, mkV h r, mkV c r))
and cont : type a s. (a, s) stack -> a sint -> a ans =
  fun s i -> match s with
  | EHndl (s, h, r)              -> hndl i h r s
  | EEmp | ELift _ | ELet _ as s -> top i s
and hndl : type a s. a sint -> a sval -> a sval -> (a, s) stack -> a ans =
  fun i h r s -> match i with
  | IVal v      -> apply r v s
  | ILet (c, f) -> go c (EHndl (s, h, SFun (SHRet (f, h, r))))
  | ILift c     -> cont s (ILet (c, SAppL r))
  | IDo v       -> cont s (ILet (CApp (h, v), SAppR r))
and top : type a. a sint -> (a, top) stack -> a ans =
  fun i s -> match i with
  | IVal v      -> contT s v
  | ILet (c, f) -> go c (ELet (s, f))
  | ILift c     -> go c (ELift s)
  | IDo v       -> ADo (v, s)
and contT : type a. (a, top) stack -> a sval -> a ans =
  fun s v -> match s with
  | EEmp        -> AVal v
  | ELet (s, f) -> appFun f v s
  | ELift s     -> contT s v
and apply : type a s. a sval -> a sval -> (a, s) stack -> a ans =
  fun u v s -> match u with
  | SVar x -> AApp (x, v, s)
  | SFun f -> appFun f v s
and appFun : type a s. a sfun -> a sval -> (a, s) stack -> a ans =
  fun f v s -> match f with
  | SCl (e, r)      -> eval e (function None -> v | Some x -> r x) s
  | SAppL u         -> apply u v s
  | SAppR u         -> apply v u s
  | SHRet (f, h, r) -> appFun f v (EHndl (s, h, r))
and go : type a s. a sclo -> (a, s) stack -> a ans =
  fun c s -> match c with
  | CPair (e, r) -> eval e r s
  | CApp (u, v)  -> apply u v s
```

Fig. 19. The machine-style evaluator for our calculus.

```
let rec rba : type a. a ans -> a nf =
  fun a -> match a with
  | AVal v         -> NV (rbv v)
  | ADo (v, s)     -> NC (rbs (NDo (rbv v)) s)
  | AApp (x, v, s) -> NC (rbs (NApp (x, rbv v)) s)
and rbv : type a. a sval -> a nv =
  fun v -> match v with
  | SVar x -> NVar x
  | SFun f -> NLam (rbf f)
and rbf : type a. a sfun -> a option nf =
  fun f -> rba (appFun (sfshift f) (SVar None) EEmp)
and rbs : type a s. (a, s) nc -> (a, s) stack -> (a, top) nc =
  fun n s -> match s with
  | EEmp            -> n
  | ELet (s, f)     -> rbs (NLet (n, rbf f)) s
  | ELift s         -> rbs (NLift n) s
  | EHndl (s, h, r) -> rbs (NHndl (n, rbv h, rbv r)) s

let nbe : 'a. 'a expr -> 'a nf =
  fun e -> rba (eval e (fun x -> SVar x) EEmp)
```

Fig. 20. The read-back functions and the top-level normalization function for the machine-style evaluator.

The evaluator itself is presented in Figure 19. It differs only cosmetically from the graph presented in Figure 15—mostly by omitting configurations and implementing the interplay of different configurations via mutually recursive functions, in the classic eval-continue style. Note that certain optimisations could be performed: in particular, at a cost of a larger number of transitions, we could compile away the type of semantic closures and inline the `go` function; however, we believe that the current presentation strikes a good balance between clarity and the number of semantic constructs introduced. Finally, the read-back functions and the top-level normalization function are presented in Figure 20. The only thing of note here is the treatment of variables: since the read-back of a function needs to apply it to a fresh variable (bound by a lambda-abstraction or a `let`-expression, and represented as `SVar None`), the function needs to be *shifted* to accommodate for a larger set of the free variables.

### *A.2 Higher-order representation of functions*

While the implementation described in the previous section corresponds very closely to the presentation in the main body of the paper and the formalization, we can also *refunctionalize* the datatype of semantic functions (Ager et al. 2003), and use the function space of our chosen meta-language, OCaml, to represent the functions of our calculus. This gives rise to a more uniform treatment of semantic functions, which are now simply appropriately deferred evaluation, but at a cost of a slightly more involved implementation of the remaining parts of the evaluator, since the stacks and answers become mutually inductively defined with values and functions. The definitions are presented in Figure 21. Note that since the semantics is

```
type 'a sval =
  | SVar of 'a
  | SFun of 'a sfun
and 'a sfun =
      {run : 's 'b. ('a -> 'b) -> 'b sval -> ('b, 's) stack -> 'b ans}
and 'a ans =
  | AVal of 'a sval
  | ADo  of 'a sval * ('a, top) stack
  | AApp : 'a * 'a sval * ('a, 's) stack -> 'a ans
and ('a, _) stack =
  | EEmp : ('a, top) stack
  | ELet : ('a, top) stack * 'a sfun -> ('a, top) stack
  | ELift : ('a, top) stack -> ('a, top) stack
  | EHndl : ('a, 's) stack * 'a sval * 'a sval -> ('a, hnd) stack

type ('a, 'b) env = 'a -> 'b sval

type 'a sclo =
  | CPair : 'a expr * ('a, 'b) env -> 'b sclo
  | CApp of 'a sval * 'a sval

type 'a sint =
  | IVal  of 'a sval
  | IDo   of 'a sval
  | ILet  of 'a sclo * 'a sfun
  | ILift of 'a sclo
```

Fig. 21. Evaluator datatypes for a higher-order representation of functions.

context-sensitive, as locating a stuck-term captures the entire context, the functions need to depend not just on the argument, but also the stack in which they are called. To allow them to be called in different *kinds* of stacks, as well as to bake in the ability to rename the function, we make heavy use of OCaml's impredicative polymorphism through record fields. On the flipside, this makes "mapping" a function trivial, since the implementation unrolls to a simple function composition.

The evaluator is presented in Figure 22. Note several structural changes with respect to the one discussed in the previous section. Firstly, as expected, the constructors of semantic functions now become auxiliary definitions, some of which were inlined. However, this means that the `mkV` function, which interprets a syntactic value with an environment as a semantic value, is now defined mutually recursively with the evaluator. This is natural, as the interpretation of a lambda-abstraction is the deferred evaluation of its body, and characteristic of any evaluator with this style of representation. Finally, the `appF` function is no longer necessary, as the OCaml function can simply be applied—note that the evaluator never needs to use the renaming capabilities: this is in line with the evaluator from the previous section not using the shifting functions, and with the fact that the evaluators respect scope. In fact,

```
let hret : 'a. 'a sfun -> 'a sval -> 'a sval -> 'a sfun = fun f h r ->
  {run = fun g v s -> f.run g v (EHndl (s, svmap g h, svmap g r))}

let rec eval : type a b s. a expr -> (a, b) env -> (b, s) stack -> b ans =
  fun e r s -> match e with
  | Val v          -> cont (IVal (mkV v r)) s
  | App (u, v)     -> apply (mkV u r) (mkV v r) s
  | Let (a, e)     -> cont (ILet (CPair (a, r), clos e r)) s
  | Lift e         -> cont (ILift (CPair (e, r))) s
  | Do v           -> cont (IDo (mkV v r)) s
  | Hndl (e, h, c) -> eval e r (EHndl (s, mkV h r, mkV c r))
and cont : type a s. a sint -> (a, s) stack -> a ans =
  fun i s -> match s with
  | EHndl (s, h, r)              -> hndl i h r s
  | EEmp | ELet _ | ELift _ as s -> top i s
and hndl : type a s. a sint -> a sval -> a sval -> (a, s) stack -> a ans =
  fun i h r s -> match i with
  | IVal v      -> apply r v s
  | ILet (c, f) -> go c (EHndl (s, h, SFun (hret f h r)))
  | ILift c     -> cont (ILet (c, {run = fun f v -> apply (svmap f r) v})) s
  | IDo v       -> cont (ILet (CApp (h, v),
                         {run = fun f v -> apply v (svmap f r)})) s
and top : type a. a sint -> (a, top) stack -> a ans =
  fun i s -> match i with
  | IVal v      -> contT v s
  | ILet (c, f) -> go c (ELet (s, f))
  | ILift c     -> go c (ELift s)
  | IDo v       -> ADo (v, s)
and contT : type a. a sval -> (a, top) stack -> a ans =
  fun v s -> match s with
  | EEmp        -> AVal v
  | ELet (s, f) -> f.run (fun x -> x) v s
  | ELift s     -> contT v s
and apply : type a s. a sval -> a sval -> (a, s) stack -> a ans =
  fun u v s -> match u with
  | SVar x -> AApp (x, v, s) | SFun f -> f.run (fun x -> x) v s
and go : type a s. a sclo -> (a, s) stack -> a ans =
  fun c s -> match c with
  | CPair (e, r) -> eval  e r s | CApp  (u, v) -> apply u v s
and mkV : type a b. a value -> (a, b) env -> b sval =
  fun v r -> match v with
  | Lam e -> SFun (clos e r) | Var x -> r x
and clos : type a b. a option expr -> (a, b) env -> b sfun = fun e r ->
  {run = fun f v -> eval e (function None -> v | Some x -> svmap f (r x))}
```

Fig. 22. Evaluator with a higher-order representation of functions.

```
let rec rba : type a. a ans -> a nf =
  fun a -> match a with
  | AVal v         -> NV (rbv v)
  | ADo (v, s)     -> NC (rbs (NDo (rbv v)) s)
  | AApp (x, v, s) -> NC (rbs (NApp (x, rbv v)) s)
and rbv : type a. a sval -> a nv =
  fun v -> match v with
  | SVar x -> NVar x
  | SFun f -> NLam (rbf f)
and rbf : type a. a sfun -> a option nf =
  fun f -> rba (f.run (fun x -> Some x) (SVar None) EEmp)
and rbs : type a s. (a, s) nc -> (a, s) stack -> (a, top) nc =
  fun n s -> match s with
  | EEmp            -> n
  | ELet (s, f)     -> rbs (NLet (n, rbf f)) s
  | ELift s         -> rbs (NLift n) s
  | EHndl (s, h, r) -> rbs (NHndl (n, rbv h, rbv r)) s

let nbe : 'a. 'a expr -> 'a nf =
  fun e -> rba (eval e (fun x -> SVar x) EEmp)
```

Fig. 23. The read-back functions and the top-level normalization function for the evaluator with a higher-order representation of functions.

the representation in Figure 21 could be significantly reduced if the evaluator only worked on closed programs—although of course this would render it useless for the purpose of normalization.

The read-back functions (and the top-level normalization function), finally, are presented in Figure 23. Note that these are virtually identical to the ones for a more machine-like representation: this is to be expected, as the shapes of answers, stacks and values remain identical between the two representations. It is only the functions that change, and thus this is where the difference in read-back lies. Since in this representation the function simply requires its arguments, we provide it with a fresh variable and an empty stack—and we only need to make sure that any free variables in the body of the function are accounted for, via the renaming function `(fun x -> Some x)` that ensures that the variable passed as the argument is, in fact, fresh.

The presence of the stacks in the evaluator may be disconcerting to some readers. We note that this is due to the context-sensitive nature of the cases where stuck answers are produced. We could refunctionalize the data type of stacks to obtain an evaluator in some form of CPS (Ager et al. 2003). Mapping this continuation-passing evaluator to direct style, however, seems to require control operators (Danvy and Filinski 1990) to capture the continuations required by the shape of stuck answers.