A General Fine-Grained Reduction Theory for Effect Handlers

A General Fine-Grained Reduction Theory for Effect Handlers

FILIP SIECZKOWSKI, Heriot-Watt University, United Kingdom
MATEUSZ PYZIK, University of Wrocław, Poland
DARIUSZ BIERNACKI, University of Wrocław, Poland

Effect handlers are a modern and increasingly popular approach to structuring computational effects in functional programming languages. However, while their traditional 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 our existing reduction theory for shift0, along with a standard reduction strategy. We relate this strategy to the traditional, non-local 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 normalisation-by-evaluation algorithm for effect handlers, and prove its soundness and completeness. Additionally, we establish non-expressibility of the lift operator, found in some effect-handler calculi, by the other constructs.

CCS Concepts: • **Theory of computation** → **Operational semantics**; **Control primitives**.

Additional Key Words and Phrases: algebraic effect, delimited continuation, reduction, normalization

ACM Reference Format:
Filip Sieczkowski, Mateusz Pyzik, and Dariusz Biernacki. 2023. A General Fine-Grained Reduction Theory for Effect Handlers. *Proc. ACM Program. Lang.* 7, ICFP, Article 206 (August 2023), 30 pages. https://doi.org/10.1145/3607848

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 integration within industrial functional languages under way.[1] However, despite its highly semantic and theoretical background, certain aspects of the approach, particularly as they relate to operational semantics, remain somewhat understudied.

[1] In December 2022, OCaml 5.0 introduced effect handlers as an experimental feature, c.f. https://v2.ocaml.org/releases/5.0/manual/effects.html.

Authors' addresses: Filip Sieczkowski, f.sieczkowski@hw.ac.uk, Heriot-Watt University, School of Mathematical & Computer Sciences, Lochside Walk, Currie, Edinburgh, United Kingdom, EH14 4AS; Mateusz Pyzik, mateusz.pyzik@cs.uni.wroc.pl, Institute of Computer Science, University of Wrocław, Joliot-Curie 15, Wrocław, Poland, 50-383; Dariusz Biernacki, dabi@cs.uni.wroc.pl, Institute of Computer Science, University of Wrocław, Joliot-Curie 15, Wrocław, Poland, 50-383.

2475-1421/2023/8-ART206
https://doi.org/10.1145/3607848

In this paper, we attempt to reduce this gap by developing a reduction theory for effect handlers, together with some applications.

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.[2] 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 how operational semantics of effect handlers is usually defined. 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 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. 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:

```
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);
```

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

```
    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 as 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.[3] 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
```

[3]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 tackle it in the present paper.

```
    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:

```
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 traditional one, based on capturing parts of the evaluation

$$\begin{array}{llll} \text{term} & e, a & ::= & \mathsf{return}\, v \mid \mathsf{let}\, x = a\, \mathsf{in}\, e \mid u\, v \mid \mathsf{do}\, v \mid \mathsf{handle}\, e\, \mathsf{with}\, h, r \mid \mathsf{lift}\ e \\ \text{value} & u, v, w, h, r & ::= & x \mid \lambda\, x\, .\, e \end{array}$$

$$\begin{array}{llll} (\beta.v) & (\lambda\, x\, .\, e)\, v & \mapsto_c & e[v/x] \\ (\mathbf{DH}.v) & \mathsf{handle}\, \mathsf{return}\, v\, \mathsf{with}\, h, r & \mapsto_c & r\, v \\ (\beta.let) & \mathsf{let}\, x = \mathsf{return}\, v\, \mathsf{in}\, e & \mapsto_t & e[v/x] \\ (lift.v) & \mathsf{lift}\ \mathsf{return}\, v & \mapsto_t & \mathsf{return}\, v \end{array}$$

Fig. 1. The common syntax and contractions shared by the context-capturing and the fine-grained semantics.

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

$$(\mathbf{DH}.j) \quad \mathsf{handle}\, J_0[\mathsf{do}\, v]\, \mathsf{with}\, h, r \quad \mapsto_n \quad \mathsf{let}\, f = h\, v\, \mathsf{in}\, f\, \lambda\, x\, .\, \mathsf{handle}\, J_0[\mathsf{return}\, x]\, \mathsf{with}\, h, r$$

$$\frac{a \mapsto_c e}{K[a] \mapsto_{\lambda_{nl}\mathbf{DH}} K[e]} \qquad \frac{a \mapsto_n e}{K[a] \mapsto_{\lambda_{nl}\mathbf{DH}} K[e]} \qquad \frac{a \mapsto_t e}{K[a] \mapsto_{\lambda_{nl}\mathbf{DH}} K[e]}$$

Fig. 2. The traditional, context-capturing operational semantics of $\lambda_{nl}\mathbf{DH}$.

context as resumptions, 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, and establish confluence and standardisation results for our calculus (with the weak-head part of the standard reduction corresponding to the evaluation of the previous section). 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, and prove its soundness and completeness, and we conclude in Section 9. With the exception of the expressibility of lift in Section 2.2 all results presented in this paper are formalised in the Coq interactive theorem prover.

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 and the common reduction rules are shown in Figure 1. The syntax of the calculi is minimal – it is a fine-grain call-by-value λ-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 terms (expressions). Values comprise variables and λ-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. Terms represent possibly effectful computations. A term 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. Contrary 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.

Context-capturing calculus. The first calculus we consider is called $\lambda_{nl\mathbf{DH}}$, and it is an untyped version of a calculus of deep handlers considered by Piróg et al. [2019]. The operational semantics of $\lambda_{nl\mathbf{DH}}$ is defined by the reduction rules in Figures 1 and 2 — the contraction relation $\mapsto$ is split into relations $\mapsto_c$, $\mapsto_t$ and $\mapsto_n$ to simplify formulation of some of the technical lemmas in Section 3. The rules ($\beta.v$), (**DH**.v), ($\beta.let$) and ($lift.v$) are standard, and define the interaction of functions, handlers, let-expressions and the lift operator with values ($e[v/x]$ stands for the usual capture-avoiding substitution). In particular, handling a value (rule (**DH**.v)) consists in passing it to the return value (no effects need to be handled), whereas lifting a value (rule ($lift.v$)) is an identity operation (no effects need to be lifted).

The most instrumental reduction rule of the calculus is (**DH**.j) and it says that handling an effect is done by capturing an appropriately 'balanced' evaluation context in one step. In order to formalize the notion of a balanced context, we index evaluation contexts J by its freeness level, i.e., the number of handlers surrounding the context hole that the context would need to neutralize the lift operators that surround the context hole to skip handlers. We can see that in the grammar of J_n, lift increases the index, whereas handle decreases it. Then, balanced contexts are exactly 0-free contexts J_0, and the handler surrounding such a context is the one that matches an effect invocation within the context. The rule (**DH**.j) further specifies that the handler h is given the value passed to the effect invocation and a captured resumption (delimited continuation), represented by a 0-free context surrounded by the same handle expression, which is characteristic of deep handlers (as opposed to shallow handlers that capture a handler-free resumption [Hillerström et al. 2020]).

The evaluation relation $\mapsto_{\lambda_{nl\mathbf{DH}}}$ is then defined as a contextual closure of $\mapsto$ 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). It can be shown that $\mapsto_{\lambda_{nl\mathbf{DH}}}$ is deterministic, and that for a closed term e, either e reduces infinitely, or it evaluates to return v, or it reaches a control-stuck term $J_n[\mathsf{do}\ v]$. Moreover, $\mapsto_{\lambda_{nl\mathbf{DH}}}$ is compatible with respect to evaluation contexts, i.e., $K[e] \mapsto_{\lambda_{nl\mathbf{DH}}} K[e']$ whenever $e \mapsto_{\lambda_{nl\mathbf{DH}}} e'$.

Fine-grained calculus. The second calculus, $\lambda_{fg\mathbf{DH}}$, 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 extends the contraction rules of Figure 1, with the three additional rules shown in Figure 3.

Reduction (**DH**.do) describes a direct interaction of an effect invocation with its surrounding handler, when there is no intermediate computation separating them – the effect gets handled, so $\mathsf{handle}\ \mathsf{do}\ v\ \mathsf{with}\ h, r$ can be rewritten 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. Reduction (**DH**.$lift$) describes a direct interaction of a lift expression with its surrounding handler – the

$$\begin{array}{lll} \text{evaluation context} & E_0 ::= & [\,] \mid E_0[\mathsf{let}\, x = [\,] \,\mathsf{in}\, e] \mid E_0[\mathsf{lift}\ [\,]] \\ & E_{n+1} ::= & E_n[\mathsf{handle}\ [\,] \,\mathsf{with}\, h, r] \end{array}$$

$$\begin{array}{lll} (\mathbf{DH}.do) & \mathsf{handle}\,\mathsf{do}\, v \,\mathsf{with}\, h, r & \mapsto_f \quad \mathsf{let}\, f = h\, v \,\mathsf{in}\, f\, r \\ (\mathbf{DH}.lift) & \mathsf{handle}\,\mathsf{lift}\ e \,\mathsf{with}\, h, r & \mapsto_f \quad \mathsf{let}\, x = e \,\mathsf{in}\, r\, x \\ (\mathbf{DH}.let) & \mathsf{handle}\,\mathsf{let}\, x = a \,\mathsf{in}\, e \,\mathsf{with}\, h, r & \mapsto_f \quad \mathsf{handle}\, a \,\mathsf{with}\, h, \lambda\, x\,.\,\mathsf{handle}\, e \,\mathsf{with}\, h, r \end{array}$$

$$\frac{a \mapsto_c e}{E_n[a] \mapsto_{\lambda_{fg\mathbf{DH}}} E_n[e]} \qquad \frac{a \mapsto_f e}{E_n[a] \mapsto_{\lambda_{fg\mathbf{DH}}} E_n[e]} \qquad \frac{a \mapsto_t e}{E_0[a] \mapsto_{\lambda_{fg\mathbf{DH}}} E_0[e]}$$

Fig. 3. The fine-grained operational semantics of $\lambda_{fg\mathbf{DH}}$.

handler h is skipped and the computation e is passed to the return clause r of the handler. The most complex contraction of the fine-grained calculus, and the one that enforces its *deep* treatment of handlers, is (**DH**.*let*). 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 guarded by a copy of the handler the reduction started with. Effectively, each use of (**DH**.*let*) or (**DH**.*lift*) rule closes the distance between an effect call and its handler until they are facing each other directly, and can be eliminated by the (**DH**.*do*) rule.

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]. In these works, however, the control operator or the operation aggregates the elementary contexts on the way to its surrounding delimiter or handler. Our calculus, on the other hand, 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.[4] Both (**DH**.*do*) and (**DH**.*let*) correspond directly to rewrite rules given by Saleh [2019] (respectively, rules With-Handled-Op and With-Do in Figure 3.2); (**DH**.*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.

The grammar of evaluation contexts E_n again takes advantage of an index n, but this time its role is simpler than in the nonlocal calculus – n is the number of handlers surrounding the context hole. As a matter of fact, we only need to keep track of the fact whether the computation takes place under a handler or not, and, therefore, a boolean flag would suffice. Note that, unlike in the nonlocal calculus, contexts are defined inside-out, which for technical reasons, e.g. in Section 3, is more convenient.

The evaluation relation $\mapsto_{\lambda_{fg\mathbf{DH}}}$ is then defined as a compatible closure of the relations $\mapsto_c$, $\mapsto_f$, and $\mapsto_t$ with respect to evaluation contexts. Notice, however, that the 'top-level' reductions $\mapsto_t$ can only occur in a handle-free context E_0, since the semantics of let- and lift- expressions depends on the presence of a surrounding handler. Therefore, we have defined a so-called hybrid evaluation strategy [Biernacka et al. 2017; García-Pérez and Nogueira 2014], where the semantics of some constructs depends on the kind of the surrounding context (our contexts are kinded by the indices n in this sense). Hybrid strategies typically are used in the semantic specification of full normalization

[4]It is worth noting that such architecture of the reduction system immediately rules out a possibility of accounting for shallow handlers whose semantics assumes that the handler is never copied to the resumption, whereas the rule (**DH**.*let*) eagerly does exactly that.

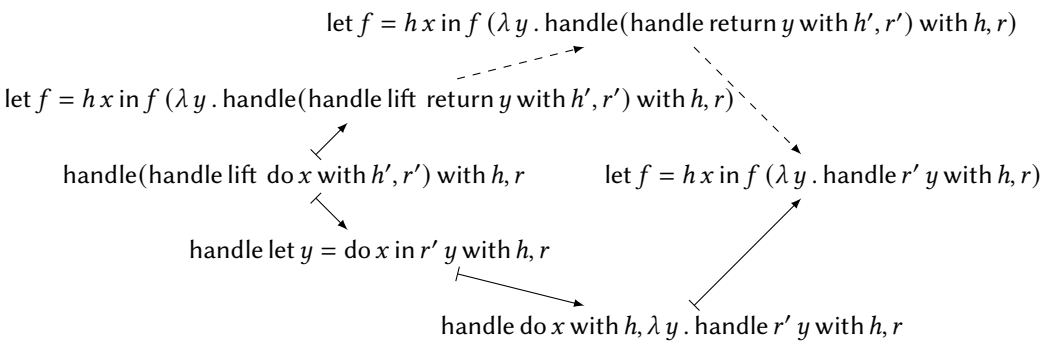

Fig. 4. Comparison of evaluation in the context-capturing and fine-grained semantics. Fine-grained semantics needs three evaluation steps to arrive at a handler call (lower path), while a single nonlocal step is sufficient (upper path). There is a mismatch between the captured resumptions that can be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

in λ-calculi. As for the traditional semantics, it can be shown that $\mapsto_{\lambda_{fg\mathrm{DH}}}$ is deterministic and that the normal forms it computes for closed expressions are returned values and control-stuck terms $E_0[\mathsf{do}\ v]$.

The semantics duplicating handlers that we have defined 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 standard 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). In order to talk about reduction viewed as code optimization, we introduce a relation $\rightarrow_{\lambda_{fg\mathrm{DH}}}$ that is a compatible closure of the contraction rules $\mapsto_t$, $\mapsto_c$, and $\mapsto_f$ with general term contexts, i.e., it allows for reduction at any position in a term (in Section 4 we show that $\rightarrow_{\lambda_{fg\mathrm{DH}}}$ is confluent). The following example demonstrates both evaluation and general reduction relations in action (such simplification could not be obtained using the traditional reduction rules):

$$\begin{aligned} & \mathsf{handle\ let}\ y = p\ x\ \mathsf{in\ do}\ y\ \mathsf{with}\ h, r \\ \mapsto_{\lambda_{fg\mathrm{DH}}} \quad & \mathsf{handle}\ p\ x\ \mathsf{with}\ h, \lambda\, y\ .\ \mathsf{handle\ do}\ y\ \mathsf{with}\ h, r \\ \rightarrow_{\lambda_{fg\mathrm{DH}}} \quad & \mathsf{handle}\ p\ x\ \mathsf{with}\ h, \lambda\, y\ .\ \mathsf{let}\ f = h\ y\ \mathsf{in}\ f\ r \end{aligned}$$

(In Sections 5 and 6, we study the formal relation between $\rightarrow_{\lambda_{fg\mathrm{DH}}}$ and $\mapsto_{\lambda_{fg\mathrm{DH}}}$.)

In the following subsection we present an additional view on the evaluation relation in the fine-grained calculus, offered by a syntactic abstract machine, whereas in Section 3 we prove that the standard and the fine-grained semantics are equivalent as far as evaluation is concerned. The abstract machine and the simulation theorems shed light on the inner workings of the new semantics and explain the architecture of the new calculus in greater detail. For now, it is instructive to compare how the context-capturing and the fine-grained semantics capture the context surrounding an effect call, which is illustrated in Figure 4.

2.1 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 goal here is to show how in the new semantics the expressions are decomposed and what the actual structure of evaluation

$$
\begin{array}{lrl}
\text{stack of handlers} & H & ::= \quad \Box \mid H[\mathsf{handle}\ \Box\ \mathsf{with}\ h, r] \\
\text{top continuation} & T & ::= \quad \Box \mid T[\mathsf{let}\ x = \Box\ \mathsf{in}\ e] \mid T[\mathsf{lift}\ \Box] \\
\text{intermediate value} & i & ::= \quad \mathsf{return}\ v \mid \mathsf{do}\ v \mid \mathsf{let}\ x = a\ \mathsf{in}\ e \mid \mathsf{lift}\ e
\end{array}
$$

$$
\begin{array}{rcl}
\langle \mathsf{handle}\ e\ \mathsf{with}\ h, r \mid H \mid T \rangle_e & \triangleright & \langle e \mid H[\mathsf{handle}\ \Box\ \mathsf{with}\ h, r] \mid T \rangle_e \\
\langle (\lambda x\,.\,e)\ v \mid H \mid T \rangle_e & \triangleright & \langle e[v/x] \mid H \mid T \rangle_e \\
\langle i \mid H \mid T \rangle_e & \triangleright & \langle H \mid i \mid T \rangle_c \\
\langle H[\mathsf{handle}\ \Box\ \mathsf{with}\ h, r] \mid i \mid T \rangle_c & \triangleright & \langle i \mid H[\mathsf{handle}\ \Box\ \mathsf{with}\ h, r] \mid T \rangle_h \\
\langle \Box \mid i \mid T \rangle_c & \triangleright & \langle i \mid T \rangle_i \\
\langle \mathsf{let}\ x = a\ \mathsf{in}\ e \mid H[\mathsf{handle}\ \Box\ \mathsf{with}\ h, r] \mid T \rangle_h & \triangleright & \langle a \mid H[\mathsf{handle}\ \Box\ \mathsf{with}\ h, \lambda x\,.\,\mathsf{handle}\ e\ \mathsf{with}\ h, r] \mid T \rangle_h \\
\langle \mathsf{return}\ v \mid H[\mathsf{handle}\ \Box\ \mathsf{with}\ h, r] \mid T \rangle_h & \triangleright & \langle r\ v \mid H \mid T \rangle_e \\
\langle \mathsf{do}\ v \mid H[\mathsf{handle}\ \Box\ \mathsf{with}\ h, r] \mid T \rangle_h & \triangleright & \langle H \mid \mathsf{let}\ x = h\ v\ \mathsf{in}\ x\ r \mid T \rangle_c \\
\langle \mathsf{lift}\ e \mid H[\mathsf{handle}\ \Box\ \mathsf{with}\ h, r] \mid T \rangle_h & \triangleright & \langle H \mid \mathsf{let}\ x = e\ \mathsf{in}\ r\ x \mid T \rangle_c \\
\langle \mathsf{let}\ x = a\ \mathsf{in}\ e \mid T \rangle_i & \triangleright & \langle a \mid \Box \mid T[\mathsf{let}\ x = \Box\ \mathsf{in}\ e] \rangle_e \\
\langle \mathsf{lift}\ e \mid T \rangle_i & \triangleright & \langle e \mid T[\mathsf{lift}\ \Box]) \rangle_i \\
\langle \mathsf{return}\ v \mid T \rangle_i & \triangleright & \langle T \mid v \rangle_p \\
\langle T[\mathsf{let}\ x = \Box\ \mathsf{in}\ e] \mid v \rangle_p & \triangleright & \langle e[v/x] \mid \Box \mid T \rangle_e \\
\langle T[\mathsf{lift}\ \Box] \mid v \rangle_p & \triangleright & \langle T \mid v \rangle_p
\end{array}
$$

Fig. 5. An abstract machine for fine-grained operational semantics. The machine starts in the configuration $\langle e \mid \Box \mid \Box \rangle_e$, for a closed expression e. The final configurations are $\langle \Box \mid v \rangle_p$, representing a computed value v, and $\langle \mathsf{do}\ v \mid T \rangle_i$, representing a control-stuck term $T[\mathsf{do}\ v]$.

contexts is. 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 has been obtained through Danvy and Nielsen's refocusing method [Biernacka et al. 2017; Danvy and Nielsen 2004] from the fine-grained reduction semantics presented above. The structure of the machine coincides with the one of Materzok and Biernacki [2011] for a calculus of delimited control, where the computation is not guaranteed to be delimited by a reset or a handler, as in our case. The evaluation context is divided into two parts. The inner part H represents a (possibly empty) stack of handlers surrounding a given expression – that it is a stack of handlers rather than a stack of expressions, each guarded by a handler, is one of the most notable differences with the traditional semantics. The outer part T represents the top-level continuation, i.e., the rest of the context beyond the outermost handler.

The machine is in refocused form which means that all it does is decomposing a term and contracting the next redex it finds, followed by repeating this procedure from the resulting configuration (no term reconstruction is necessary). The machine operates in five modes: $\langle e \mid H \mid T \rangle_e$ dispatches on the structure of the expression e being evaluated, $\langle H \mid i \mid T \rangle_c$ dispatches on the structure of the handler stack H when given an intermediate value i, $\langle i \mid H \mid T \rangle_h$ dispatches on the structure of the intermediate value i when under a handler, i.e., when H is non-empty, $\langle i \mid T \rangle_i$ dispatches on the structure of the intermediate value i when not under a handler, and $\langle T \mid v \rangle_p$

dispatches on the structure of the top-level continuation T to complete the computation when given a value v. Each configuration represents a complete state of the machine, e.g., $\langle e \mid H \mid T \rangle_e$ should be understood as a term $T[H[e]]$. The initial configuration is $\langle e \mid \Box \mid \Box \rangle_e$ for a closed expression e, whereas the final configurations are $\langle \Box \mid v \rangle_p$ and $\langle \mathsf{do}\, v \mid T \rangle_i$.

The hybrid nature of the fine-grained semantics manifests itself in the abstract machine expressly: the intermediate values are interpreted differently in the configurations $\langle i \mid H \mid T \rangle_h$ and $\langle i \mid T \rangle_i$.

2.2 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. It has been noted, however, that in a calculus with type-and-effect system for shift0, a more traditional delimited-control operator [Danvy and Filinski 1990; Kiselyov and Shan 2007; Materzok 2013], the lift operator can be straightforwardly macro expressed [Piróg et al. 2019]. 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 $\lambda_{nl\mathbf{DH}}$, but as we prove in Section 3, the two operational semantics are equivalent and the same reasoning can be carried out in $\lambda_{fg\mathbf{DH}}$. This result is new and it justifies the inclusion of lift in the calculus we consider as a construct that increases its expressive power.[5]

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

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

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

- the term constructors of $\lambda^{-}_{nl\mathbf{DH}}$ are a subset of the term constructors of $\lambda_{nl\mathbf{DH}}$
- the phrases (expressions) of $\lambda^{-}_{nl\mathbf{DH}}$ are a subset of the phrases of $\lambda_{nl\mathbf{DH}}$
- the programs (closed expressions) of $\lambda^{-}_{nl\mathbf{DH}}$ are a subset of the programs of $\lambda_{nl\mathbf{DH}}$
- the semantics of $\lambda^{-}_{nl\mathbf{DH}}$ is a restriction of the semantics of $\lambda_{nl\mathbf{DH}}$, i.e., the semantics of a program in $\lambda^{-}_{nl\mathbf{DH}}$ is given by the rules $(\beta.v)$, $(\mathbf{DH}.v)$, $(\beta.let)$, and $(\mathbf{DH}.j)$, 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 $\lambda_{nl\mathbf{DH}}$

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

where v is any closed value. Applying the reduction rules of $\lambda_{nl\mathbf{DH}}$, 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 ϕ is a map from $\lambda_{nl\mathbf{DH}}$ to $\lambda^{-}_{nl\mathbf{DH}}$ that preserves program-ness and is homomorphic on $\lambda^{-}_{nl\mathbf{DH}}$, the image of e through ϕ is the following program e'

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

[5] 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.

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 $\lambda^{-}_{nl\mathbf{DH}}$. Therefore, there does not exist a translation from $\lambda_{nl\mathbf{DH}}$ to $\lambda^{-}_{nl\mathbf{DH}}$ that makes it possible to locally express (let alone macro-express[6]) lift in terms of algebraic operations and deep handlers.

3 SIMULATION OF FINE-GRAINED EVALUATION BY NONLOCAL EVALUATION

In this section we study the relation of the two calculi defined above as evaluation theories. We establish two simulation properties, namely that:

(1) nonlocal evaluation can be simulated by fine-grained evaluation, and
(2) fine-grained evaluation can be simulated by nonlocal evaluation.

These are formally stated below.

Theorem 3.1 (Simulation of nonlocal evaluation). *If $a \mapsto^{*}_{\lambda_{nl\mathbf{DH}}} e$ and $a \rightsquigarrow a'$, then there exists e' such that $e \rightsquigarrow e'$ and $a' \mapsto^{*}_{\lambda_{fg\mathbf{DH}}} e'$.*

Theorem 3.2 (Simulation of fine-grained evaluation). *If $a \mapsto^{*}_{\lambda_{fg\mathbf{DH}}} e$ and $a' \rightsquigarrow a$, then there exists e' such that $e' \rightsquigarrow \mathrm{NF}_{\rightarrowtail}(e)$ and $a' \mapsto^{*}_{\lambda_{nl\mathbf{DH}}} e'$.*

While we only define the similarity relation in the following section, we remark that, as corollaries of these theorems we obtain the fact that if a program e evaluates to a value v using one semantics, than it evaluates to some value v' using the other, and moreover v and v' have the same structure (up to η-expansion). From this corollary, in turn, it follows that the evaluation of a given closed term e, either terminates in both semantics yielding the same kind of result (a value or a control-stuck term), or it diverges in both semantics. Thus, the semantic functions arising from the two definitions can be considered equivalent.

3.1 Similarity Relation and Flattening

Both simulations are formulated in terms of the similarity relation $\rightsquigarrow$, defined in Figure 6. The definition consists of three parts: the *structural* relation on expressions ($\sim$), the relation on values ($\sim_v$) and the similarity ($\rightsquigarrow$). These are defined in a way that enforces a structure of terms that generally matches on both sides, but accounts for the distinct nature of the two semantics.

The structural similarity, $\sim$, is built congruentially over the relation on values, enforcing a matching structure of related computations. In turn, it is used by $\rightsquigarrow$ to loosen this relationship by allowing the fine-grained evaluation to perform certain reductions. These "simulation" reductions, written $\rightarrowtail$, comprise the contractions that describe the interactions of the handle expressions with other computation formers — let and lift expressions (($\mathbf{DH}.\mathit{let}$) and ($\mathbf{DH}.\mathit{lift}$) respectively) — and values ($\mathbf{DH}.v$). We discuss the rationale behind this choice in the following. Finally, the relation on values relates values of the matching structure (which in the case of variables enforces their equality), while allowing the bodies of functions to be related by $\rightsquigarrow$. It also allows the nonlocal values to be η-longer than their fine-grained counterparts. Both of these allowances stem from the way the nonlocal calculus handles resumptions. First, their reification introduces "new" lambda-abstractions, which may lead to forms that are η-longer than on the fine-grained side. The second aspect is connected to the rationale behind the simulation reductions.

To understand the intuition behind the inclusion of simulation reductions, consider a ($\mathbf{DH}.j$)-redex: a handle expression with a do expression plugged into a 0-free context J. Since J may contain

[6]For macro-expressibility, the translation function ϕ is additionally required to be defined compositionally on the expression being eliminated, in this case, the lift expression [Felleisen 1991].

$$(\rightsquigarrow) \subseteq \lambda_{nl\mathbf{DH}} \times \lambda_{fg\mathbf{DH}}$$

$$\frac{}{x \sim_v x}\ \textsc{var} \qquad \frac{e \rightsquigarrow e'}{\lambda x\,.\,e \sim_v \lambda x\,.\,e'}\ \lambda \qquad \frac{v \sim_v \lambda x\,.\,v'\,x \qquad x \notin \mathrm{FV}(v')}{v \sim_v v'}\ \eta$$

$$\frac{v \sim_v v'}{\mathsf{return}\, v \sim \mathsf{return}\, v'}\ \textsc{ret} \qquad \frac{v \sim_v v'}{\mathsf{do}\, v \sim \mathsf{do}\, v'}\ \textsc{do} \qquad \frac{u \sim_v u' \quad v \sim_v v'}{u\,v \sim u'\,v'}\ \textsc{app}$$

$$\frac{a \sim a' \quad e \sim e'}{\mathsf{let}\, x = a\, \mathsf{in}\, e \sim \mathsf{let}\, x = a'\, \mathsf{in}\, e'}\ \textsc{let} \qquad \frac{e \sim e'}{\mathsf{lift}\ e \sim \mathsf{lift}\ e'}\ \textsc{lift}$$

$$\frac{e \sim e' \quad h \sim_v h' \quad r \sim_v r'}{\mathsf{handle}\, e\, \mathsf{with}\, h, r \sim \mathsf{handle}\, e'\, \mathsf{with}\, h', r'}\ \textsc{handle}$$

$$\frac{e \mapsto_{(\mathbf{DH}.v)} e' \vee e \mapsto_{(\mathbf{DH}.let)} e' \vee e \mapsto_{(\mathbf{DH}.lift)} e'}{E_n[e] \rightarrowtail E_n[e']}\ \textsc{step} \qquad \frac{e \sim e' \quad e' \rightarrowtail^* e''}{e \rightsquigarrow e''}\ \textsc{sim}$$

Fig. 6. Similarity relation $\rightsquigarrow$.

let- and lift-expressions, we cannot guarantee that a structurally similar term is a (**DH**.*do*)-redex. However, a fine-grained computation starting from a structurally related term will always reach such a redex by repeated application of (**DH**.*let*) and (**DH**.*lift*) rules. Since J is 0-free, we can in fact guarantee that it will be appropriately flattened and stored as part of the extended continuation of the original handle expression. Thus, we would in fact reach a (**DH**.*do*)-redex with related handlers and arguments, and a significantly extended return clause. However, this extended return clause (which takes the role of the resumption in the reduct of (**DH**.*do*)) is not, in most cases, structurally related to the resumption captured by (**DH**.*j*), since the reordering of J was already performed. In order to bridge this gap, we allow the simulation reductions to happen silently on the fine-grained side of the simulation relation and under lambda-expressions related by the similarity of values.

The flattening reduction, $\rightarrowtail$, is very simple, and trivially normalising. This is useful in the proof process, as its normal forms are very simple: they are either a return expression in a 0-level evaluation context (i.e., one that contains no handlers), or an application or do expression, this time in an arbitrary evaluation context. Thus, the slightly imprecise simulation in Theorem 3.2, which only gives us similarity with a fully flattened version of the term, is not particularly problematic.

3.2 Simulation of Nonlocal Evaluation

To simulate nonlocal evaluation, we simulate each step separately. Each step is simulated by zero or more steps of fine-grained evaluation. To simulate a single step, we employ a lemma on simulation of a contraction and certain facts about the simulation reduction, $\rightarrowtail$, and flattening.

Most of the cases of simulation of a contraction are simple. The first interesting case, (**DH**.*j*), it uses a Context Capture Lemma, described informally above, which relates how any 0-free context in a handle-expression can be moved to resumption part of a handle-expression by flattening. The other non-trivial case, ($\beta.v$), relies on a separate lemma proven by structural induction on the derivation of $\sim_v$-similarity between a λ-abstraction and a value, which needs to account for possible η-expansions.

Simulation of a single step uses also an inversion lemma for $\sim$, where the left hand side is a term plugged in a nonlocal evaluation context. It work in tandem with a procedure and lemmas for context flattening to reduce problem to simulation of a contraction.

3.3 Simulation of Fine-Grained Evaluation

For the purpose of this proof, we divide the fine-grained reductions into two categories: the flattening reductions ($\rightarrowtail$), already discussed above, and the remaining, which we dub *real*. Since the fine-grained term may be significantly reordered by flattening, we proceed by decomposing the non-local term, to establish its shape. As in the other directions, the most interesting cases arise when the expression is a (**DH**.j) or ($\beta.v$) redex in a context, and the shape of the non-local term restricts the real step that the fine-grained side could have taken (any simulation steps can be folded into the simulation relation without taking any steps on the non-local side). Although this restriction is not entirely precise (for instance, a ($\beta.let$) redex on the non-local side can give rise to either ($\beta.let$) or ($\beta.v$) redexes on the fine-grained side, since the let-expression may turn into a handle through flattening), it is sufficient to recover the necessary elements of the simulation. In the case of a (**DH**.j)-redex we need to account for the flattening of the 0-free context in a manner similar to the other direction of the proof, and the cases of ($\beta.let$)- and ($lift.v$)-redexes are only involved due to the complexities arising from flattening of the evaluation context.

The remaining two cases, however, introduce substantial complexity. This is due to the fact that both ($\beta.v$) and (**DH**.v)-redexes in the non-local calculus lead to ($\beta.v$) redexes on the fine-grained side — and the similarity relation for values may introduce an unbounded number of η-expansions on the non-local side. Moreover, since the bodies of the functions may be similar only up to flattening, this may force the non-local side to perform a significant amount of computation in order to obtain an appropriately similar substitution. The resulting technical lemma turns out to be rather complex, and we present it below.

Lemma 3.3. *Let* $e_1 \rightsquigarrow^{\mathrm{L}}_{\mathrm{NF}} e_2 \triangleq \exists e_1'.\ e_1 \mapsto^*_{\lambda_{n\mathrm{DH}}} e_1' \wedge e_1' \not\rightsquigarrow \mathrm{NF}_{\rightarrowtail}(e_2)$*, and let* $\sim_K$ *denote the natural extension of the similarity relation* $\sim$ *to evaluation contexts* K *of non-local semantics. Then, the following implications hold:*

- *if* $K_1 \sim_K K_2$, $v_1 \sim_v v_2$ *and* $u \sim_v \eta^n(\lambda x\,.\,e)$, *then* $K_1[u\,v_1] \rightsquigarrow^{\mathrm{L}}_{\mathrm{NF}} K_2[e[v_2/x]]$;
- *if* $e_1 \rightsquigarrow E[(\eta^n(\lambda x\,.\,e))\,v]$, *then* $e_1 \rightsquigarrow^{\mathrm{L}}_{\mathrm{NF}} E[e[v/x]]$;
- *if* $K_1 \sim_K K_2$, $K_2[e_2] \rightarrowtail^* E[(\eta^n(\lambda x\,.\,e)\,v)]$, $e_1 \sim e_2$ *and* $\mathrm{wf}(e_1)$, *then* $K_1[e_1] \rightsquigarrow^{\mathrm{L}}_{\mathrm{NF}} E[e[v/x]]$.

Above, $\eta^n(v)$ *denotes* n*-fold* η*-expansion of* v*, and the expression is considered* well-formed *when, assuming it is a value in an evaluation context, the context terminates with a* 0*-free part contained in a handler, i.e.,*

$$\mathrm{wf}(e) \triangleq \forall v, K.\ e = K[\mathsf{return}\,v] \Rightarrow \exists K', h, r, J_0.\ K = K'[\mathsf{handle}\,J\,\mathsf{with}\,h, r].$$

The proof proceeds by mutual induction on the three aspects of the similarity definition, with the dependency structure matching the one of the definition. For the first lemma, the case for variables is impossible, and the one for η-expansion is solved inductively. For lambda-abstraction, we can either establish the conclusion directly, if $n = 0$, or use the second lemma in the other case. The second lemma establishes well-formedness of its argument and uses the third, while the third proceeds inductively in most cases. The interesting cases here are application, when we can use the first lemma inductively, and the case for handle if the context K' resulting from the well-formedness is empty (and thus $e_1 = \mathsf{handle}\,J_0\,\mathsf{with}\,h, r$). Here, depending on the shape of J we can either use the first lemma inductively (if J is empty), or establish the conclusion directly, since in the other cases no η-expansions are possible.

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

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

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

Fig. 7. Selected rules from the definition of the super-parallel reduction $\Rrightarrow$ for the pure fragment of the calculus.

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 $\rightarrow_{\lambda_{fg\mathrm{DH}}}$ one expects, namely the confluence of the reduction relation or the Church-Rosser theorem [Barendregt 1985], expressed as the following diamond property for $\rightarrow^*_{\lambda_{fg\mathrm{DH}}}$:

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

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], which 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

(1) if $e \rightarrow_{\lambda_{fg\mathrm{DH}}} e'$ then $e \Rrightarrow_{\mathrm{e}} e'$
(2) if $e \Rrightarrow_{\mathrm{e}} e'$ then $e \rightarrow^*_{\lambda_{fg\mathrm{DH}}} e'$
(3) if $e_1 \Rrightarrow_{\mathrm{e}} e_1'$ and $e_2 \Rrightarrow_{\mathrm{e}} e_2'$ then $e_1[e_2/x] \Rrightarrow_{\mathrm{e}} e_1'[e_2'/x]$ for any x that may occur freely in e_2, e_2'.

By (1) and (2), we have that $\rightarrow^*_{\lambda_{fg\mathrm{DH}}}$ and $\Rrightarrow^*_{\mathrm{e}}$ coincide, and therefore, to establish the confluence of $\rightarrow_{\lambda_{fg\mathrm{DH}}}$, it suffices to show that $\Rrightarrow_{\mathrm{e}}$ satisfies the diamond property, which requires (3). All three properties are proved by a routine induction on either $\rightarrow_{\lambda_{fg\mathrm{DH}}}$ or $\Rrightarrow_{\mathrm{e}}$.

In Figure 7, we present a couple of representative rules defining $\Rrightarrow_{\mathrm{e}}$ for the pure fragment of the calculus. We can observe that the definition is almost identical with the one, we would write if we decided to use a parallel reduction relation instead. The only difference is in the combination of the rules s-let and nl-red, where a potentially newly created redex $\mathsf{let}\, x = \mathsf{return}\, v_1\, \mathsf{in}\, e_2'$ is contracted on the fly. In contrast, the parallel reduction relation contracts only redexes that are present in the original term. So, the parallel reduction rule that would correspond to the combination of s-let and nl-red would read:

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

This extra eagerness of the super-parallel reductions is not critical for the proof of the diamond property of $\Rrightarrow_{\mathrm{e}}$, but in conjunction with a more eager version of the complete developments that we use in the proof [McKinna 2022], it slightly simplifies the Coq formalization – we save on some inversions.

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

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

$$\frac{(r, x) \Rightarrow_{\text{app}} e_1 \quad (x, e, e_1) \Rightarrow_{\text{let}} e_2 \quad x \notin \mathrm{FV}(r)}{(\mathsf{lift}\ e, h, r) \Rightarrow_{\text{handle}} e_2}\ \textsc{nh-lift}$$

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

$$\frac{e_1 \Rrightarrow e_2 \quad h_1 \Rrightarrow_v h_2 \quad r_1 \Rrightarrow_v r_2 \quad (e_2, h_2, r_2) \Rightarrow_{\text{handle}} e}{\mathsf{handle}\ e_1\ \mathsf{with}\ h_1, r_1 \Rrightarrow_{\mathrm{e}} e}\ \textsc{s-handle}$$

Fig. 8. Excerpt from definition of superparallel reduction $\Rrightarrow$.

The remaining redexes are treated along the same lines. In particular, in Figure 8 we present the rules for the handle construct. The auxiliary relation $\Rightarrow_{\text{handle}}$ non-deterministically either reconstructs a handle expression from the given subexpressions or it contracts the redex introduced by the relation $\Rrightarrow_{\mathrm{e}}$ on the subexpression of the original term.

As for the proof of the diamond property of $\Rrightarrow_{\mathrm{e}}$, we take advantage of the notion of the complete super development of a term, which is a *deterministic* relation $\Rrightarrow_{\mathrm{e}}^{\mathrm{c}}$ ($\Rrightarrow_{\mathrm{v}}^{\mathrm{c}}$) that simultaneously contracts all the redexes existing in a term as well as those introduced by the rules such as NL-RED. More precisely, $\Rrightarrow_{\mathrm{e}}^{\mathrm{c}}$ is a subrelation of $\Rrightarrow_{\mathrm{e}}$ obtained from $\Rrightarrow_{\mathrm{e}}$ by restricting the rules NL-LET, NH-HANDLE, etc., so that they are not allowed to introduce a top-level redex. For example, in NL-LET, e_1 is not allowed to have the form return v.

Since the relation $\Rrightarrow_{\mathrm{e}}^{\mathrm{c}}$ contracts at least as many redexes as $\Rrightarrow_{\mathrm{e}}$, we are in a position to prove by induction on $\Rrightarrow_{\mathrm{e}}^{\mathrm{c}}$ the *triangle lemma* of Takahashi [1995]:

Lemma 4.2. *If* $e \Rrightarrow_{\mathrm{e}} e_1$ *and* $e \Rrightarrow_{\mathrm{e}}^{\mathrm{c}} e_2$, *then* $e_1 \Rrightarrow_{\mathrm{e}} e_2$.

For the proof of this lemma, an essential role in the definition of $\Rrightarrow_{\mathrm{e}}$ is played by the auxiliary relations such as $\Rightarrow_{\text{handle}}$. They allow the relation $\Rrightarrow_{\mathrm{e}}$ to catch up with $\Rrightarrow_{\mathrm{e}}^{\mathrm{c}}$ in the presence of critical pairs such as, e.g., in the expression

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

where reducing the outer redex leads to an intermediate term

$$\mathsf{handle}\ \mathsf{return}\ v\ \mathsf{with}\ h, \lambda x\,.\,\mathsf{handle}\ e\ \mathsf{with}\ h, r$$

that can be then reduced in two steps to

$$\mathsf{handle}\ e[v/x]\ \mathsf{with}\ h, r$$

thanks to the rule NH-RET. This rule takes advantage of the auxiliary relation $\Rightarrow_{\text{app}}$, not shown here, which can either reconstruct an application of two values or, as in our example, β-contract it.

Now, using the facts that $\Rrightarrow_{\mathrm{e}}^{\mathrm{c}}$ is functional, i.e., deterministic and total, if $e \Rrightarrow_{\mathrm{e}} e_1$ and $e \Rrightarrow_{\mathrm{e}} e_2$, we can construct two triangles given by Lemma 4.2, and compose them into the desired diamond,

such that $e_1 \Rrightarrow_{\mathrm{e}} e'$ and $e_2 \Rrightarrow_{\mathrm{e}} e'$, where e' is the unique term such that $e \Rrightarrow_{\mathrm{e}}^{\mathrm{c}} e'$ (e' does not depend on e_1 or e_2).

5 STANDARDISATION THEOREM

The standardisation theorem, 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 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 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, 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 (($\beta.let$) and ($lift.v$)), 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 9. 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 ($\mapsto_{\mathrm{i}}^*$), 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 ($\mapsto^*_{\lambda_{fg\mathrm{DH}}}$) 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 ($\twoheadrightarrow_e^{\mathrm{i}}$) and values ($\twoheadrightarrow_v^{\mathrm{i}}$), 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.

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

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

$$\frac{}{u\, v \twoheadrightarrow^{\mathrm{f}} u\, v}\ \textsc{f-app} \qquad \frac{}{\mathsf{do}\, v \twoheadrightarrow^{\mathrm{f}} \mathsf{do}\, v}\ \textsc{f-do} \qquad \frac{}{\mathsf{handle}\, e\, \mathsf{with}\, h, r \twoheadrightarrow^{\mathrm{f}} \mathsf{handle}\, e\, \mathsf{with}\, h, r}\ \textsc{f-handle}$$

$$\frac{e_1 \mapsto_{\mathrm{i}}^{*} e_2 \quad e_2 \twoheadrightarrow^{\mathrm{f}} e_3}{e_1 \twoheadrightarrow^{\mathrm{w}} e_3}\ \textsc{w}$$

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

$$\frac{a_1 \twoheadrightarrow_e^{\mathrm{i}} a_2 \quad e_1 \twoheadrightarrow^{\mathrm{s}} e_2}{\mathsf{let}\, x = a_1\, \mathsf{in}\, e_1 \twoheadrightarrow_e^{\mathrm{i}} \mathsf{let}\, x = a_2\, \mathsf{in}\, e_2}\ \textsc{i-let} \qquad \frac{e_1 \twoheadrightarrow_e^{\mathrm{i}} e_2}{\mathsf{lift}\ e_1 \twoheadrightarrow_e^{\mathrm{i}} \mathsf{lift}\ e_2}\ \textsc{i-lift} \qquad \frac{v_1 \twoheadrightarrow_v^{\mathrm{i}} v_2}{\mathsf{do}\, v_1 \twoheadrightarrow_e^{\mathrm{i}} \mathsf{do}\, v_2}\ \textsc{i-do}$$

$$\frac{u_1 \twoheadrightarrow_v^{\mathrm{i}} u_2 \quad v_1 \twoheadrightarrow_v^{\mathrm{i}} v_2}{u_1\, v_1 \twoheadrightarrow_e^{\mathrm{i}} u_2\, v_2}\ \textsc{i-app} \qquad \frac{e_1 \twoheadrightarrow^{\mathrm{f}} e_2 \quad e_2 \twoheadrightarrow_e^{\mathrm{i}} e_3 \quad h_1 \twoheadrightarrow_v^{\mathrm{i}} h_2 \quad r_1 \twoheadrightarrow_v^{\mathrm{i}} r_2}{\mathsf{handle}\, e_1\, \mathsf{with}\, h_1, r_1 \twoheadrightarrow_e^{\mathrm{i}} \mathsf{handle}\, e_3\, \mathsf{with}\, h_2, r_2}\ \textsc{i-handle}$$

$$\frac{e_1 \twoheadrightarrow^{\mathrm{w}} e_2 \quad e_2 \twoheadrightarrow_e^{\mathrm{i}} e_3}{e_1 \twoheadrightarrow^{\mathrm{s}} e_3}\ \textsc{s}$$

Fig. 9. Standard reduction ($\twoheadrightarrow^{\mathrm{s}}$), internal reduction ($\twoheadrightarrow_e^{\mathrm{i}}$)/($\twoheadrightarrow_v^{\mathrm{i}}$), frozen weak head reduction ($\twoheadrightarrow^{\mathrm{f}}$), and weak head reduction ($\twoheadrightarrow^{\mathrm{w}}$).

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:*

- *If* $e \twoheadrightarrow^{\mathrm{f}} \mathsf{return}\ v$, *then* $\mathsf{handle}\, e\, \mathsf{with}\, h, r \twoheadrightarrow^{\mathrm{w}} r\, v$.
- *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:*

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

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

- *If* $e \twoheadrightarrow^{\mathrm{f}} \mathsf{let}\, x = a\, \mathsf{in}\, e'$, *then 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* $a' \twoheadrightarrow^{\mathrm{f}} a$.
- *If* $e \twoheadrightarrow^{\mathrm{w}} \mathsf{let}\, x = a\, \mathsf{in}\, e'$, *then 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.

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. With transitivity established, it is easy to prove semi-standardisation.

Lemma 5.5 (Recomposition). *Let* ; *denote left-to-right relation composition. The following inclusions hold:*

- $(\twoheadrightarrow^{\mathrm{i}}_{e}); (\mapsto_{c}) \subseteq (\twoheadrightarrow^{\mathrm{s}})$,
- $(\twoheadrightarrow^{\mathrm{i}}_{e}); (\mapsto_{f}) \subseteq (\twoheadrightarrow^{\mathrm{s}})$,
- $(\twoheadrightarrow^{\mathrm{i}}_{e}); (\mapsto_{t}) \subseteq (\twoheadrightarrow^{\mathrm{f}}); (\twoheadrightarrow^{\mathrm{i}}_{e})$,
- $(\twoheadrightarrow^{\mathrm{i}}_{e}); (\mapsto_{\mathrm{i}}) \subseteq (\twoheadrightarrow^{\mathrm{s}})$,
- $(\twoheadrightarrow^{\mathrm{i}}_{e}); (\mapsto^{*}_{\mathrm{i}}) \subseteq (\twoheadrightarrow^{\mathrm{s}})$,
- $(\twoheadrightarrow^{\mathrm{i}}_{e}); (\twoheadrightarrow^{\mathrm{f}}) \subseteq (\twoheadrightarrow^{\mathrm{f}}); (\twoheadrightarrow^{\mathrm{i}}_{e})$,
- $(\twoheadrightarrow^{\mathrm{i}}_{e}); (\twoheadrightarrow^{\mathrm{w}}) \subseteq (\twoheadrightarrow^{\mathrm{s}})$,
- $(\twoheadrightarrow^{\mathrm{i}}_{e}); (\twoheadrightarrow^{\mathrm{i}}_{e}) \subseteq (\twoheadrightarrow^{\mathrm{i}}_{e})$,
- $(\twoheadrightarrow^{\mathrm{i}}_{v}); (\twoheadrightarrow^{\mathrm{i}}_{v}) \subseteq (\twoheadrightarrow^{\mathrm{i}}_{v})$,
- $(\twoheadrightarrow^{\mathrm{s}}); (\twoheadrightarrow^{\mathrm{s}}) \subseteq (\twoheadrightarrow^{\mathrm{s}})$.

Theorem 5.6 (Semi-standardisation). *If* $e \rightarrow^{*}_{\lambda_{\mathsf{fgDH}}} e'$, *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 nonlocal semantics. To formally state this, let us define $=_{\lambda_{fg\mathrm{DH}}}$ as a symmetric closure of $\rightarrow_{\lambda_{fg\mathrm{DH}}}$. Let us also define the notion of observational equivalence $\approx_{\lambda_{nl\mathrm{DH}}}$ as follows: $a \approx_{\lambda_{nl\mathrm{DH}}} e$ (a is observationally equivalent to e), if for all contexts C such that both $C[a]$ and $C[e]$ are closed terms, $(\exists v, C[a] \mapsto^*_{\lambda_{nl\mathrm{DH}}} \mathsf{return}\, v)$ if and only if $(\exists u, C[e] \mapsto^*_{\lambda_{nl\mathrm{DH}}} \mathsf{return}\, u)$.[7]

Corollary 6.1 (Soundness of fine-grained equational reasoning). *If* $a =_{\lambda_{fg\mathrm{DH}}} e$ *then* $a \approx_{\lambda_{nl\mathrm{DH}}} e$.

Proof. Let us assume that $a =_{\lambda_{fg\mathrm{DH}}} e$ and $C[a] \mapsto^*_{\lambda_{nl\mathrm{DH}}} v$. From the former assumption, we know that $C[a] =_{\lambda_{fg\mathrm{DH}}} C[e]$. From the latter assumption, we get that $C[a] \mapsto^*_{\lambda_{fg\mathrm{DH}}} v'$ by the simulation of nonlocal evaluation (Theorem 3.1). Therefore, we have that $C[e] =_{\lambda_{fg\mathrm{DH}}} v'$, which by confluence of $\rightarrow_{\lambda_{fg\mathrm{DH}}}$ (Theorem 4.1) gives us that $C[e] \rightarrow^*_{\lambda_{fg\mathrm{DH}}} w$ for some value w. Now, applying the standardisation theorem (Theorem 5.6), we obtain that $C[e] \mapsto^*_{\lambda_{fg\mathrm{DH}}} w'$ for some value w'. Finally, by the simulation of fine-grained evaluation (Theorem 3.2), we arrive at the conclusion $C[e] \mapsto^*_{\lambda_{nl\mathrm{DH}}} u$ for some value u. □

From the above corollary it follows that the directed reduction theory defined as $\rightarrow_{\lambda_{fg\mathrm{DH}}}$ and understood as a program optimization, as in the motivating example in Section 1, is sound.

7 NORMAL FORMS AND NORMALISATION

One of the useful consequences of the introduction of a reduction theory for effect handlers is the ability to study its normal forms and normalisation 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.

We begin by characterising normal forms of our calculus, presented in Figure 10. The normal computations (μ) consist of either normal values (ξ) or *neutral computations* (ν), with normal values encompassing variables and lambda-abstractions with normalised bodies. The remainder of our normal forms are the neutral computations: expressions that are either *control-stuck*, i.e., containing a do expression unenclosed by a handle, or a *environment-stuck*, i.e., containing an application of a variable to a (normalised) value in a (normalised) evaluation context. The distinction in the allowable contexts is achieved, by indexing them with the number of handlers, and preventing the appearance of control-stuck terms if there are handlers present. 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

[7] 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.

$$\mu ::= \xi \mid \nu_0 \qquad \text{(normal expressions)}$$
$$\xi ::= x \mid \lambda\, x\,.\, \mu \qquad \text{(normal values)}$$
$$\nu_n ::= \mathsf{do}\ \xi_{(n=0)} \mid \mathsf{let}\, x = \nu_n\, \mathsf{in}\, \mu_{(n=0)} \mid \mathsf{lift}\ \nu_{n\,(n=0)} \mid x\, \xi \mid \mathsf{handle}\, \nu_{n+1}\, \mathsf{with}\, \xi, \xi' \qquad \text{(neutral expressions)}$$

Fig. 10. Normal forms for fine-grained reduction

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 $\mathsf{let}\, x = y\, \mathsf{in}\, e$ is a valid redex. 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 computation, but not if they enclose a value: a substitution in a value cannot turn a stuck term into a redex.

We formalise the notion that our definitions characterise normal forms by the following lemmas (note that μ and ν trivially inject into expressions, and ξ into values):

Lemma 7.1. *The sets μ and ξ are precisely normal forms of expressions and values, respectively, with respect to fine-grained reduction $\rightarrow_{\lambda_{\mathrm{fgDH}}}$, i.e.,*

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

7.1 Normalisation by Evaluation

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. We begin by presenting the evaluator in two styles: as a partial function, implementable in a general-purpose programming language, such as OCaml, and as its graph, implementable as an inductively-defined relation; the latter corresponds to our formalisation in Coq. Both evaluators work utilise environment semantics rather than substitutions; therefore, we begin by introducing the necessary semantic artifacts.

Semantic constructs. The necessary components of an environment-based evaluator comprise semantic values, *functions* (lambda-like values), environments, closures (computations), stacks and answers; these are presented in Figure 11. Note that syntactic expressions appear only as parts of the closures, and the remainder of the instrumentation is achieved through simple inductive definitions.[8] For the sake of uniformity, the evaluation will operate on pairs of stacks and *configurations*, and produce an answer as a result. The first three of these configurations (annotated with E, A and F) will inspect the first element of the configuration, while the remaining four will inspect the stack.

Note that additional semantic functions and computations are present, beyond the expected closures for functions and expressions. These arise derivationally as semantic counterparts of the code that reduction rules create, as will become apparent once we see the evaluator. To

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

$$
\begin{array}{rll}
\mathsf{SVal} \ni \mathsf{v} & ::= x \mid \mathsf{f} & \text{(semantic values)} \\
\mathsf{SFun} \ni \mathsf{f} & ::= \langle x, e, \rho \rangle \mid \mathsf{v}- \mid -\mathsf{v} \mid \mathsf{H}(\mathsf{f}, \mathsf{v}, \mathsf{v}') & \text{(semantic functions)} \\
\rho \in \mathsf{SEnv} & \triangleq \mathsf{SVal}^{\mathsf{Var}} & \text{(environments)} \\
\mathsf{SComp} \ni \mathsf{c} & ::= \langle e, \rho \rangle \mid \mathsf{v}\,\mathsf{v}' & \text{(closures/semantic computations)} \\
\mathsf{SCont} \ni \mathsf{E}_n & ::= \Box_{(n=0)} \mid \mathsf{E}_n[\mathsf{let}\,\mathsf{f}]_{(n=0)} \mid \mathsf{E}_n[\mathsf{lift}]_{(n=0)} \mid \mathsf{E}_m[\mathsf{handle}\,\mathsf{v}_h\,\mathsf{v}_r]_{(n=m+1)} & \text{(stacks)} \\
\mathsf{SAns} \ni \mathsf{a} & ::= \mathsf{v} \mid \mathsf{E}_0[\mathsf{do}\,\mathsf{v}] \mid \mathsf{E}_n[x\,\mathsf{v}] & \text{(answers)} \\
\mathsf{Conf} \ni \mathsf{C} & ::= \langle \mathsf{c} \rangle_{\mathsf{E}} \mid \langle \mathsf{v}_1, \mathsf{v}_2 \rangle_{\mathsf{A}} \mid \langle \mathsf{f}, \mathsf{v} \rangle_{\mathsf{F}} \mid \langle \mathsf{v} \rangle_{\mathsf{CV}} \mid \langle \mathsf{c}, \mathsf{f} \rangle_{\mathsf{CL}} \mid \langle \mathsf{c} \rangle_{\mathsf{CF}} \mid \langle \mathsf{v} \rangle_{\mathsf{CD}} & \text{(configurations)}
\end{array}
$$

Fig. 11. Syntax instrumented for environment-based semantics.

$$
\begin{array}{rl}
\lfloor - \rfloor_{\mathsf{v}} : \mathsf{SVal} \to \mathsf{Val} & \\
\lfloor x \rfloor_{\mathsf{v}} = x & \\
\lfloor \mathsf{f} \rfloor_{\mathsf{v}} = \lambda x . \lfloor \mathsf{f} \rfloor_{\mathsf{f}}^{x} & \\
\lfloor - \rfloor_{\mathsf{f}}^{-} : \mathsf{SFun} \times \mathsf{Var} \to \mathsf{Exp} & \\
\lfloor \langle x, e, \rho \rangle \rfloor_{\mathsf{f}}^{y} = ((a \mapsto \lfloor \rho(a) \rfloor_{\mathsf{v}})[x \mapsto y])(e) & \\
\lfloor \mathsf{v}- \rfloor_{\mathsf{f}}^{x} = \lfloor \mathsf{v} \rfloor_{\mathsf{v}}\, x & \\
\lfloor -\mathsf{v} \rfloor_{\mathsf{f}}^{x} = x\, \lfloor \mathsf{v} \rfloor_{\mathsf{v}} & \\
\lfloor \mathsf{H}(\mathsf{f}, \mathsf{v}, \mathsf{v}') \rfloor_{\mathsf{f}}^{x} = \mathsf{handle} \lfloor \mathsf{f} \rfloor_{\mathsf{f}}^{x} \,\mathsf{with} \lfloor \mathsf{v} \rfloor_{\mathsf{v}}, \lfloor \mathsf{v}' \rfloor_{\mathsf{v}} &
\end{array}
\qquad
\begin{array}{rl}
\lfloor - \rfloor_{\mathsf{C}} : \mathsf{Conf} \to \mathsf{Exp} & \\
\lfloor \langle \mathsf{c} \rangle_{\mathsf{E}} \rfloor_{\mathsf{C}} = \lfloor \mathsf{c} \rfloor_{\mathsf{c}} & \\
\lfloor \langle \mathsf{v}_1, \mathsf{v}_2 \rangle_{\mathsf{A}} \rfloor_{\mathsf{C}} = \lfloor \mathsf{v}_1 \rfloor_{\mathsf{v}}\, \lfloor \mathsf{v}_2 \rfloor_{\mathsf{v}} & \\
\lfloor \langle \mathsf{f}, \mathsf{v} \rangle_{\mathsf{F}} \rfloor_{\mathsf{C}} = \lfloor \mathsf{f} \rfloor_{\mathsf{f}}^{x} [\lfloor \mathsf{v} \rfloor_{\mathsf{v}} / x] & \\
\lfloor \langle \mathsf{v} \rangle_{\mathsf{CV}} \rfloor_{\mathsf{C}} = \mathsf{return}\, \lfloor \mathsf{v} \rfloor_{\mathsf{v}} & \\
\lfloor \langle \mathsf{c}, \mathsf{f} \rangle_{\mathsf{CL}} \rfloor_{\mathsf{C}} = \mathsf{let}\, x = \lfloor \mathsf{c} \rfloor_{\mathsf{c}} \,\mathsf{in} \lfloor \mathsf{f} \rfloor_{\mathsf{f}}^{x} & \\
\lfloor \langle \mathsf{c} \rangle_{\mathsf{CF}} \rfloor_{\mathsf{C}} = \mathsf{lift}\ \lfloor \mathsf{c} \rfloor_{\mathsf{c}} & \\
\lfloor \langle \mathsf{v} \rangle_{\mathsf{CD}} \rfloor_{\mathsf{C}} = \mathsf{do}\ \lfloor \mathsf{v} \rfloor_{\mathsf{v}} &
\end{array}
$$

$$
\begin{array}{rl}
\lfloor - \rfloor_{\mathsf{c}} : \mathsf{SComp} \to \mathsf{Exp} & \\
\lfloor \langle e, \rho \rangle \rfloor_{\mathsf{c}} = (a \mapsto \lfloor \rho(a) \rfloor_{\mathsf{v}})(e) & \\
\lfloor \mathsf{v}\,\mathsf{v}' \rfloor_{\mathsf{c}} = \lfloor \mathsf{v} \rfloor_{\mathsf{v}}\, \lfloor \mathsf{v}' \rfloor_{\mathsf{v}} &
\end{array}
\qquad
\begin{array}{rl}
\mathsf{V} : \mathsf{Val} \times \mathsf{SEnv} \to \mathsf{SVal} & \\
\mathsf{V}(x, \rho) = \rho(x) & \\
\mathsf{V}(\lambda x . e, \rho) = \langle x, e, \rho \rangle &
\end{array}
$$

Fig. 12. Erasure of semantic components of the evaluator and the interpretation of values.

provide a better intuition for the interpretation of these constructs (and since they will be useful in the following) we present erasure functions for semantic values, functions, computations 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.

Evaluation and Read-Back. 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 normalisation-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 13, is an abstract-machine-style big-step semantics, reminiscent of Harper *et al.*'s account of call/cc in ML [Harper et al. 1993].[9] The evaluation proceeds

[9]Defunctionalizing a higher-order evaluator in continuation-passing style [Reynolds 1998a], expressed in a suitable meta-language, e.g., a domain theory meta-language [Reynolds 1998b] or an ML family representative [Ager et al. 2003], we

$$\frac{\langle \mathsf{V}(v,\rho)\rangle_{\mathrm{CV}}, \mathsf{E} \searrow \mathsf{a}}{\langle\langle \mathsf{return}\ v,\rho\rangle\rangle_{\mathrm{E}}, \mathsf{E} \searrow \mathsf{a}} \qquad \frac{\langle \mathsf{V}(v_1,\rho), \mathsf{V}(v_2,\rho)\rangle_{\mathrm{A}}, \mathsf{E} \searrow \mathsf{a}}{\langle\langle v_1\ v_2,\rho\rangle\rangle_{\mathrm{E}}, \mathsf{E} \searrow \mathsf{a}} \qquad \frac{\langle\langle a,\rho\rangle, \langle x,e,\rho\rangle\rangle_{\mathrm{CL}}, \mathsf{E} \searrow \mathsf{a}}{\langle\langle \mathsf{let}\ x = a\ \mathsf{in}\ e,\rho\rangle\rangle_{\mathrm{E}}, \mathsf{E} \searrow \mathsf{a}}$$

$$\frac{\langle\langle e,\rho\rangle\rangle_{\mathrm{CF}}, \mathsf{E} \searrow \mathsf{a}}{\langle\langle \mathsf{lift}\ e,\rho\rangle\rangle_{\mathrm{E}}, \mathsf{E} \searrow \mathsf{a}} \qquad \frac{\langle \mathsf{V}(v,\rho)\rangle_{\mathrm{CD}}, \mathsf{E} \searrow \mathsf{a}}{\langle\langle \mathsf{do}\ v,\rho\rangle\rangle_{\mathrm{E}}, \mathsf{E} \searrow \mathsf{a}} \qquad \frac{\langle\langle e,\rho\rangle\rangle_{\mathrm{E}}, \mathsf{E}[\mathsf{handle}\ \mathsf{V}(h,\rho)\ \mathsf{V}(r,\rho)] \searrow \mathsf{a}}{\langle\langle \mathsf{handle}\ e\ \mathsf{with}\ h,r,\rho\rangle\rangle_{\mathrm{E}}, \mathsf{E} \searrow \mathsf{a}}$$

$$\frac{\langle \mathsf{v}_1, \mathsf{v}_2\rangle_{\mathrm{A}}, \mathsf{E} \searrow \mathsf{a}}{\langle \mathsf{v}_1\ \mathsf{v}_2\rangle_{\mathrm{E}}, \mathsf{E} \searrow \mathsf{a}} \qquad \frac{}{\langle x, \mathsf{v}\rangle_{\mathrm{A}}, \mathsf{E} \searrow \mathsf{E}[x\ \mathsf{v}]} \qquad \frac{\langle \mathsf{f}, \mathsf{v}\rangle_{\mathrm{F}}, \mathsf{E} \searrow \mathsf{a}}{\langle \mathsf{f}, \mathsf{v}\rangle_{\mathrm{A}}, \mathsf{E} \searrow \mathsf{a}}$$

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

$$\frac{\langle \mathsf{v}_1, \mathsf{v}_2\rangle_{\mathrm{A}}, \mathsf{E} \searrow \mathsf{a}}{\langle -\mathsf{v}_2, \mathsf{v}_1\rangle_{\mathrm{F}}, \mathsf{E} \searrow \mathsf{a}} \qquad \frac{}{\langle \mathsf{v}\rangle_{\mathrm{CD}}, \mathsf{E}_0 \searrow \mathsf{E}_0[\mathsf{do}\ \mathsf{v}]} \qquad \frac{\langle \mathsf{v}_h\ \mathsf{v}, -\mathsf{v}_r\rangle_{\mathrm{CL}}, \mathsf{E} \searrow \mathsf{a}}{\langle \mathsf{v}\rangle_{\mathrm{CD}}, \mathsf{E}[\mathsf{handle}\ \mathsf{v}_h\ \mathsf{v}_r] \searrow \mathsf{a}}$$

$$\frac{\langle \mathsf{c}\rangle_{\mathrm{E}}, \mathsf{E}_0[\mathsf{let}\ \mathsf{f}] \searrow \mathsf{a}}{\langle \mathsf{c}, \mathsf{f}\rangle_{\mathrm{CL}}, \mathsf{E}_0 \searrow \mathsf{a}} \qquad \frac{\langle \mathsf{c}\rangle_{\mathrm{E}}, \mathsf{E}[\mathsf{handle}\ \mathsf{v}_h\ \mathsf{H}(\mathsf{f}, \mathsf{v}_h, \mathsf{v}_r)] \searrow \mathsf{a}}{\langle \mathsf{c}, \mathsf{f}\rangle_{\mathrm{CL}}, \mathsf{E}[\mathsf{handle}\ \mathsf{v}_h\ \mathsf{v}_r] \searrow \mathsf{a}}$$

$$\frac{\langle \mathsf{c}\rangle_{\mathrm{E}}, \mathsf{E}_0[\mathsf{lift}] \searrow \mathsf{a}}{\langle \mathsf{c}\rangle_{\mathrm{CF}}, \mathsf{E}_0 \searrow \mathsf{a}} \qquad \frac{\langle \mathsf{c}, \mathsf{v}_r -\rangle_{\mathrm{CL}}, \mathsf{E} \searrow \mathsf{a}}{\langle \mathsf{c}\rangle_{\mathrm{CF}}, \mathsf{E}[\mathsf{handle}\ \mathsf{v}_h\ \mathsf{v}_r] \searrow \mathsf{a}}$$

$$\frac{}{\langle \mathsf{v}\rangle_{\mathrm{CV}}, \square \searrow \mathsf{v}} \qquad \frac{\langle \mathsf{f}, \mathsf{v}\rangle_{\mathrm{F}}, \mathsf{E} \searrow \mathsf{a}}{\langle \mathsf{v}\rangle_{\mathrm{CV}}, \mathsf{E}[\mathsf{let}\ \mathsf{f}] \searrow \mathsf{a}} \qquad \frac{\langle \mathsf{v}\rangle_{\mathrm{CV}}, \mathsf{E} \searrow \mathsf{a}}{\langle \mathsf{v}\rangle_{\mathrm{CV}}, \mathsf{E}[\mathsf{lift}] \searrow \mathsf{a}} \qquad \frac{\langle \mathsf{v}_r, \mathsf{v}\rangle_{\mathrm{A}}, \mathsf{E} \searrow \mathsf{a}}{\langle \mathsf{v}\rangle_{\mathrm{CV}}, \mathsf{E}[\mathsf{handle}\ \mathsf{v}_h\ \mathsf{v}_r] \searrow \mathsf{a}}$$

Fig. 13. The evaluation judgment. The depth indices on stacks are omitted where unconstrained or clear from context.

by inspecting the configuration and — in the case of the four "continue" configurations — the stack. The eval configuration (denoted by E) inspects the closure, decomposes the expression and chooses the appropriate continuation or application configuration. The apply configuration (A) either recognises that the computation is environment-stuck, producing an appropriate answer, or continues with the application of the function. This transition is the analogue of the $(\beta.v)$ reduction of the reduction theory. 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. The two "application" functions apply one of the arguments to the other, 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. Three of these, for do (denoted with CD), let (CL) and lift (CF) only perform non-trivial computation if the context contains a handler. If it does not, a do expression is control-stuck, and thus returns the answer, while the other two

obtain a first-order tail-recursive evaluator that lends itself to two equivalent interpretations: either as an abstract machine whose transition function is defined according to the tail-calls of the first-order evaluator, or, as in this work, as a stylized single-premised big-step semantics in which derivation rules, read bottom-up, correspond to the tail-calls of the first-order evaluator.

simply extend the stack and return to the eval configuration. If the stack does contain a handler, however, the computation performs the analogues of (**DH**.*do*), (**DH**.*let*) and (**DH**.*lift*) reductions, respectively. Note that these three transitions are responsible for all the non-standard functions and computations of the machine: the reduction of the let expression creates the H function as the modified resumption of the handler, the reduction of the lift sequences the computation through what is in effect an η-expansion of the resumption, while the complex result of the reduction of do is responsible for the final two constructs. Finally, the remaining configuration (CV) represents continuation of the evaluation when a value has been encountered: this can either lead to the final result of the computation when the stack is empty, or the remaining three contractions: (*lift.v*), (β.*let*) or (**DH**.*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 normalisation, we remark that the evaluation judgment defined above corresponds with the weak-head reduction defined in Section 5.

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

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

- *for any* 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* a_2 *such that* $\mathsf{a}_1 \sim_{\mathrm{a}} \mathsf{a}_2$*;*
- *for any* 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* a_1 *such that* $\mathsf{a}_1 \sim_{\mathrm{a}} \mathsf{a}_2$*.*

The similarity relations $\sim_{\mathrm{E}}$ *and* $\sim_{\mathrm{a}}$ *are given as appropriate congruences over equality of erasures of semantic values and functions.*

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.

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* a *such that* $\langle\langle e_1, x \mapsto x \rangle\rangle_{\mathrm{E}}, \Box \searrow \mathsf{a}$ *and* $\lfloor \mathsf{a} \rfloor = e_2$*.*

More important, from the normalisation 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 $\mathsf{v} \twoheadrightarrow_{\mathrm{v}} \mathsf{v}' \triangleq \lfloor \mathsf{v} \rfloor \twoheadrightarrow^{\mathrm{i}} \lfloor \mathsf{v}' \rfloor$ and $\mathsf{f} \twoheadrightarrow_{\mathrm{f}} \mathsf{f}' \triangleq \lfloor \mathsf{f} \rfloor^x \twoheadrightarrow^{\mathrm{s}} \lfloor \mathsf{f}' \rfloor^x$, and defining the notions for stacks and answers (as $\twoheadrightarrow_{\mathrm{E}}$ and $\twoheadrightarrow_{\mathrm{a}}$ respectively) as a congruential closure over the constructors of the notions for values and functions. 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* $\lfloor \mathsf{E}_1 \rfloor [\lfloor \mathsf{C}_1 \rfloor] \twoheadrightarrow^{\mathrm{s}} \lfloor \mathsf{E}_2 \rfloor [\lfloor \mathsf{C}_2 \rfloor]$*:*

(1) *if* $\mathsf{C}_1, \mathsf{E}_1 \searrow \mathsf{a}_1$ *for some answer* a_1*, there exists* 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 answer* a_2*, there exists* a_1 *such that* $\mathsf{C}_1, \mathsf{E}_1 \searrow \mathsf{a}_1$ *and* $\mathsf{a}_1 \twoheadrightarrow_{\mathrm{a}} \mathsf{a}_2$*.*

With the evaluation aspect defined, we now turn to the read-back function, whose graph is defined inductively in Figure 14. It consists of four mutually inductive predicates: reading back answers as normal forms, semantic values as normal values, semantic functions (with a variable name as a second parameter) as normal forms, and n-deep stacks, together with n-deep neutral computations as 0-deep neutral computations. 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

$$\frac{\mathsf{RB_v}\ \mathsf{v} \searrow \xi}{\mathsf{RB_a}\ \mathsf{v} \searrow \xi} \qquad \frac{\mathsf{RB_v}\ \mathsf{v} \searrow \xi \qquad \mathsf{RB}^n_{\mathrm{E}}\ \mathsf{E}[x\ \xi] \searrow \nu}{\mathsf{RB_a}\ \mathsf{E}_n[x\ \mathsf{v}] \searrow \nu} \qquad \frac{\mathsf{RB_v}\ \mathsf{v} \searrow \xi \qquad \mathsf{RB}^0_{\mathrm{E}}\ \mathsf{E}[\mathsf{do}\ \xi] \searrow \nu}{\mathsf{RB_a}\ \mathsf{E}_0[\mathsf{do}\ \mathsf{v}] \searrow \nu}$$

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

$$\frac{}{\mathsf{RB}^0_{\mathrm{E}}\ \Box[\nu] \searrow \nu} \qquad \frac{\mathsf{RB_f}\ \mathsf{f}^x \searrow \mu \qquad \mathsf{RB}^0_{\mathrm{E}}\ \mathsf{E}[\mathsf{let}\, x = \nu\, \mathsf{in}\, \mu] \searrow \nu'}{\mathsf{RB}^0_{\mathrm{E}}\ \mathsf{E}[\mathsf{let}\ \mathsf{f}][\nu] \searrow \nu'} \qquad \frac{\mathsf{RB}^0_{\mathrm{E}}\ \mathsf{E}[\mathsf{lift}\ \nu] \searrow \nu'}{\mathsf{RB}^0_{\mathrm{E}}\ \mathsf{E}[\mathsf{lift}][\nu] \searrow \nu'}$$

$$\frac{\mathsf{RB_v}\ \mathsf{v} \searrow \xi \qquad \mathsf{RB_v}\ \mathsf{v}' \searrow \xi' \qquad \mathsf{RB}^n_{\mathrm{E}}\ \mathsf{E}[\mathsf{handle}\ \nu\ \mathsf{with}\ \xi, \xi'] \searrow \nu'}{\mathsf{RB}^{n+1}_{\mathrm{E}}\ \mathsf{E}[\mathsf{handle}\ \mathsf{v}\ \mathsf{v}'][\nu] \searrow \nu'}$$

Fig. 14. The read-back judgment.

binder in the resulting normal form). 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* a *and normal form* μ*, if* $\mathsf{RB_a}\ \mathsf{a} \searrow \mu$*, then* $\lfloor a \rfloor \twoheadrightarrow^{\mathrm{i}} \mu$*.*

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* $\mathsf{a}_1 \twoheadrightarrow_{\mathrm{a}} \mathsf{a}_2$

- *if* $\mathsf{RB_a}\ \mathsf{a}_1 \searrow \mu$ *for some* μ*, then* $\mathsf{RB_a}\ \mathsf{a}_2 \searrow \mu$*;*
- *if* $\mathsf{RB_a}\ \mathsf{a}_2 \searrow \mu$ *for some* μ*, 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 normalisation process.

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

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

is sound and complete, i.e.,

- *for any* e, μ*, if* $\mathrm{norm}(e) \searrow \mu$ *then* $e \to^*_{\lambda_{fg\mathrm{DH}}} \mu$*;*
- *for any* e_1, e_2 *if* $e_1 =_{\lambda_{fg\mathrm{DH}}} e_2$ *and* $\mathrm{norm}(e_1) \searrow \mu$ *for some* μ*, then* $\mathrm{norm}(e_2) \searrow \mu$*.*

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; 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

In [Pretnar 2015] Bauer and Pretnar present 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. 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 (**DH**.v), (**DH**.do), and (**DH**.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 λ-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 [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 [Kahn 1987]. These normalizers are in the functional correspondence [Ager et al. 2003] with fully normalizing abstract machines [Biernacka et al. 2020; 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 formalised normalisation-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.

Formalisation. As noted in the introduction, the bulk of the results presented above has been formalised using the Coq 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 formalisation uses a functorial, intrinsically well-scoped representation; this, however, is clearly an adequate choice. Except for this point, the presentation follows the formalisation closely, although there are cosmetic differences in the presentation of the evaluator in Section 7. We believe that our formalisation 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. 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.[10] 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 normalisation 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.

DATA-AVAILABILITY STATEMENT

The formalisation associated with the paper, which covers all the main results except for Section 2.2, as well as an OCaml implementation of the normaliser described in Section 7 are available online [Sieczkowski et al. 2023]. For reproduction purposes, we note that the formalisation was checked with Coq v.8.16.1, while the implementation was developed using OCaml v.4.14.0.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable comments on the presentation of our results, and in particular on related work. Thanks are also due to James McKinna for valuable discussions, particularly on the standardisation theorem. This work was partially funded by the National Science Centre of Poland, under grant number 2019/33/B/ST6/00289.

REFERENCES

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

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

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

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

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, 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

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 λ-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, 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

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

Oleg Kiselyov and Chung-chieh Shan. 2007. A Substructural Type System for Delimited Continuations. In *Typed Lambda Calculi and Applications, 8th International Conference, TLCA 2007, Paris, France, June 26-28, 2007, Proceedings (Lecture Notes in Computer Science, Vol. 4583)*, Simona Ronchi Della Rocca (Ed.). Springer, 223–239. https://doi.org/10.1007/978-3-540-73228-0_17

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. 2013. Axiomatizing Subtyped Delimited Continuations. In *Computer Science Logic 2013 (CSL 2013), CSL 2013, September 2-5, 2013, Torino, Italy (LIPIcs, Vol. 23)*, Simona Ronchi Della Rocca (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 521–539. https://doi.org/10.4230/LIPIcs.CSL.2013.521

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 λ-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

John C. Reynolds. 1998a. Definitional Interpreters for Higher-Order Programming Languages. *Higher-Order and Symbolic Computation* 11, 4 (1998), 363–397. https://doi.org/10.1023/A:1010027404223

John C. Reynolds. 1998b. *Theories of programming languages.* Cambridge University Press.

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

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, Mateusz Pyzik, and Dariusz Biernacki. 2023. *A General Fine-Grained Reduction Theory for Effect Handlers: Formalisation.* https://doi.org/10.5281/zenodo.7993545

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

Received 2023-03-01; accepted 2023-06-27