Handle with Care

Relational Interpretation of Algebraic Effects and Handlers

DARIUSZ BIERNACKI, University of Wrocław
MACIEJ PIRÓG, University of Wrocław
PIOTR POLESIUK, University of Wrocław
FILIP SIECZKOWSKI, University of Wrocław

Algebraic effects and handlers have received a lot of attention recently, both from the theoretical point of view and in practical language design. This stems from the fact that algebraic effects give the programmer unprecedented freedom to define, combine, and interpret computational effects. This plenty-of-rope, however, demands not only a deep understanding of the underlying semantics, but also access to practical means of reasoning about effectful code, including correctness and program equivalence. In this paper we tackle this problem by constructing a step-indexed relational interpretation of a call-by-value calculus with algebraic effect handlers and a row-based polymorphic type-and-effect system. Our calculus, while striving for simplicity, enjoys desirable theoretical properties, and is close to the cores of programming languages with algebraic effects used in the wild, while the logical relation we build for it can be used to reason about non-trivial properties, such as contextual equivalence and contextual approximation of programs. Our development has been fully formalised in the Coq proof assistant.

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

Additional Key Words and Phrases: algebraic effect, row polymorphism, logical relation

ACM Reference format:
Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. 2018. Handle with Care. *Proc. ACM Program. Lang.* 1, 1, Article 1 (January 2018), 34 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Programming with algebraic effects and handlers (see, e.g., [Pretnar 2015]) is a fairly novel technique of representing computational effects: exceptions, mutable state, nondeterminism, among others. Rooted in the theoretical work of Plotkin and Power [2004] and of Plotkin and Pretnar [2013], effects and their handlers are rapidly making their way into practical programming, in the form of dedicated languages [Bauer and Pretnar 2015; Leijen 2017b; Lindley et al. 2017], extensions of existing languages [Hillerström and Lindley 2016; Kammar et al. 2013], or libraries [Brady 2013b; Kiselyov and Sivaramakrishnan 2016; Schrijvers et al. 2014]. Effect handlers owe their success, in large part, to modularity: they make it possible to program against an interface, and with a number of different effects simultaneously – which is where the more traditional monadic approach often fails.

The central idea behind the handler framework is an explicit distinction between effect constructors and effect deconstructors. The former take the form of *operations*, which provide an interface used by effectful expressions. Operations are purely syntactic, that is, they do not carry any meaning by themselves. Instead, a particular meaning is provided by a *handler*: a language

2018. 2475-1421/2018/1-ART1 $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

construct that, when an operation needs to be interpreted in order to proceed with a computation, assumes control over the execution to compute the result of the operation.

It is best understood through an informal example. We first consider a reader effect, denoted R, which is useful for passing an implicit value of a type σ. The value can be retrieved with the $ask_{\mathsf{R}} : 1 \to_{\langle \mathsf{R} \rangle} \sigma$ operation.[1] An effectful expression can use the ask_{R} operation as if it were a function: for instance, let σ be *int*, and the expression be $ask_{\mathsf{R}}\ () + ask_{\mathsf{R}}\ () + 2$. A meaning of ask_{R} is given by a handler. The intended interpretation, that is, ask_{R} revealing a hidden value, say 5, can be expressed as follows:

$$\mathsf{handle}_{\mathsf{R}}\ ask_{\mathsf{R}}\ () + ask_{\mathsf{R}}\ () + 2\ \{ask\ (), r.\ r\ 5\}.$$

The part within the curly braces defines how the handler should react to each of the operations associated with the effect. In this case, there is only one such operation, ask_{R}, which is handled by replacing the effect in the original expression with 5. This is achieved through a *resumption* (in the example above denoted with a variable r): a construct that represents the result of applying the same handler to the same expression, but with the currently handled occurrence of the operation substituted with the argument of the resumption (in our case, 5).[2] In the end the entire expression we considered would evaluate to 12. In general, the handler can use any expression as the interpretation of an operation: in particular, it can resume more than once, or not resume at all – as in the following, less conventional interpretation

$$\mathsf{handle}_{\mathsf{R}}\ ask_{\mathsf{R}}\ () + ask_{\mathsf{R}}\ () + 2\ \{ask\ (), r.\ 13\}.$$

In this case, the handler assumes control when the first occurrence of ask_{R} is encountered and returns 13 immediately, rather than continue to evaluate the effectful expression.

The distinction between syntax and semantics of effects makes it easy to define multi-effect programs, since we can simply use operations from many different effects in a single expression. Then, to handle such an expression, we can enclose it in a number of handlers, one for each effect. The order in which the handlers are placed, and their behaviour on pure expressions specify the semantics of all the effects combined. To help the programmer manage the tangle of different effects, real-life languages offer different flavours of type-and-effect systems, which keep track of the defined and handled effects. One successful approach is based on so-called *rows* of effects, implemented, for example, in Leijen's Koka [2017b] (see also [Hillerström and Lindley 2016]). These enable a simple, yet powerful approach to effect polymorphism, related in spirit to the approach of Remy [1994] in the seemingly far removed context of row-polymorphic records.

On the flip side, the free hand in defining new effects by providing signatures for operations, together with handlers, which can express advanced control structures in the style of delimited continuations, and non-trivial type-and-effect systems make reasoning about programs that utilise algebraic effects and handlers particularly challenging. An example of a non-trivial property to reason about is contextual equivalence [Morris 1968]: given two expressions, it states that in any program, any occurrence of the first expression can be replaced with the second expression without changing the semantics of the program. Establishing contextual equivalence is troublesome even in the case of pure programs, and usually requires some dedicated reasoning tools, such as bisimulations [Abramsky 1990; Lassen 2005; Sangiorgi et al. 2011] or logical relations [Ahmed et al. 2009; Ahmed 2006; Benton et al. 2007; Dreyer et al. 2011; Reynolds 1983; Thamsborg and Birkedal

[1] Note the syntax that we use throughout the paper: the R symbol in the name of the operation assigns the operation to the appropriate effect, while R in the type signature of the operation is an element of a type-and-effect system; the details can be found in Section 2.

[2] We only treat *deep* handlers in the current work. *Shallow* handlers, where the handler is not replicated in the resumption, thus being effectively one-shot, are left as a future extension.

2011]. The problem becomes even more difficult when we consider programs with algebraic effects and handlers, as the (non-handled) operations present in the two expressions can be interpreted by the context.

As an instance of the problem, assume that there exists a function $f : \forall\alpha.(1 \to_\alpha \sigma) \to_\alpha \sigma$. Its type reads that it is a higher-order function that takes an effectful function as an argument, and is polymorphic in the row of effects α that both f and its argument perform. In other words, f exposes exactly the effects present latently in its argument, whatever these effects may be. We can apply it to a pure function, for example:

$$f\ (\lambda x.5) \tag{1}$$

We can also apply it to an effectful function, and handle the effects on the outside, as in the following:

$$\mathsf{handle}_{\mathrm{R}}\ f\ (\lambda x.ask_{\mathrm{R}}\ ())\ \{ask\ (), r.\ r\ 5\} \tag{2}$$

The challenge becomes to establish the contextual equivalence of (1) and (2) for all functions f of the given type. In this paper, we take up this challenge, and, more generally, the challenge of expanding the state-of-the-art relational reasoning techniques to the case of algebraic effects and handlers.

In order to enable reasoning about such examples, we use the standard technique of biorthogonality [Pitts and Stark 1998] to build a relational interpretation of a calculus with row-polymorphic algebraic effects. The logical relation we define can be used to easily show equivalence of various concrete programs, in particular relating effectful code to pure implementations, as well as more general type-and-effect-directed program equivalences. As a corollary, we also obtain a simple proof of type soundness of the calculus.

The challenge in building a biorthogonal relational interpretation of a language with algebraic effects is twofold. Firstly, in this context values are not the only sensible irreducible expressions: the second kind of normal forms are effect operations applied to a value. Thus, to check that two evaluation contexts are related, it would no longer suffice to observe their behaviour when plugged with related values. For this reason, we need to extend the framework to handle the effects of a computation properly. Moreover, since the type system includes quantification over arbitrary effect rows, the extension has to be uniform enough to allow for universal quantification over the interpretations of effect rows. We discuss the construction of the interpretation and the relevant problems in more detail in Section 3.

Interestingly, although without much surprise, we find that through the process of constructing a relational interpretation of effectful computations, we are able to glean insights into the nature of programming with algebraic effects, in particular with unrestricted row polymorphism. Thus, while we begin with a variant of lambda calculus with the call-by-value reduction strategy in the spirit of Leijen's Koka [2017b], we take advantage of this feedback loop, and extend it to obtain a minimal calculus that captures what we believe to be the essence of programming with algebraic effects. A detailed explanation of the semantics of the language can be found in Section 2; in the following we touch on the most important innovation.

In order to explore the potential of algebraic effects in a programming language more fully, we introduce a new construct, *lift*, written $[-]_l$, which dynamically creates a new occurrence of a particular effect l in the type, pushing the original occurrences deeper in the row. For example, consider the following expression:

$$ask_{\mathrm{R}}\ () + [ask_{\mathrm{R}}\ ()]_{\mathrm{R}}$$

The effect type of this program consists of two R's. Thus, to interpret this program, one needs to put it in two handlers for R. The inner handler interprets the ask_{R} operation on the left-hand side of +, while the outer handler interprets the other ask_{R}.

A more advanced example that uses this construct is defining mutable state as a composition of R and W (writer). The latter consists of a single operation $tell_{\mathsf{W}} : \sigma \to_{\langle \mathsf{W} \rangle} 1$. The interpretation of state can be obtained by plugging an expression that uses R and W in the following context:

$$\lambda s.\mathsf{handle}_{\mathsf{R}}\ \mathsf{handle}_{\mathsf{W}}\ \Box\ \{tell\ s', r.\ [\mathsf{handle}_{\mathsf{R}}\ r\ ()\ \{ask\ (), r.\ r\ s'\}]_{\mathsf{R}}\}\ \{ask\ (), r.\ r\ s\}$$

In this example, we interpret each ask_{R} with some initial state s, until $tell_{\mathsf{W}}$ is encountered. The handler for W switches the handler for R: the old handler is neutralised by enclosing the entire expression with $[\ ]_{\mathsf{R}}$, and a new handler for R (with the 'told' value used for the current state) is placed around the expression.[3] The obvious question arises: how to formally prove that this context is (observationally) equivalent to the usual handler for state? We answer this question in Section 4.

While the examples above may justify the introduction of a new calculus for programming with algebraic effects, they hardly provide an in-depth understanding of the differences between the proposed language and the alternative proposals. However, in the context of our relational interpretation, we are able to compare the different approaches to row polymorphism for algebraic effects and weigh their pros and cons. Since this comparison is informed by the construction of the interpretation of effects and effect rows, we discuss it, along with other aspects of related work, in Section 5. We conclude and discuss some of the potential directions for future work in Section 6.

Contributions. In this paper, we make the following contributions:

- We propose a novel relational interpretation for a programming language equipped with row-polymorphic algebraic effects and handlers, in effect a first semantic tool for showing program equivalences in such languages.
- We use the insights gleaned from the construction to introduce a novel programming construct that allows us to manage multiple occurrences of the same effect in a row and strengthen the parametricity of the universal quantifiers.
- We use the logical relation to justify interesting program examples, including general, type-directed equivalences, thus showing the construction is powerful.
- We formalise the language and its semantics, as well as the logical relation in Coq, thus providing the first machine-checked model of a programming language with algebraic effects and handlers.

Formalisation effort. The results presented in this paper, including some of the example proofs in Section 4, have been machine-checked using the Coq proof assistant. The development, along with an appendix that contains some additional technical material (an abstract machine that implements our calculus and a more precise definition of the logical relation), can be found at `https://bitbucket.org/pl-uwr/aleff-logrel`. This is, to our knowledge, the first such effort for a language with algebraic effects, hence we report briefly on the techniques used.

We build the step-indexed relational interpretation with the help of Polesiuk's IxFree library [Biernacki and Polesiuk 2015], which – while somewhat limited in terms of more advanced constructions – provides a helpful set of both definitions and tactics that perfectly match the scope of this development. The only other major issue, common to many formalisation developments, is the representation of open terms and (more importantly) types. We use a variant of the technique

[3]Note that while the use of lift is not crucial for ensuring that the appropriate handler is called – after all, the new handler is the innermost one – its presence is necessary to ensure that any *additional* occurrences of reader that may be present alongside the one that we use to encode the state remain uninterpreted, and can safely proceed to their intended handler.

described by [Altenkirch et al. 2014], which allows us to interpret the open types with minimal amount of boilerplate. By utilising the above two approaches, we obtain a clean and reasonably lightweight formalisation that can be used to prove concrete program equivalences.

2 A CALCULUS OF ALGEBRAIC EFFECTS

In this section we introduce a calculus of algebraic effects with handlers and lifts, called $\lambda^{\mathsf{H/L}}$. Out of the calculi discussed in the literature, it is most similar to the call-by-value core calculus of the Koka programming language [Leijen 2017b], hence we make on-the-fly comparisons between the two. The $\lambda^{\mathsf{H/L}}$ calculus is designed to be a minimal calculus that embodies the essence of row polymorphism [Hillerström and Lindley 2016; Leijen 2017b], for which, as motivated by both practical examples and the relational interpretation given in Section 3, we need to extend Koka with an additional construct and strengthen the effect subsumption rules in the type system.

2.1 Syntax

$$
\begin{array}{rll}
\mathrm{Var} \ni f, r, x, y, \ldots & & \text{(variables)} \\
\mathrm{RVar} \ni \alpha, \beta, \ldots & & \text{(row variables)} \\
\mathcal{EN} \ni l & & \text{(effect names)} \\
\mathcal{ON} \ni op & & \text{(operation names)} \\
\mathrm{Exp} \ni e & ::= v \mid e\ e \mid e * \mid & \text{(expressions)} \\
 & \quad [e]_l \mid \mathsf{handle}_l\ e\ \{h; \mathsf{return}\ x.\ e\} & \\
\mathrm{Val} \ni u, v & ::= x \mid \lambda x.e \mid \Lambda.e \mid op_l \mid () & \text{(values)} \\
h & ::= \cdot \mid op_l\ x, r.\ e; h & \text{(effect handlers)} \\
\mathrm{ECont} \ni E & ::= \Box \mid E\ e \mid v\ E \mid E * \mid & \text{(evaluation contexts)} \\
 & \quad [E]_l \mid \mathsf{handle}_l\ E\ \{h; \mathsf{return}\ x.\ e\} & \\
\mathrm{Type} \ni \sigma, \tau & ::= 1 \mid \tau \to_\varepsilon \tau \mid \forall \alpha.\tau & \text{(types)} \\
\mathrm{Eff} \ni \varepsilon & ::= \alpha \mid \langle\rangle \mid \langle l \mid \varepsilon \rangle & \text{(effects)} \\
\Delta & ::= \cdot \mid \Delta, \alpha & \text{(row contexts)} \\
\Gamma & ::= \cdot \mid \Gamma, x : \tau & \text{(variable contexts)} \\
\Sigma & ::= \cdot \mid \Sigma, l \mapsto \overline{op : \tau \to \tau} & \text{(effect contexts)}
\end{array}
$$

Fig. 1. The syntax of $\lambda^{\mathsf{H/L}}$. By convention, we omit the return clause of handlers if it is return $x.\ x$.

The syntax of $\lambda^{\mathsf{H/L}}$ is shown in Figure 1. We assume an infinite set Var of expression variables ranged over by $f, r, x, y, z, \ldots$ possibly with indices and primes. Similarly, we assume a set RVar of row variables, ranged over by $\alpha, \beta, \ldots$, that allow for abstraction over rows of effects. Effect names l are drawn from the set $\mathcal{EN}$, whereas the set $\mathcal{ON}$ contains the operation names op associated with effects. In the grammar of expressions, we make the association explicit by annotating each operation name with the corresponding effect name.

$$\frac{}{0\text{–free}(l, \Box)} \qquad \frac{n\text{–free}(l, E)}{n\text{–free}(l, E\ e)} \qquad \frac{n\text{–free}(l, E)}{n\text{–free}(l, v\ E)}$$

$$\frac{n\text{–free}(l, E)}{(n+1)\text{–free}(l, [E]_l)} \qquad \frac{n\text{–free}(l, E) \qquad l \neq l'}{n\text{–free}(l, [E]_{l'})}$$

$$\frac{(n+1)\text{–free}(l, E)}{n\text{–free}(l, \mathsf{handle}_l\ E\ \{h; \mathsf{return}\ x.\ e\})} \qquad \frac{n\text{–free}(l, E) \qquad l \neq l'}{n\text{–free}(l, \mathsf{handle}_{l'}\ E\ \{h; \mathsf{return}\ x.\ e\})}$$

Fig. 2. Definition of n-freeness of effects in an evaluation context.

Expressions. The expressions Exp and values Val include the call-by-value λ-calculus (variables are values), the observable unit value $()$, the polymorphic abstraction $\Lambda.e$ and polymorphic instantiation $e *$ presented in the style of [Ahmed 2006], and the effect-specific constructs. In particular, as in the core calculus of Koka, the grammar of values includes operation names op_l, whereas the grammar of expressions includes effect-handling expressions $\mathsf{handle}_l\ e\ \{h; \mathsf{return}\ x.\ e\}$, where $\mathsf{return}\ x.\ e$ is a return clause, and h is an effect handler for l, that is, a finite list of operation-handling expressions. The order in the list is irrelevant, but we assume that all operations associated with an effect are mentioned exactly once in a given handler. We occasionally use the metavariable d to range over phrases $\mathsf{return}\ x.\ e$, and hr to range over phrases $\{h; d\}$; we omit trivial return clauses of the form $\mathsf{return}\ x.\ x$. The calculus $\lambda^{\mathsf{H/L}}$ differs from the core calculus of Koka in that it includes lift expressions $[e]_l$, the role of which is explained in Section 2.2.

An operation handler $op_l\ x, r.\ e$ binds the variables x (representing the single argument of the operation) and r (standing for *r*esume and representing the continuation of the operation), whereas a return clause $\mathsf{return}\ x.\ e$ binds x. We use the standard notions of free variables, closed and open expressions, and we work modulo renaming of bound variables in expressions.

Evaluation contexts. The grammar of evaluation contexts fixes the left-to-right call-by-value reduction strategy for $\lambda^{\mathsf{H/L}}$. We interpret the contexts outside-in, with $\Box$ representing the hole of the context, and with $E[e]$ standing for the expression obtained by plugging context E with expression e. The notions of bound and free variables as well as of open and closed contexts carry over from the corresponding notions for expressions in the obvious way.

Both in the operational semantics, and in the relational interpretation of the calculus, we require some evaluation contexts to be appropriately *free*. We say that an effect l is n-free in a context E, written $n\text{–free}(l, E)$, when an operation from l would only be handled by $n+1$-st handle operation *outside* E, which ensures that pairs of lift and handle operations for l are well-bracketed in E. The formal definition is presented in Figure 2. We also extend the notion of freeness from natural numbers to partial maps from effect names to natural numbers, that is, for $\rho : \mathcal{EN} \hookrightarrow \mathbb{N}$, we write $\rho\text{–free}(E)$ if it is the case that $\forall l \in \mathrm{dom}(\rho).\ \rho(l)\text{–free}(l, E)$.

Types. The types we consider are: the unit type 1, an effect-row annotated arrow type $\tau \to_\varepsilon \sigma$, and a universal quantification over effect rows $\forall \alpha.\tau$ that binds α (as with expressions, we adopt the usual conventions concerning such binders). As for the effect rows, they take the form of finite lists of effect names, constructed with a cons operator $\langle l \mid \varepsilon \rangle$, and terminated either by an empty row $\langle\rangle$ (a closed row) or by a row variable α (an open row). By convention, we omit the effect annotation from the arrow type if the effect is an empty row. We assume a finite map Σ that

$$
\begin{array}{rcl}
E[(\lambda x.e)\ v] & \to & E[e\{v/x\}] \\
E[(\Lambda.e)\ *] & \to & E[e] \\
E[[v]_l] & \to & E[v] \\
E[\mathsf{handle}_l\ v\ \{h; \mathsf{return}\ x.\ e'\}] & \to & E[e'\{v/x\}] \\
E[\mathsf{handle}_l\ E'[op_l\ v]\ \{h; \mathsf{return}\ x.\ e'\}] & \to & E[e\{v/x\}\{\lambda z.\mathsf{handle}_l\ E'[z]\ \{h; \mathsf{return}\ x.\ e'\}/r\}] \\
 & & \text{if } 0\text{−free}(l, E') \text{ and } op\ x, r.\ e \in h
\end{array}
$$

Fig. 3. Operational semantics.

provides the information about the argument and the result type of all the operations for a given effect. Open types are interpreted in a row context Δ, mentioning known row variables, whereas open expressions are interpreted in a variable context Γ, associating free variables with their types.

2.2 Operational semantics

The control structure of the computation in the calculi for algebraic effects is determined by a context of layered delimited continuations guarded by effect handlers. The crucial concept of the operational semantics of such calculi, e.g., the core calculus of Koka, is that an operation op_l of an effect l is interpreted, or handled, by the nearest dynamically enclosing handler for l. What is crucial, the handler gets access not only to the argument of the operation, but also to the delimited continuation representing the context guarded by the handler. This makes it possible to handle a variety of algebraic effects, by suitably manipulating and resuming the captured continuation, so that the handler interacts with the delimited subcomputation in non-trivial ways, e.g., passing control back and forth.

In $\lambda^{\mathsf{H/L}}$, however, rather than searching for the first handler for l in the context, we search for the first handler that encloses a 0−free sub-context. The reason is that the calculus contains expressions $[e]_l$ that are intended to skip the nearest handler for l, should an unhandled operation for l occur in e, thus making it possible for the programmer to distinguish between several occurrences of the same effect and to handle them in arbitrarily different ways. As we argue in Sections 1, 2.3 and 3, such an intervention to the core calculus of Koka is necessary if one aims at a calculus where duplicate effect names in rows are treated in a comprehensive way, which is of particular importance in the presence of row polymorphism.

Example 2.1. Let us consider an expression

$$\mathsf{handle}_l\ E_2[\mathsf{handle}_l\ E_1[e]\ \{h_1\}]\ \{h_2\}$$

where E_1 and E_2 do not mention l. If $e = op_l\ v$, then the operation op_l would be handled by the inner handler h_1. If, however, $e = [op_l\ v]_l$, then the inner handler h_1 would be skipped, and the operation would be handled by the outer handler h_2. □

We define the operational semantics of the calculus as a reduction semantics with explicit representation of evaluation contexts defined in Figure 1. The semantics is presented in Figure 3, where we write $e\{v/x\}$ for the usual capture-avoiding substitution of v for x in e.

The first two rules are standard β-reduction rules for functions and polymorphic abstractions. The third and fourth rule make it explicit that values are effect-free. The last rule is at the heart of the computations in $\lambda^{\mathsf{H/L}}$. Given an application $op_l\ v$ in a 0−free context E' for l, guarded by a handler $\mathsf{handle}_l\ \square\ \{h; \mathsf{return}\ x.\ e\}$, the handling clause $op\ x, r.\ e$ for op in h is selected and the

expression e is evaluated with the operation argument v substituted for x and with the context $\mathsf{handle}_l\ E'\ \{h; \mathsf{return}\ x.\ e\}$ reified as a lambda abstraction and substituted for r.

Example 2.2. Let us assume that op_l is the single operation associated with effect l.

(1) Let E be a 0–free context for l, v – a value, and h – a handler for l defined as $op\ x, r.\ r\ v$. Then the effect l can be seen as a reader effect:

$$\begin{aligned} \mathsf{handle}_l\ E[op_l\ ()]\ \{h\} &\rightarrow \\ (\lambda z.\mathsf{handle}_l\ E[z]\ \{h\})\ v &\rightarrow \\ \mathsf{handle}_l\ E[v]\ \{h\} & \end{aligned}$$

(2) Let E be a 0–free context for l, v, v_1 and v_2 – values, and h – a handler for l defined as $op\ x, r.\ x\ (r\ v_1)\ (r\ v_2)$. Then the effect l provides a form of nondeterminism:

$$\begin{aligned} \mathsf{handle}_l\ E[op_l\ v]\ \{h\} &\rightarrow \\ v\ ((\lambda z.\mathsf{handle}_l\ E[z]\ \{h\})\ v_1)\ ((\lambda z.\mathsf{handle}_l\ E[z]\ \{h\})\ v_2) & \end{aligned}$$

(3) Let E be a 0–free context for l, v – a value, and h – a handler for l defined as $op\ x, r.\ x$. Then the effect l provides a form of primitive exceptions:

$$\begin{aligned} \mathsf{handle}_l\ E[op_l\ v]\ \{h\} &\rightarrow \\ v & \end{aligned}$$

(4) Let E be a 0–free context for l, and h – a handler for l defined as $op\ x, r.\ r\ (\lambda y.op_l\ y\ y)$. Then the effect l introduces nontermination to the system:[4]

$$\begin{aligned} \mathsf{handle}_l\ E[op_l\ ()\ ()]\ \{h\} &\rightarrow \\ (\lambda z.\mathsf{handle}_l\ E[z\ ()]\ \{h\})\ (\lambda y.op_l\ y\ y) &\rightarrow^* \\ \mathsf{handle}_l\ E[op_l\ ()\ ()]\ \{h\} & \end{aligned}$$

(5) Let E_1 and E_2 be 0–free contexts for l, v – a value, and h_1, h_2 – handlers for l defined as $h_1 = h_2 = op\ x, r.\ x$. Then we have the means to decide which handler for the effect l to choose, and either jump to E_2:

$$\begin{aligned} \mathsf{handle}_l\ E_2[\mathsf{handle}_l\ E_1[op_l\ v]\ \{h_1\}]\ \{h_2\} &\rightarrow \\ \mathsf{handle}_l\ E_2[v]\ \{h_2\} & \end{aligned}$$

or to the top level:

$$\begin{aligned} \mathsf{handle}_l\ E_2[\mathsf{handle}_l\ E_1[[op_l\ v]_l]\ \{h_1\}]\ \{h_2\} &\rightarrow \\ v & \end{aligned}$$

□

While the reduction rules are meant to describe how complete programs reduce, it is not hard to see that the reduction relation is compatible with respect to evaluation contexts, i.e., if $e \rightarrow e'$, then $E[e] \rightarrow E[e']$. It is also deterministic. We write $e \not\rightarrow$ when e is a normal form.

[4]Note that the effect l is used in a recursive fashion in the handler: in principle, this usage pattern could be disallowed through the type system. We do not explore this design choice, keeping recursion through effects and handlers as the primitive source of recursion. However, we posit that a restriction on recursive occurrence of effects would render the calculus terminating.

$$\frac{}{\Sigma;\Delta \vdash \varepsilon \le \varepsilon} \qquad \frac{\Sigma;\Delta \vdash \varepsilon_1 \le \varepsilon_2 \qquad \Sigma;\Delta \vdash \varepsilon_2 \le \varepsilon_3}{\Sigma;\Delta \vdash \varepsilon_1 \le \varepsilon_3} \qquad \frac{}{\Sigma;\Delta \vdash \langle l_1, l_2 \mid \varepsilon\rangle \le \langle l_2, l_1 \mid \varepsilon\rangle}$$

$$\frac{}{\Sigma;\Delta \vdash \langle\rangle \le \varepsilon} \qquad \frac{\Sigma;\Delta \vdash \varepsilon_1 \le \varepsilon_2}{\Sigma;\Delta \vdash \langle l \mid \varepsilon_1\rangle \le \langle l \mid \varepsilon_2\rangle}$$

$$\frac{}{\Sigma;\Delta \vdash \tau \le \tau} \qquad \frac{\Sigma;\Delta \vdash \tau_1 \le \tau_2 \qquad \Sigma;\Delta \vdash \tau_2 \le \tau_3}{\Sigma;\Delta \vdash \tau_1 \le \tau_3} \qquad \frac{\Sigma;\Delta,\alpha \vdash \tau_1 \le \tau_2}{\Sigma;\Delta \vdash \forall\alpha.\tau_1 \le \forall\alpha.\tau_2}$$

$$\frac{\Sigma;\Delta \vdash \sigma_2 \le \sigma_1 \qquad \Sigma;\Delta \vdash \varepsilon_1 \le \varepsilon_2 \qquad \Sigma;\Delta \vdash \tau_1 \le \tau_2}{\Sigma;\Delta \vdash \sigma_1 \to_{\varepsilon_1} \tau_1 \le \sigma_2 \to_{\varepsilon_2} \tau_2}$$

Fig. 4. The type-and-effect system. Subtyping and effect subsumption judgements. We implicitly assume that all types and effects are well-formed in the given contexts.

2.3 Type system

The subtyping and typing relations for $\lambda^{\mathsf{H/L}}$ implicitly assume that the types and effects are well-formed in the appropriate contexts, defined in a standard way. Just as in the core calculus of Koka, the system allows duplicate effect labels in a row.

The subtyping and effect subsumption relations are shown in Figure 4. Note how the subsumption rules allow for both opening of closed rows, and for contravariant weakening of argument types, thus strengthening the subsumption rules of Koka.

The typing relation is presented in Figure 5. The first four rules assign effect-free types, i.e., such that their effect rows are empty, to values. Such types can be coerced to appropriate effectful types with the subtyping rule present in the system. Note that we do not prove soundness of the system using progress and preservation: rather, type soundness follows as a corollary of the fundamental property of the logical relation, established as Theorem 9.

Functions, denoted with λ-abstractions, are given an arrow type annotated with an effect row describing the possible effects of evaluating the body of the λ-abstraction. Operations are also given a functional type, where it is assumed that their only effect is the one they are associated with. The rule for application makes an assumption that all the effect rows mentioned by the function and the argument are equal. Again, thanks to the subtyping rule, this is not a serious restriction.

In the rule for polymorphic abstraction we require that the abstracted expression be effect-free – a requirement that is sufficient, but weaker than the usual value restriction. The rule for polymorphic instantiation allows for instantiating a type scheme with any well-formed effect row.

The rule for effect handlers takes advantage of an auxiliary judgement $\Sigma;\Delta;\Gamma \vdash_l h : \tau \;/\; \varepsilon$, annotated with effect name l that defines how operation handlers for l are typed. Let us consider an expression $\mathsf{handle}_l\ E[op_l\ v]\ \{h\}$, where $op_l : \tau_1 \to \tau_2$ and E is a 0−free context that can be plugged with values of type τ_2 and produces values of type τ, possibly with effect ε. Assuming that $op\ x, r.\ e \in h$, the reduction rule for effect handlers determines that if the operation argument x is of type τ_1 and the resumption r is of type $\tau_2 \to_\varepsilon \tau$, then e should be of type τ. As for the expression guarded by the handler, its type should agree with the argument type of the return operation, while its effects should include l – the effect handled by the handler.

$$\frac{}{\Sigma;\Delta;\Gamma \vdash () : 1 \;/\; \langle\rangle} \qquad \frac{x : \tau \in \Gamma}{\Sigma;\Delta;\Gamma \vdash x : \tau \;/\; \langle\rangle} \qquad \frac{op : \sigma \to \tau \in \Sigma(l)}{\Sigma;\Delta;\Gamma \vdash op_l : \sigma \to_{\langle l \rangle} \tau \;/\; \langle\rangle}$$

$$\frac{\Sigma;\Delta;\Gamma, x : \sigma \vdash e : \tau \;/\; \varepsilon}{\Sigma;\Delta;\Gamma \vdash \lambda x.e : \sigma \to_\varepsilon \tau \;/\; \langle\rangle} \qquad \frac{\Sigma;\Delta;\Gamma \vdash e_1 : \sigma \to_\varepsilon \tau \;/\; \varepsilon \qquad \Sigma;\Delta;\Gamma \vdash e_2 : \sigma \;/\; \varepsilon}{\Sigma;\Delta;\Gamma \vdash e_1\ e_2 : \tau \;/\; \varepsilon}$$

$$\frac{\Sigma;\Delta,\alpha;\Gamma \vdash e : \tau \;/\; \langle\rangle}{\Sigma;\Delta;\Gamma \vdash \Lambda.e : \forall\alpha.\tau \;/\; \langle\rangle} \qquad \frac{\Sigma;\Delta;\Gamma \vdash e : \forall\alpha.\tau \;/\; \varepsilon}{\Sigma;\Delta;\Gamma \vdash e * : \tau\{\varepsilon'/\alpha\} \;/\; \varepsilon}$$

$$\frac{\Sigma;\Delta;\Gamma \vdash e : \tau_1 \;/\; \varepsilon_1 \qquad \Sigma;\Delta \vdash \tau_1 \le \tau_2 \qquad \Sigma;\Delta \vdash \varepsilon_1 \le \varepsilon_2}{\Sigma;\Delta;\Gamma \vdash e : \tau_2 \;/\; \varepsilon_2} \qquad \frac{\Sigma;\Delta;\Gamma \vdash e : \tau \;/\; \varepsilon}{\Sigma;\Delta;\Gamma \vdash [e]_l : \tau \;/\; \langle l \mid \varepsilon\rangle}$$

$$\frac{\Sigma;\Delta;\Gamma \vdash e : \sigma \;/\; \langle l \mid \varepsilon\rangle \qquad \Sigma;\Delta;\Gamma \vdash_l h : \tau \;/\; \varepsilon \qquad \Sigma;\Delta;\Gamma, x : \sigma \vdash e_r : \tau \;/\; \varepsilon}{\Sigma;\Delta;\Gamma \vdash \mathsf{handle}_l\ e\ \{h; \mathsf{return}\ x.\ e_r\} : \tau \;/\; \varepsilon}$$

$$\frac{}{\Sigma;\Delta;\Gamma \vdash_l \cdot : \tau \;/\; \varepsilon} \qquad \frac{\begin{array}{c} op : \tau_1 \to \tau_2 \in \Sigma(l) \qquad \Sigma;\Delta;\Gamma \vdash_l h : \tau \;/\; \varepsilon \\ \Sigma;\Delta;\Gamma, x : \tau_1, r : \tau_2 \to_\varepsilon \tau \vdash e : \tau \;/\; \varepsilon \end{array}}{\Sigma;\Delta;\Gamma \vdash_l op\ x, r.\ e; h : \tau \;/\; \varepsilon}$$

Fig. 5. The type-and-effect system. The typing relation. We implicitly assume that all types and effects are well-formed in the given contexts.

Example 2.3. (1) Let op_l be the single operation associated with effect l and $op : 1 \to (1 \to_\varepsilon 1) \in \Sigma(l)$. Then for any Γ and Δ one can derive:

$$\Sigma;\Delta;\Gamma \vdash \mathsf{handle}_l\ op_l\ ()\ ()\ \{op\ x, r.\ r\ (\lambda y.op_l\ y\ y)\} : 1 \;/\; \varepsilon$$

Interestingly, in this example the type of the resume parameter r is $(1 \to_{\langle l \mid \varepsilon\rangle} 1) \to_\varepsilon 1$, which shows that the resumption handles effect l, frozen in the thunk expected as argument for r.

(2) Taking advantage of the fact $\Sigma;\Delta \vdash \langle\rangle \le \alpha$ (with $\alpha \in \Delta$), expressing row opening, we can show that the following judgement is derivable:

$$\Sigma;\Delta;\Gamma \vdash \Lambda.\lambda f.f\ () : \forall\alpha.(1 \to_\alpha \tau) \to_\alpha \tau \;/\; \varepsilon$$

□

In $\lambda^{\mathsf{H/L}}$ we can define functions that are effect-polymorphic. Even with dynamic nature of effect handlers, our effect polymorphism is parametric, which informally means that polymorphic functions cannot make any decisions based on instantiation of polymorphic variables.

Example 2.4. Let f be a polymorphic function of type $\forall\alpha.(\tau_1 \to_\alpha \tau_2) \to_\alpha \tau_3$. Using effect T with a single operation *tick* of type $1 \to 1$, we could try to define another polymorphic function f_{cnt} that for any g counts how many times $f\ g$ uses its parameter. Our first attempt would be:

$$\begin{array}{l} f_{cnt} = \Lambda.\lambda g.\mathsf{handle}_\mathsf{T}\ f\ *\ (\lambda x.tick_\mathsf{T}\ (); g\ x) \\ \qquad\qquad \{tick\ \_, r.\ \lambda n.r\ ()\ (n+1) \\ \qquad\qquad ; \mathsf{return}\ \_.\ \lambda n.n \\ \qquad\qquad \}\ 0 \end{array}$$

The idea is to apply f to a function that behaves like g, but also raises the $tick_{\mathsf{T}}$ operation. The handler for this operation is stateful and increases state by one (stateful handlers are represented in a standard way as an application to a state value). When f finishes its computation, the result is discarded and the current state is returned.

Unfortunately, this implementation of f_{cnt} does not work as we may expect. If the function g uses the effect T, the $tick_{\mathsf{T}}$ operation will be handled by the internal handler of f_{cnt} and not passed to the outside context. Surprisingly, we cannot break parametricity this way: the function f_{cnt} has type $\forall\alpha.(\tau_1 \to_{\langle \mathsf{T}|\alpha\rangle} \tau_2) \to_\alpha int$, so it is authorised to handle effect T of g. If we tried to give f_{cnt} a more polymorphic type $\forall\alpha.(\tau_1 \to_\alpha \tau_2) \to_\alpha int$, we would fail, as we cannot coerce α to $\langle \mathsf{T} \mid \alpha\rangle$ using the subtyping rules. This problem can be solved by guarding $g\ x$ by a lift construct, as follows:

$$\begin{aligned} f'_{cnt} = \Lambda.\lambda g.\mathsf{handle}_{\mathsf{T}}\ f\ *\ (\lambda x.tick_{\mathsf{T}}\ (); [g\ x]_{\mathsf{T}}) \\ \{tick\ \_, r.\ \lambda n.r\ ()\ (n+1) \\ ; \mathsf{return}\ \_.\ \lambda n.n \\ \}\ 0 \end{aligned}$$

The function f'_{cnt} has the desired type and behaves as expected, even if effect variable α is instantiated with a row that contains the effect T. Note that in this case, the effect of the function $\lambda x.tick_{\mathsf{T}}\ (); [g\ x]_{\mathsf{T}}$ contains multiple occurrences of T.

2.4 Implementation

The calculus $\lambda^{\mathsf{H/L}}$ can be naturally implemented on a CEK-like abstract machine [Felleisen and Friedman 1986], where, as in typical abstract machines for delimited control, the complete continuation is represented by a stack and a meta-stack [Biernacka et al. 2005]. The meta-stack is enriched with markers [Dybvig et al. 2007] corresponding to lift expressions and effect handlers, and the additional transitions take care of searching stack for the right handler, and for resuming (copying) a captured continuation. An abstract machine for algebraic effects and handlers defined in this style has been recently presented by [Hillerström and Lindley 2016]. The presence of the lift operator slightly complicates the architecture of the abstract machine in that it requires counting the stack markers related to a given effect, when searching for the right handler. However, since the architectural differences are rather limited and the machine has little bearing on the remainder of this paper, we refrain from presenting its definition.

3 THE LOGICAL RELATION

3.1 Step-indexing

In order to interpret the calculus presented in the previous section, we build a logical relation that is step-indexed [Appel and McAllester 2001]. The core idea of step-indexing lies in parametrising the logical relation by a natural number n, the eponymous *step-index*, to describe the behaviour of a program over the first n computation steps. This additional parameter allows to use not only the structure of types, but also the inductive structure of natural numbers to define the relation and reason about it. Step-indexing is necessary for $\lambda^{\mathsf{H/L}}$ because effect signature may be recursive: types in an effect signature $\Sigma(l)$ may contain effect l.

Unfortunately, when we define step-indexed logical relations directly, the definition and proofs are obscured by ubiquitous step-index arithmetic. In order to hide indices and give a concise, readable definition we work in the category **COFE** of complete ordered families of equivalences, introduced by Di Gianantonio and Miculan [2002], which is recently enjoying more widespread use in similar contexts. This means that throughout the paper formulas (and relations) are implicitly indexed: they are interpreted as monotone non-increasing sequences of truth values. All the logical

$$
\begin{aligned}
(v_1, v_2) \in [\![1]\!]_\eta &\iff v_1 = v_2 = () \\
(v_1, v_2) \in [\![\tau_1 \to_\varepsilon \tau_2]\!]_\eta &\iff \forall (u_1, u_2) \in [\![\tau_1]\!]_\eta.\ (v_1\ u_1, v_2\ u_2) \in \mathcal{E}[\![\tau_2 \ / \ \varepsilon]\!]_\eta \\
(v_1, v_2) \in [\![\forall \alpha.\tau]\!]_\eta &\iff \forall R \in \mathbf{Eff}.\ (v_1\ *, v_2\ *) \in \mathcal{E}[\![\tau \ / \ \langle\rangle]\!]_{\eta[\alpha \mapsto R]}
\end{aligned}
$$

Fig. 6. Interpretation of types. We assume that the type is well-formed in implicit parameters Σ and Δ, and that $\eta \in \mathbf{Eff}^\Delta$ throughout.

connectives are interpreted pointwise, except the implication: to ensure monotonicity, $A \Rightarrow B$ is valid at index n iff A pointwise implies B for every index $k \leq n$. With this interpretation all inference rules of intuitionistic logic are valid.

Later modality and recursive predicates. In order to force a step-index to decrement, we use a special connective: the *later* modal operator ($\triangleright$), which shifts the interpretation by one. The formula $\triangleright A$ is always valid at index 0, and is valid at $n + 1$ iff A is valid at index n. The later modality distributes over all other connectives (except the existential quantifier) and has two additional inference rules, that we use in proofs:

$$\frac{A, B \vdash C}{A, \triangleright B \vdash \triangleright C} \qquad \frac{A, \triangleright B \vdash B}{A \vdash B}$$

The former rule ($\triangleright$-introduction) allows to shift reasoning to a future world with a smaller index. To prove $\triangleright C$, we may remove all $\triangleright$ modalities in assumptions and show C. The latter rule is the Löb induction principle – a variant of well-founded induction on indices introduced in this context in [Appel et al. 2007]. We also generalise the $\triangleright$ operator for any relation.

One of the key features of step-indexed logical relations is that they can be defined using recursion over indices, which is also possible when indices are hidden in the interpretation of the logic. A predicate can be defined recursively provided that the recursive occurrences are guarded by the later operator, i.e., they are used with strictly smaller index.

In order to simplify presentation, we use set-theoretic notations in the definitions of the (step-indexed) relations, keeping in mind the interpretation discussed above. The proofs in Section 4 are done directly in the logic sketched above, so they also keep all the indices implicit.

3.2 Interpretation of types and effects

We define a space **Type** of *semantic types* and a space **Eff** of *semantic effects*. For the semantic types, we take the standard space of step-indexed relations over closed values, while the space of semantic effects is more complex:

$$
\begin{aligned}
\mathbf{Type} &\equiv \mathbf{UPred}(\mathrm{Val}^2) \\
\mathbf{Eff} &\equiv \mathbf{UPred}(\mathrm{Exp}^2 \times (\mathcal{EM} \hookrightarrow \mathbb{N})^2 \times \mathbf{UPred}(\mathrm{Exp}^2))
\end{aligned}
$$

The $\mathbf{UPred}(X)$ denotes the space of (step-indexed) predicates over a (non-indexed) set X. We explain the role of the various components of this definition in the following; for now we only claim that it is a well-formed **COFE**.[5]

The interpretation of types in the space **Type** is given in Figure 6. Most things are standard here: unit values are related at the unit type and functional values are related if they map related

[5]In a calculus with kinded type variables, we could take these exact spaces as the interpretations of the base kinds of types and effect rows. Due to the cartesian-closed nature of **COFE**, this treatment would naturally extend to product and arrow kinds.

$$
\begin{aligned}
(e_1, e_2) \in \mathcal{E}[\![\tau \;/\; \varepsilon]\!]_\eta &\iff \forall (E_1, E_2) \in \mathcal{K}[\![\tau \;/\; \varepsilon]\!]_\eta.\ (E_1[e_1], E_2[e_2]) \in \mathbf{Obs} \\
(E_1, E_2) \in \mathcal{K}[\![\tau \;/\; \varepsilon]\!]_\eta &\iff \forall (v_1, v_2) \in [\![\tau]\!]_\eta.\ (E_1[v_1], E_2[v_2]) \in \mathbf{Obs} \wedge \\
&\qquad \forall (e_1, e_2) \in \mathcal{S}[\![\tau \;/\; \varepsilon]\!]_\eta.\ (E_1[e_1], E_2[e_2]) \in \mathbf{Obs} \\
(E_1[e_1], E_2[e_2]) \in \mathcal{S}[\![\tau \;/\; \varepsilon]\!]_\eta &\iff \exists \rho_1, \rho_2, \mu.\ (e_1, e_2, \rho_1, \rho_2, \mu) \in [\![\varepsilon]\!]_\eta \wedge \\
&\qquad \rho_1\text{–free}(E_1) \wedge \rho_2\text{–free}(E_2) \wedge \\
&\qquad \forall (e_1', e_2') \in \mu.\ (E_1[e_1'], E_2[e_2']) \in \triangleright \mathcal{E}[\![\tau \;/\; \varepsilon]\!]_\eta \\
(e_1, e_2) \in \mathbf{Obs} &\iff (e_1 = () \wedge e_2 \to^* ()) \vee \exists e_1'.(e_1 \to e_1' \wedge (e_1', e_2) \in \triangleright \mathbf{Obs}).
\end{aligned}
$$

Fig. 7. Closure operation for expressions, evaluation contexts and simple expressions.

arguments to related results. Since we interpret open types, due to the presence of polymorphism, the relation is parametrised by an interpretation of row variables (denoted by a metavariable η). The denotation of a polymorphic type reveals the essence of parametricity: two polymorphic values are related if they are related for any possible interpretation of the quantified row variable. The two things of note are the explicit occurrence of the empty effect row in the interpretation of the polymorphic type, as our type system requires these types to be pure, and the fact that we build appropriate applications at functional and polymorphic types ($v_i\ u_i$ and $v_i\ *$), rather than inspect the construction of values. This design choice allows us to treat the two classes of arrow-typed values, i.e., algebraic operations and functions, in a uniform fashion.

The interpretation of computations, $\mathcal{E}[\![-]\!]$, is shown in Figure 7 and is defined as a biorthogonal closure [Pitts and Stark 1998]: two expressions are related if they behave the same in any related contexts. Two complete programs e_1 and e_2 have the same behaviour (written $(e_1, e_2) \in \mathbf{Obs}$) for index n if and only if termination of e_1 to unit value in at most n steps implies $e_2 \to^* ()$ (note that e_2 may use any number of reduction steps). Since we can define guarded recursive predicates, the relation **Obs** is defined without mentioning any indices.

The interesting and novel fragment, and the capstone of our treatment of effects is the relation for evaluation contexts, $\mathcal{K}[\![-]\!]$. The computation of an effectful expression can be interrupted in two ways – by returning a value or by raising an effect. This intuition is reflected in the definition of $\mathcal{K}[\![-]\!]$: two contexts are related if they behave in the same way if plugged with related values or related expressions that raise an effect (elements of $\mathcal{S}[\![-]\!]$). The relation $\mathcal{S}[\![-]\!]$ can be thought of as a relation on expressions that are *control-stuck*, i.e., ones where an unhandled algebraic operation is found in the evaluation position.[6] In the presence of polymorphism, the elements of $\mathcal{S}[\![-]\!]$ are not necessarily control-stuck,[7] so we call them *simple expressions*. Note that $\mathcal{S}[\![-]\!]$ has to be larger than the denotation of effect rows: control-stuck terms are still control-stuck when enclosed with a context that does not handle any effects. However, in order to understand the $\mathcal{S}[\![-]\!]$ relation fully, we need to turn our attention to the interpretation of effects and effect rows.

Denotation of effects is given in Figure 8. Effect rows, as well as single effects, are interpreted as tuples of five elements. The first two elements of such a quintuple are the expressions that raise the effect without any enclosing evaluation contexts. The next two elements are maps that describe

[6] A similar and related pattern appears, unsurprisingly, in constructions of bisimulations for calculi with control effects, see [Biernacki and Lenglet 2012] for an example.

[7] Consider trying to relate pure and effectful code at effect α: clearly, on the pure side there simply are no control-stuck terms to put in the relation, yet we are bound to pick *some* expressions to match properly control-stuck effectful terms; cf. Section 4.1 for a worked-out example.

$$(op_l\ v_1, op_l\ v_2, [l \mapsto 0], [l \mapsto 0], \triangleright[\![\tau_2]\!]_\emptyset) \in [\![l]\!] \iff$$
$$op : \tau_1 \to \tau_2 \in \Sigma(l) \wedge (v_1, v_2) \in \triangleright[\![\tau_1]\!]_\emptyset$$

$$[\![\langle\rangle]\!]_\eta \equiv \emptyset$$
$$[\![\alpha]\!]_\eta \equiv \eta(\alpha)$$
$$[\![\langle l \mid \varepsilon\rangle]\!]_\eta \equiv [\![l]\!] \cup [\![\varepsilon]\!]_\eta \uparrow l$$

Fig. 8. Interpretation of effects. We assume that the type is well-formed in implicit parameters Σ and Δ, and that $\eta \in \mathbf{Eff}^\Delta$ throughout.

how free the enclosing contexts should be. The final element is an *output relation* that specifies how the algebraic operation ought to be interpreted, i.e., what values or other possibly effectful expressions can replace the first two elements of the tuple.

For a single known effect, l, the first two elements of the quintuple are no surprise: an operation of the effect applied to related values of the argument type τ_1. Since the operations of the effect l can be interpreted by any handler that respects the typing rules, the output relation is the interpretation of type τ_2 (guarded by the later operator, to ensure well-formedness of the definition). The remaining components are two maps that detail which contexts would form control-stuck terms when plugged with the operation that appears in the first argument. The question that begs asking is why we should require this information: after all, we know which effect we do not want handled – the information seems to be present in the first two components of the tuple.

To understand why this information is useful, we must turn to the interpretation of effect rows, particularly the extension of a row ε by an effect l. Most of that definition is as expected: we take the sum of the interpretations of the effect and of the row – but the latter with a slight modification. The operation $Q \uparrow l$ shifts the maps of third and fourth element of the interpretation Q by one for the effect l and is formally defined as

$$(e_1, e_2, \rho_1 \uparrow l, \rho_2 \uparrow l, \mu) \in Q \uparrow l \iff (e_1, e_2, \rho_1, \rho_2, \mu) \in Q,$$

where the shifting of single map $\rho \uparrow l$ is defined as follows:

$$\begin{aligned} (\rho \uparrow l)(l) &= \rho(l) + 1 \\ (\rho \uparrow l)(l') &= \rho(l') \qquad \text{for } l \neq l'. \end{aligned}$$

This is precisely what makes for a sensible interpretation of multiple occurrences of the same effect in a row: we require that any effects that make use of l in the interpretation of ε be treated as *more free*, i.e., we acknowledge that neither of these occurrences are the first occurrence of l in the row, and thus should be considered unhandled even when enclosed with a handler. This, however, only accounts for a natural number being present, and the construct we use, a partial map from effect names to numbers is decidedly more complex.

The reason why we choose to use a more complex notion here is the presence of row polymorphism: as noted in the introduction, we want to be able to relate computations that instantiate universal quantifiers with *different* rows – in particular, pure and impure computations. Thus, we allow the relation to specify which effects ought to be considered pertinent. For the same reason, we also allow the expressions in the relation to be arbitrary, and not just operations applied to values – after all, pure computations cannot evaluate to control-stuck terms.

Finally, we can return to the way the interpretations of effect rows are used in the relation $\mathcal{S}[\![-]\!]$. There are three main components to the definition. First, we take expressions e_i that are

$$(\gamma_1, \gamma_2) \in \mathcal{G}[\![\Gamma]\!]_\eta \iff \mathrm{dom}(\gamma_1) = \mathrm{dom}(\gamma_2) = \mathrm{dom}(\Gamma) \wedge \forall x \in \mathrm{dom}(\Gamma).\ (\gamma_1(x), \gamma_2(x)) \in [\![\Gamma(x)]\!]_\eta$$

$$\Sigma; \Delta; \Gamma \models e_1 \precsim e_2 : \tau \ / \ \varepsilon \equiv \forall \eta \in \mathbf{Eff}^\Delta.\ \forall (\gamma_1, \gamma_2) \in \mathcal{G}[\![\Gamma]\!]_\eta.\ (e_1\gamma_1, e_2\gamma_2) \in \mathcal{E}[\![\tau \ / \ \varepsilon]\!]_\eta$$

$$\Sigma; \Delta; \Gamma \models e_1 \simeq e_2 : \tau \ / \ \varepsilon \equiv \Sigma; \Delta; \Gamma \models e_1 \precsim e_2 : \tau \ / \ \varepsilon \wedge \Sigma; \Delta; \Gamma \models e_2 \precsim e_1 : \tau \ / \ \varepsilon$$

Fig. 9. The logical relation.

in the interpretation of the effect row ε, along with associated maps ρ_i and the output relation μ. Then, we allow these expressions to be plugged into any evaluation contexts E_i, provided they are appropriately free with respect to the maps ρ_i: this ensures that the effect operations in e_i are not handled by the contexts and, crucially, that they will behave appropriately under context extension. Finally, we require that any pair of expressions that belongs to the output relation will be related as computation when plugged into the contexts E_i. We believe that this interpretation captures the essence of algebraic effects: the related effects can be interpreted in *any* way that is allowed by the output relation, as long as the context in which this happens is appropriately free (intuitively, does not handle the effect).

With all the components in place, we can define the notions of logical approximation and logical equivalence, as presented in Figure 9. The definition is standard: the logical equivalence is a logical approximation in both directions, while the logical approximation is a generalisation of relation $\mathcal{E}[\![-]\!]$ for open terms by closing them by any pointwise-related substitutions and any interpretation of row variables.

Note that as presented here, all definitions are mutually recursive in a complex way. We present them in this fashion to enhance clarity of reasoning. An interested reader may wish to consult the Coq development for a more fine-grained way of defining the appropriate relations.

3.3 Reasoning with the logical relation

Reasoning about programs using the definition of the logical relation directly is possible, but because of its biorthogonal nature, one needs to go through the relation $\mathcal{K}[\![-]\!]$ at every step of the process. In order to simplify the proofs we show that the relation $\mathcal{E}[\![-]\!]$ is preserved by backward reduction and includes relation $\mathcal{S}[\![-]\!]$ and the interpretation of types.

Lemma 1. *The following assertions hold:*
(1) $[\![\tau]\!]_\eta \subseteq \mathcal{E}[\![\tau \ / \ \varepsilon]\!]_\eta$;
(2) $\mathcal{S}[\![\tau \ / \ \varepsilon]\!]_\eta \subseteq \mathcal{E}[\![\tau \ / \ \varepsilon]\!]_\eta$;
(3) if $e_1 \to e_1'$ *then* $(e_1', e_2) \in \triangleright \mathcal{E}[\![\tau \ / \ \varepsilon]\!]_\eta$ *implies* $(e_1, e_2) \in \mathcal{E}[\![\tau \ / \ \varepsilon]\!]_\eta$;
(4) if $e_2 \to e_2'$ *then* $(e_1, e_2') \in \mathcal{E}[\![\tau \ / \ \varepsilon]\!]_\eta$ *implies* $(e_1, e_2) \in \mathcal{E}[\![\tau \ / \ \varepsilon]\!]_\eta$.

This lemma allows us to reduce expressions in the goal when proving program equivalences. Additionally, if a reduction takes place in the left-hand-side expression, we can remove an occurrence of the later operator from any assumption where it appears (see the later introduction rule in Section 3.1).

We also introduce an auxiliary relation for *partial* evaluation contexts, i.e., evaluation contexts that do not build a complete program. Two such contexts are related if they form related expressions when plugged with related values and related simple expressions.

$$(E_1, E_2) \in C[\![\tau_1 \ / \ \varepsilon_1 \rightsquigarrow \tau_2 \ / \ \varepsilon_2]\!]_\eta \iff (\forall (v_1, v_2) \in [\![\tau_1]\!]_\eta . (E_1[v_1], E_2[v_2]) \in \mathcal{E}[\![\tau_2 \ / \ \varepsilon_2]\!]_\eta) \wedge (\forall (e_1, e_2) \in \mathcal{S}[\![\tau_1 \ / \ \varepsilon_1]\!]_\eta . (E_1[e_1], E_2[e_2]) \in \mathcal{E}[\![\tau_2 \ / \ \varepsilon_2]\!]_\eta)$$

This relation is compatible with the relation $\mathcal{E}[\![-]\!]$.

Lemma 2. *For any contexts* $(E_1, E_2) \in C[\![\tau_1 \mathbin{/} \varepsilon_1 \rightsquigarrow \tau_2 \mathbin{/} \varepsilon_2]\!]_\eta$ *and expressions* $(e_1, e_2) \in \mathcal{E}[\![\tau_1 \mathbin{/} \varepsilon_1]\!]_\eta$ *we have* $(E_1[e_1], E_2[e_2]) \in \mathcal{E}[\![\tau_2 \mathbin{/} \varepsilon_2]\!]_\eta$.

This lemma is very useful when reasoning about programs with handlers: to prove that two handlers behave equivalently, it suffices to show that they are in the (symmetric closure of) relation $C[\![-]\!]$. Such a proof can be further simplified, since the relation $\mathcal{S}[\![-]\!]$ can be characterised by conditions that are much simpler to check. To show that two partial contexts are related, it suffices to check that they are related when plugged with values and those simple expressions that are *relevant* to these contexts. This characterisation is formalised through the following two lemmas.[8]

Lemma 3. *Let* ε_1 *be an effect row, and* ε_2 *be effect row* ε_1 *with* n *additional labels* l *at the front. Additionally, assume the following:*

(1) for any $(v_1, v_2) \in [\![\tau_1]\!]_\eta$ *we have* $(E_1[v_1], E_2[v_2]) \in \mathcal{E}[\![\tau_2 \mathbin{/} \varepsilon_2]\!]_\eta$*;*
(2) $n{-}\mathrm{free}(l, E_1)$ *and* $n{-}\mathrm{free}(l, E_2)$*;*
(3) $0{-}\mathrm{free}(l', E_1)$ *and* $0{-}\mathrm{free}(l', E_2)$ *for any* $l' \neq l$.

Then $(E_1, E_2) \in C[\![\tau_1 \mathbin{/} \varepsilon_1 \rightsquigarrow \tau_2 \mathbin{/} \varepsilon_2]\!]_\eta$.

Lemma 4. *Let* ε_1 *be an effect row, and* ε_2 *be an effect row* ε_1 *with* n *additional labels* l *at the front. Additionally, assume the following:*

(1) for any $(v_1, v_2) \in [\![\tau_1]\!]_\eta$ *we have* $(E_1[v_1], E_2[v_2]) \in \mathcal{E}[\![\tau_2 \mathbin{/} \varepsilon_2]\!]_\eta$*;*
(2) $n{-}\mathrm{free}(l, E_1[[\Box]_l])$ *and* $n{-}\mathrm{free}(l, E_2[[\Box]_l])$*;*
(3) $0{-}\mathrm{free}(l', E_1)$ *and* $0{-}\mathrm{free}(l', E_2)$ *for any* $l' \neq l$*;*
(4) for any $(op : \sigma_1 \to \sigma_2) \in \Sigma(l)$, $(v_1, v_2) \in \triangleright[\![\sigma_1]\!]_\eta$ *and any contexts* E_1' *and* E_2' *such that* $0{-}\mathrm{free}(l, E_1')$ *and* $0{-}\mathrm{free}(l, E_2')$*, if we have* $(E_1'[u_1], E_2'[u_2]) \in \triangleright\mathcal{E}[\![\tau_1 \mathbin{/} \langle l \mid \varepsilon_1 \rangle]\!]_\eta$ *for any values* $(u_1, u_2) \in \triangleright[\![\sigma_2]\!]_\eta$ *then we have* $(E_1[E_1'[op_l\ v_1]], E_2[E_2'[op_l\ v_2]]) \in \mathcal{E}[\![\tau_2 \mathbin{/} \varepsilon_2]\!]_\eta$.

Then $(E_1, E_2) \in C[\![\tau_1 \mathbin{/} \langle l \mid \varepsilon_1 \rangle \rightsquigarrow \tau_2 \mathbin{/} \varepsilon_2]\!]_\eta$.

Note that the fourth requirement of Lemma 4 is similar to checking that contexts E_1 and E_2 are compatible with elements of $\mathcal{S}[\![\tau_1 \mathbin{/} \langle l \mid \varepsilon_1 \rangle]\!]_\eta$, but one only needs to consider the effect l, since the other effects are not handled by these contexts.

With these tools ready, we are in a position to prove standard *compatibility lemmas* which state that the logical relation is preserved by all language constructs. We list the three lemmas that are explicitly related to algebraic effects.

Lemma 5 (compat-op). *If* $op : \sigma \to \tau \in \Sigma(l)$*, then* $\Sigma; \Delta; \Gamma \models op_l \precsim op_l : \sigma \to_{\langle l \rangle} \tau \mathbin{/} \langle\rangle$.

Lemma 6 (compat-lift). *If* $\Sigma; \Delta; \Gamma \models e_1 \precsim e_2 : \tau \mathbin{/} \varepsilon$*, then* $\Sigma; \Delta; \Gamma \models [e_1]_l \precsim [e_2]_l : \tau \mathbin{/} \langle l \mid \varepsilon \rangle$.

Lemma 7 (compat-handle). *Take any expressions* e_1, e_2, e_1', e_2' *and handlers* h_1, h_2 *such that:*

(1) $\Sigma; \Delta; \Gamma \models e_1 \precsim e_2 : \sigma \mathbin{/} \langle l \mid \varepsilon \rangle$,
(2) for each $(op : \tau_1 \to \tau_2) \in \Sigma(l)$ *there exist* $(op\ x, r.\ e_1^h \in h_1)$ *and* $(op\ x, r.\ e_2^h) \in h_2$ *such that* $\Sigma; \Delta; \Gamma, x : \tau_1, r : \tau_2 \to_\varepsilon \tau \models e_1^h \precsim e_2^h : \tau \mathbin{/} \varepsilon$,
(3) $\Sigma; \Delta; \Gamma, x : \sigma \models e_1' \precsim e_2' : \tau \mathbin{/} \varepsilon$.

Then $\Sigma; \Delta; \Gamma \models \mathsf{handle}_l\ e_1\ \{h_1; \mathsf{return}\ x.\ e_1'\} \precsim \mathsf{handle}_l\ e_2\ \{h_2; \mathsf{return}\ x.\ e_2'\} : \tau \mathbin{/} \varepsilon$.

[8] This notion can be further generalised to partial contexts that have any number of relevant effects. However, the general formulation is complex and requires additional definitions, so we present only the most useful instances. An interested reader may consult the Coq development for details.

As a corollary of the compatibility lemmas, we obtain the fundamental property of the logical relations, i.e., that all well-typed programs are related to themselves.

Theorem 8 (fundamental). *For any expression e, if $\Sigma; \Delta; \Gamma \vdash e : \tau \;/\; \varepsilon$, then $\Sigma; \Delta; \Gamma \models e \simeq e : \tau \;/\; \varepsilon$.*

3.4 Soundness

Having defined a relational model of $\lambda^{\mathsf{H/L}}$ we can use it for reasoning about programs in our calculus. First of all, as a corollary of the fundamental property we can prove type-soundness for complete programs:

Theorem 9 (type-soundness). *For any expression e, if $\Sigma; \cdot; \cdot \vdash e : 1 \;/\; \langle\rangle$ and $e \to^* e' \not\to$, then e' is a unit value ($e' = ()$).*

Another important property of the logical relation is that it is sound with respect to the contextual equivalence. Two programs are contextually equivalent if they have the same observations in any context:

$$\Sigma; \Delta; \Gamma \vdash e_1 \simeq e_2 : \tau \;/\; \varepsilon \equiv$$
$$\forall C \in \text{Cont}.\ \Sigma; \cdot; \cdot \vdash C[e_1], C[e_2] : 1 \;/\; \langle\rangle \Rightarrow (C[e_1] \to^* () \Leftrightarrow C[e_2] \to^* ())$$

Note that the only observation is the termination to unit value. Even though the unit type has exactly one value, this design choice does not decrease the discriminating power of contextual equivalence: since computations can diverge and we quantify over all possible contexts, any difference in observational behaviour of two programs can be used to build a context that makes exactly one of them diverge.

It is notoriously difficult to use this definition directly. However, the logical relation is specifically chosen to imply it, as in the following theorem, thus giving us a useful method of showing contextual equivalence.

Theorem 10 (soundness). *For any expressions e_1 and e_2, if $\Sigma; \Delta; \Gamma \models e_1 \simeq e_2 : \tau \;/\; \varepsilon$ holds for all step-indices, then $\Sigma; \Delta; \Gamma \vdash e_1 \simeq e_2 : \tau \;/\; \varepsilon$.*

4 REASONING ABOUT ALGEBRAIC EFFECTS AND HANDLERS

We now show how one can use the logical relation defined in Section 3 to prove non-trivial program equivalences. We discuss five examples: the first two illustrate how one can employ the parametricity of effect polymorphism in reasoning about algebraic effects and handlers, while in the others we formally investigate the expressive power of our lift construct. To help the reader understand the particularities of our logical relation, the first example is simpler and explained in greater detail. The other examples are more condensed, but they exhibit similar proof structure.

To increase readability, we use type *int* together with some literals – but no operations. For simplicity of presentation, the $\lambda^{\mathsf{H/L}}$ calculus does not contain base types or constants (see also Section 5.1), so the reader should treat *int* as syntactic sugar on the meta-level standing for any type with a number of distinct values.

4.1 Pure vs effectful computations

In the first example we return to the problem posed in the introduction, namely showing the contextual equivalence of two programs: a pure one and an effectful one interpreted by a handler, given by expressions (1) and (2) respectively. To state the problem using the definition of the contextual equivalence and the full syntax of $\lambda^{\mathsf{H/L}}$, let f be a row-polymorphic function of type $\forall\alpha.(1 \to_\alpha \mathit{int}) \to_\alpha \tau$ in an environment Γ. We want to prove the following equivalence:

$$\Sigma; \cdot; \Gamma \vdash f * (\lambda x.5) \simeq \mathsf{handle}_{\mathsf{R}}\ f * \mathit{ask}_{\mathsf{R}}\ \{\mathit{ask}\ \_, r.\ r\ 5\} : \tau \;/\; \langle\rangle$$

We can establish the above by using soundness (Theorem 10) together with the definition of the logical equivalence (Figure 9). Proving the logical equivalence amounts to verifying two approximations $\precsim$, but we show only the first one here, as the proof of the other one is analogous.

By definition of $\precsim$, it is enough to show that assuming $(f_1, f_2) \in [\![\forall\alpha.(1 \to_\alpha \mathit{int}) \to_\alpha \tau]\!]_\emptyset$, it is the case that

$$(f_1 * (\lambda x.5), \mathsf{handle}_\mathsf{R}\ f_2 * \mathit{ask}_\mathsf{R}\ \{\mathit{ask}\ \_, r.\ r\ 5\}) \in \mathcal{E}[\![\tau\ /\ \langle\rangle]\!]_\emptyset. \tag{3}$$

For this, we define a relation R in **Eff**, which relates only one quintuple:

$$R \quad \equiv \quad \{((\lambda x.5)\ (), \mathit{ask}_\mathsf{R}\ (), \emptyset, [\mathsf{R} \mapsto 0], \{(5,5)\})\}$$

Note that the pure expression $(\lambda x.5)\ ()$ related with $\mathit{ask}_\mathsf{R}\ ()$ is not a normal form and can be reduced. We pick this expression because it is the part of the snapshot of the execution of the first program that exactly corresponds to the point in the execution of the second program where the effect is raised. The third and fourth components of this quintuple are maps that describe the level of freeness of the evaluation contexts in which mentioned expressions may occur. Indeed, the first map is empty, because the first expression is pure, and the second expression uses the first reader effect in a row, so the context should be 0-free for the effect R. The last element of the quintuple is a relation that contains only one pair of values, $(5, 5)$, that is, the only possible interpretation of the considered effect in the compared programs. We also need the following three simple facts:

(A) $(f_1\ *, f_2\ *) \in \mathcal{E}[\![(1 \to_\alpha \mathit{int}) \to_\alpha \tau\ /\ \langle\rangle]\!]_{[\alpha\mapsto R]}$.
This fact actually holds for any relation R, and follows directly from the definition of the interpretation of polymorphic types.

(B) $(\lambda x.5, \mathit{ask}_\mathsf{R}) \in [\![1 \to_\alpha \mathit{int}]\!]_{[\alpha\mapsto R]}$.
Since the relation $\mathcal{S}[\![-]\!]$ is included in $\mathcal{E}[\![-]\!]$, it suffices to show that $(\Box[(\lambda x.5)\ ()], \Box[\mathit{ask}_\mathsf{R}\ ()]) \in \mathcal{S}[\![\mathit{int}\ /\ \alpha]\!]_{[\alpha\mapsto R]}$. The quintuple in R relates $(\lambda x.5)\ ()$ with $\mathit{ask}_\mathsf{R}\ ()$, so, by the definition of $\mathcal{S}[\![-]\!]$, it is enough to show the following facts about the empty context:
  (i) $\emptyset$−free($\Box$),
  (ii) 0−free(R, $\Box$),
  (iii) for any $(e_1, e_2) \in \{(5, 5)\}$, it is the case that $(\Box[e_1], \Box[e_2]) \in \triangleright\mathcal{E}[\![\mathit{int}\ /\ \alpha]\!]_{[\alpha\mapsto R]}$.
  These facts are easy to verify. Note that in this example the results of $\lambda x.5$ and ask_R may be passed to some unknown code, so we have to make sure that they are equivalent. Since we consider these functions effectful, in (iii) we only check that the results given in the specification (that is, the relation R) are equivalent. For an incorrect specification, e.g., when the last element of the quintuple is $\{(5, 5), (7, 42)\}$, we might be unable to prove (iii).

(C) $(\Box, \mathsf{handle}_\mathsf{R}\ \Box\ \{\mathit{ask}\ \_, r.\ r\ 5\}) \in C[\![\tau\ /\ \alpha \rightsquigarrow \tau\ /\ \langle\rangle]\!]_{[\alpha\mapsto R]}$.
We prove this fact using Löb induction. First, we assume that these contexts are in the relation $\triangleright C[\![\tau\ /\ \alpha \rightsquigarrow \tau\ /\ \langle\rangle]\!]_{[\alpha\mapsto R]}$. By the definition of $C[\![-]\!]$, we need to show that these contexts form related expressions when plugged with related values or related simple expressions. The case of values is trivial, since both contexts just return the respective plugged values. For the case of simple expressions, we show that for any $(e_1, e_2) \in \mathcal{S}[\![\tau\ /\ \alpha]\!]_{[\alpha\mapsto R]}$, it is the case that $(e_1, \mathsf{handle}_\mathsf{R}\ e_2\ \{\mathit{ask}\ \_, r.\ r\ 5\}) \in \mathcal{E}[\![\tau\ /\ \langle\rangle]\!]_{[\alpha\mapsto R]}$. Since there is only one quintuple in the interpretation of α, we need to show that for any E_1 and E_2 such that 0−free(R, E_2) $(*)$ and $(E_1[5], E_2[5]) \in \triangleright\mathcal{E}[\![\tau\ /\ \alpha]\!]_{[\alpha\mapsto R]}$ $(**)$, it is the case that

$$(E_1[(\lambda x.5)\ ()], \mathsf{handle}_\mathsf{R}\ E_2[\mathit{ask}_\mathsf{R}\ ()]\ \{\mathit{ask}\ \_, r.\ r\ 5\}) \in \mathcal{E}[\![\tau\ /\ \langle\rangle]\!]_{[\alpha\mapsto R]}.$$

Because of $(*)$, the handler matches the ask_R operation and both expressions can be reduced. So, it is left to show that

$$(E_1[5], \mathsf{handle}_\mathsf{R}\ E_2[5]\ \{\mathit{ask}\ \_, r.\ r\ 5\}) \in \triangleright\mathcal{E}[\![\tau\ /\ \langle\rangle]\!]_{[\alpha\mapsto R]}.$$

Note that in the above we performed reduction on the left-hand side, so the later operator appeared in our goal. We shift reasoning to the future world, removing the later operator from the induction hypothesis, (∗∗), and our goal. Then we conclude using the induction hypothesis and (∗∗). Note that in this part of the proof we check that the handler matches the specification given in our relation R. If the last component of the quintuple did not contain the pair $(5, 5)$, we would not be able to use (∗∗) to conclude the proof.

To show (3), we can now use Lemma 2 to obtain the following:

$$(f_1 * (\lambda x.5), \mathsf{handle}_\mathsf{R}\ f_2 * ask_\mathsf{R}\ \{ask\ \_, r.\ r\ 5\}) \in \mathcal{E}[\![\tau\ /\ \langle\rangle]\!]_{[\alpha \mapsto R]}.$$

The first premise of Lemma 2 is instantiated with (C), while the second premise is instantiated with the compatibility of pointwise application of (A) to (B). Now, α occurs neither in τ nor $\langle\rangle$, hence, using weakening, we can exchange the row variable interpretation $[\alpha \mapsto R]$ to the empty map $\emptyset$, which concludes the proof of (3).

4.2 State as a composition of reader and writer

In this example we show that the representation of state using the reader and writer effects mentioned in the introduction is equivalent to the standard handler for state. We assume that Σ contains the following effects:

- reader effect R with one operation $ask : 1 \to \sigma$,
- writer effect W with one operation $tell : \sigma \to 1$,
- state effect S with two operations $get : 1 \to \sigma$ and $put : \sigma \to 1$.

Given a polymorphic function f of type $\forall\alpha.(1 \to_\alpha \sigma) \to (\sigma \to_\alpha 1) \to_\alpha \tau$ in an environment Γ, we can show that the following two functions are equivalent for type $\sigma \to \tau$ and the empty effect row:

$$\begin{array}{l}\lambda s.\mathsf{handle}_\mathsf{R} \\ \quad \mathsf{handle}_\mathsf{W}\ f * ask_\mathsf{R}\ tell_\mathsf{W} \\ \quad \{tell\ s', r.\ [\mathsf{handle}_R\ r\ ()\ \{ask\ \_, r.\ r\ s'\}]_\mathsf{R}\} \\ \{ask\ \_, r.\ r\ s\}\end{array} \qquad \begin{array}{l}\lambda s.\mathsf{handle}_\mathsf{S}\ f * get_\mathsf{S}\ put_\mathsf{S} \\ \quad \{get\ \_, r.\ \lambda s.r\ s\ s \\ \quad ; put\ s', r.\ \lambda s.r\ ()\ s' \\ \quad ; \mathsf{return}\ x.\ \lambda\_.x \\ \quad \}\ s\end{array}$$

Note that equivalence of the two expressions above is not the ultimate equivalence of the two implementations of state: indeed, we shall see that because of the limiting type that we picked for f, the lift in the reader/writer definition will never be actively used. A more useful equivalence would be obtained through a translation of one effect into another; however, we believe the argument in the following presents the proof method well, while at the same time being easier to follow.

As previously, we show the approximation only in one direction. We start with an instantiation for the free variable f in both programs with any $(f_1, f_2) \in [\![\forall\alpha.(1 \to_\alpha \sigma) \to (\sigma \to_\alpha 1) \to_\alpha \tau]\!]_\emptyset$, and $(s_1, s_2) \in [\![\sigma]\!]_\emptyset$ for arguments of the λ-abstractions. We pick the following interpretation of α:

$$\begin{array}{rcl} R & \equiv & \{(ask_\mathsf{R}\ (), get_\mathsf{S}\ (), [\mathsf{R} \mapsto 0], [\mathsf{S} \mapsto 0], [\![\sigma]\!]_\emptyset)\} \\ & \cup & \{(tell_\mathsf{W}\ v_1, put_\mathsf{S}\ v_2, [\mathsf{W} \mapsto 0], [\mathsf{S} \mapsto 0], \{((), ())\}) \mid (v_1, v_2) \in [\![\sigma]\!]_\emptyset\} \end{array}$$

In R we say that $ask_\mathsf{R}\ ()$ is related with $get_\mathsf{S}\ ()$, but we do not specify how they are interpreted by a handler. We only require that they are interpreted as equivalent elements of type σ (the last element of the first quintuple is the interpretation of σ). Effectful expressions $tell_\mathsf{W}\ v_1$ and $put_\mathsf{S}\ v_2$ are considered equivalent if the new state values v_1 and v_2 are equivalent. Similarly to the previous example, we use the following three facts:

(A) $(f_1 *, f_2 *) \in \mathcal{E}[\![(1 \to_\alpha \sigma) \to (\sigma \to_\alpha 1) \to_\alpha \tau\ /\ \langle\rangle]\!]_{[\alpha \mapsto R]}$.
(B) $(ask_\mathsf{R}, get_\mathsf{S}) \in [\![1 \to_\alpha \sigma]\!]_{[\alpha \mapsto R]}$ and $(tell_\mathsf{W}, put_\mathsf{S}) \in [\![\sigma \to_\alpha 1]\!]_{[\alpha \mapsto R]}$.

(C) Contexts

$$\begin{array}{l}\mathsf{handle}_{\mathsf{R}}\\ \quad \mathsf{handle}_{\mathsf{W}}\ \Box\\ \quad \{\mathit{tell}\ s', r.\ [\mathsf{handle}_R\ r\ ()\ \{\mathit{ask}\ \_, r.\ r\ s'\}]_{\mathsf{R}}\}\\ \{\mathit{ask}\ \_, r.\ r\ s_1\}\end{array} \quad \text{and} \quad \begin{array}{l}\mathsf{handle}_{\mathsf{S}}\ \Box\\ \{\mathit{get}\ \_, r.\ \lambda s.r\ s\ s\\ ; \mathit{put}\ s', r.\ \lambda s.r\ ()\ s'\\ ; \mathsf{return}\ x.\ \lambda\_.x\\ \}\ s_2\end{array}$$

are in the relation $C[\![\tau\ /\ \alpha \rightsquigarrow \tau\ /\ \langle\rangle]\!]_{[\alpha\mapsto R]}$.
In order to show this fact, we first generalise it for any $(s_1, s_2) \in [\![\sigma]\!]_\emptyset$ (because the state may change during the computation), and then we prove it by Löb induction. Both contexts return plugged values, so we only need to test them against simple expressions. Since the relation R is a sum of two relations, we have to check two cases: for reader and writer respectively. For the reader effect, we test contexts using expressions $E_1[\mathit{ask}_{\mathsf{R}}\ ()]$ and $E_2[\mathit{get}_{\mathsf{S}}\ ()]$ such that $0{-}\mathrm{free}(\mathsf{R}, E_1)$ and $0{-}\mathrm{free}(\mathsf{S}, E_2)$. In this case we proceed as in the previous example: contexts plugged with these expressions after a number of reduction steps recreate themselves plugged with equivalent expressions $E_1[s_1]$ and $E_2[s_2]$, so we conclude using the induction hypothesis (we performed reduction steps, so we can remove the later operator). For the writer effect, the situation is more complicated, since after a number of reduction steps we obtain two terms of the forms $\mathsf{handle}_{\mathsf{R}}\ [e_1]_{\mathsf{R}}\ \{\mathit{ask}\ \_, r.\ r\ s_1\}$ and e_2 respectively, where e_1 and e_2 are tested contexts (with a new state) plugged with some equivalent expressions. However, because we can show that e_1 and e_2 are related in $\mathcal{E}[\![\tau\ /\ \langle\rangle]\!]_{[\alpha\mapsto R]}$ (using the induction hypothesis), it suffices to show that

$$(\mathsf{handle}_{\mathsf{R}}\ [\Box]_{\mathsf{R}}\ \{\mathit{ask}\ \_, r.\ r\ s_1\}, \Box) \in C[\![\tau\ /\ \langle\rangle \rightsquigarrow \tau\ /\ \langle\rangle]\!]_{[\alpha\mapsto R]}.$$

This, in turn, is easy to verify: because the relation $\mathcal{S}[\![\tau\ /\ \langle\rangle]\!]_{[\alpha\mapsto R]}$ is empty, we only have to test these contexts against values.

The rest of the proof is obtained by combining facts (A), (B), and (C) using compatibility and weakening lemmas as in the previous example.

4.3 Return clause

In this example we show that one consequence of having the lift operator in the language is that we do not need the return clause in the grammar of handlers. That is, each handler with a non-trivial return clause can be expressed as a handler in which the return clause is trivial.

Assume expressions e and e_r, and a handler h such that the following hold:

- $\Sigma; \Delta; \Gamma \vdash e : \sigma\ /\ \langle l \mid \varepsilon\rangle$,
- $\Sigma; \Delta; \Gamma \vdash_l h : \tau\ /\ \varepsilon$,
- $\Sigma; \Delta; \Gamma, x : \sigma \vdash e_r : \tau\ /\ \varepsilon$.

Then, as we now prove, the expressions $\mathsf{handle}_l\ e\ \{h; \mathsf{return}\ x.\ e_r\}$ and $\mathsf{handle}_l\ (\lambda x.[e_r]_l)\ e\ \{h\}$ are equivalent.

As usual, we show the approximation only in one direction. By the soundness theorem, it is enough to show that for any η and $(\gamma_1, \gamma_2) \in \mathcal{G}[\![\Gamma]\!]_\eta$, we have the following:

$$(\mathsf{handle}_l\ e\gamma_1\ \{h\gamma_1; \mathsf{return}\ x.\ e_r\gamma_1\}, \mathsf{handle}_l\ (\lambda x.[e_r\gamma_2]_l)\ e\gamma_2\ \{h\gamma_2\}) \in \mathcal{E}[\![\tau\ /\ \varepsilon]\!]_\eta.$$

Since $(e\gamma_1, e\gamma_2) \in \mathcal{E}[\![\sigma\ /\ \langle l \mid \varepsilon\rangle]\!]_\eta$ (by the fundamental property), it suffices to show that the evaluation contexts enclosing the two expressions are related:

$$(\mathsf{handle}_l\ \Box\ \{h\gamma_1; \mathsf{return}\ x.\ e_r\gamma_1\}, \mathsf{handle}_l\ (\lambda x.[e_r\gamma_2]_l)\ \Box\ \{h\gamma_2\}) \in C[\![\sigma\ /\ \langle l \mid \varepsilon\rangle \rightsquigarrow \tau\ /\ \varepsilon]\!]_\eta. \quad (4)$$

We proceed by Löb induction. By Lemma 4 it is sufficient to test these contexts against values and the single relevant effect l.

When we plug related values $(v_1, v_2) \in [\![\sigma]\!]_\eta$ into these contexts, after one reduction step we obtain the terms $e_r\gamma_1\{v_1/x\}$ and $\mathsf{handle}_l\ [e_r\gamma_2\{v_2/x\}]_l\ \{\gamma_2 h\}$. By the fundamental property, we have $(e_r\gamma_1\{v_1/x\}, e_r\gamma_2\{v_2/x\}) \in \mathcal{E}[\![\tau\ /\ \varepsilon]\!]_\eta$, so it suffices to show that

$$(\Box, \mathsf{handle}_l\ [\Box]_l\ \{h\gamma_2\}) \in C[\![\tau\ /\ \varepsilon \rightsquigarrow \tau\ /\ \varepsilon]\!]_\eta,$$

which in turn is easy to verify using Lemma 3.

Now, we test contexts from (4) with l. Let us take any $op : \tau_1 \to \tau_2 \in \Sigma(l)$ and any values v_1, v_2 together with contexts E_1, E_2 such that:

(A) $(v_1, v_2) \in \triangleright[\![\tau_1]\!]_\emptyset$,
(B) $0{-}\mathrm{free}(l, E_1)$,
(C) $0{-}\mathrm{free}(l, E_2)$,
(D) for any $(u_1, u_2) \in \triangleright[\![\tau_2]\!]_\emptyset$, we have $(E_1[u_1], E_2[u_2]) \in \triangleright\mathcal{E}[\![\sigma\ /\ \langle l \mid \varepsilon\rangle]\!]_\eta$.

We need to show

$$(\mathsf{handle}_l\ E_1[op_l\ v_1]\ \{h\gamma_1; \mathsf{return}\ x.\ e_r\gamma_1\}, \mathsf{handle}_l\ (\lambda x.[e_r\gamma_2]_l)\ (E_2[op_l\ v_2])\ \{h\gamma_2\}) \in \mathcal{E}[\![\tau\ /\ \varepsilon]\!]_\eta.$$

But the handler h is well-typed, so there exists an expression e_h, such that $(op\ x, r.\ e_h) \in h$ and $\Sigma; \Delta; \Gamma, x : \tau_1, r : \tau_2 \to_\varepsilon \tau \vdash e_h : \tau\ /\ \varepsilon$, so we can reduce the tested expressions and show that

$$(e_h\gamma_1\{v_1/x\}\{r_1/r\}, e_h\gamma_2\{v_2/x\}\{r_2/r\}) \in \mathcal{E}[\![\tau\ /\ \varepsilon]\!]_\eta,$$

where r_1 and r_2 are reified continuations:

$$\begin{aligned} r_1 &= \lambda z.\mathsf{handle}_l\ E_1[z]\ \{h\gamma_1; \mathsf{return}\ x.\ e_r\gamma_1\} \\ r_2 &= \lambda z.\mathsf{handle}_l\ (\lambda x.[e_r\gamma_2]_l)\ (E_2[z])\ \{h\gamma_2\}. \end{aligned}$$

Note that this reduction allows us to remove later operators from (A), (D), and the induction hypothesis. Using the fundamental property, we can conclude the proof if we show that $(r_1, r_2) \in [\![\tau_2 \to_\varepsilon \tau]\!]_\eta$. This, in turn, can be obtained using (D), the induction hypothesis, and the compatibility lemmas.

4.4 Generalised lift

The lift construct $[-]_l$ in $\lambda^{\mathsf{H/L}}$ allows one to explicitly extend any effect row ε by a dummy effect l, yielding a row $\langle l \mid \varepsilon\rangle$. This new effect l is interpreted as the first effect in the row, and is handled by the first corresponding handler. We could generalise this construct in a way that inserts a new effect at any position in the row. For example, we can have a construct $[-]^1_l$ with the following typing rule:

$$\frac{\Sigma; \Delta; \Gamma \vdash e : \tau\ /\ \langle l \mid \varepsilon\rangle}{\Sigma; \Delta; \Gamma \vdash [e]^1_l : \tau\ /\ \langle l, l \mid \varepsilon\rangle}$$

which places the dummy effect at the second position. Then $[op_l\ v]^1_l$ and $[[op_l\ v]_l]^1_l$ would be handled by the first and the third corresponding handler respectively.

Interestingly, $\lambda^{\mathsf{H/L}}$ is expressive enough to define such a generalised lift, but the construction is not as straightforward as one might expect. For instance, assume an effect l with a single operation op_l. Then the construct $[e]^1_l$ can be macro-expressed as follows:

$$\begin{aligned} [e]^1_l \equiv\ & [\,[\ \mathsf{handle}_l\ e \\ & \quad \{op\ x, r.\ \lambda\_.(\lambda y.[[r\ y]_l]_l\ ())\ (op_l\ x) \\ & \quad ; \mathsf{return}\ x.\ \lambda\_.x \\ & \quad \} \\ &]_l]_l\ (). \end{aligned}$$

The high-level idea of the solution is to handle the first effect in a row, and create a thunk with the desired effect (the effect of the suspended computation inside a λ-abstraction can be different than the effect of the λ-abstraction itself). However, the process of creating a thunk might raise a latent effect from ε – including yet another instance of l, which needs to be appropriately guarded. Since $[e]^1_l$ has to have the effect $\langle l, l \mid \varepsilon\rangle$, the handler must be lifted twice before the thunk is forced. A similar story happens inside the handler for op_l, because the resumption r has the effect ε and returns a thunk with the desired effect.

In order to see that $[e]^1_l$ has the desired behaviour, we show that handling and then lifting the effect l is equivalent to lifting using $[-]^1_l$ and then handling. Assume we have $\Sigma(l) = op_l : \tau_1 \to \tau_2$. Then for any value f of type $\tau_1 \to (\tau_2 \to_{\langle l|\varepsilon\rangle} \tau) \to_\varepsilon \tau$ and expression e with type τ and effect $\langle l \mid \varepsilon\rangle$ we have

$$([\mathsf{handle}_l\ e\ \{op_l\ x, r.\ f\ x\ (\lambda y.[r\ y]_l)\}]_l, \mathsf{handle}_l\ [e]^1_l\ \{op_l\ x, r.\ [f\ x\ r]_l\}) \in \mathcal{E}[\![\tau\ /\ \langle l \mid \varepsilon\rangle]\!]_\eta.$$

We show using Löb induction principle that contexts $[\mathsf{handle}_l\ \square\ \{op_l\ x, r.\ f\ x\ (\lambda y.[r\ y]_l)\}]_l$ and $\mathsf{handle}_l\ [\square]^1_l\ \{op_l\ x, r.\ [f\ x\ r]_l\}$ enclosing expression e are in relation $C[\![\tau\ /\ \langle l \mid \varepsilon\rangle \rightsquigarrow \tau\ /\ \langle l \mid \varepsilon\rangle]\!]_\eta$. By Lemma 4 it suffices to check only effect l (checking values in this case is trivial). Let us test these contexts with any control-stuck expressions $E_1[op_l\ v_1]$ and $E_2[op_l\ v_2]$ such that

(A) $(v_1, v_2) \in \triangleright[\![\tau_1]\!]_\emptyset$,
(B) $0{-}\mathrm{free}(l, E_1)$ and $0{-}\mathrm{free}(l, E_2)$,
(C) for every $(u_1, u_2) \in \triangleright[\![\tau_2]\!]_\emptyset$ we have $(E_1[u_1], E_2[u_2]) \in \triangleright\mathcal{E}[\![\tau\ /\ \langle l \mid \varepsilon\rangle]\!]_\eta$.

The first program reduces to $[f\ v_1\ (\lambda y.[r_1\ y]_l)]_l$ where r_1 is a resumption

$$r_1 \quad = \quad \lambda z.\mathsf{handle}_l\ E_1[z]\ \{op_l\ x, r.\ f\ x\ (\lambda y.[r\ y]_l)\}.$$

The reduction of the second program is more complicated. At first, the effect is handled inside the definition of $[-]^1_l$, where the thunk is returned and immediately forced. So after four reduction steps we get

$$\mathsf{handle}_l\ (\lambda y.s_y)\ (op_l\ v_2)\ \{op_l\ x, r.\ [f\ x\ r]_l\},$$

where s_y is an expression that in one step reduces to $[E_2[y]]^1_l$. Now, we have an operation op_l, which is matched by the outside handler, so we can reduce further, obtaining the term $[f\ v_2\ r_2]_l$, where r_2 is the following resumption:

$$r_2 \quad = \quad \lambda z.\mathsf{handle}_l\ (\lambda y.s_y)\ z\ \{op_l\ x, r.\ [f\ x\ r]_l\}.$$

Reducing both programs, we obtain terms of the same form, so, by compatibility, we only need to show the equivalence of the fragments that differ, that is, that for any $(u_1, u_2) \in [\![\tau_2]\!]_\eta$, we want $((\lambda y.[r_1\ y]_l)\ u_1, r_2\ u_2) \in \mathcal{E}[\![\tau\ /\ \langle l \mid \varepsilon\rangle]\!]_\eta$. But these expressions reduce to the initial contexts plugged with $E_1[u_1]$ and $E_2[u_2]$ respectively, so we can conclude using the induction hypothesis and the property of the contexts E_1 and E_2.

4.5 Swapping effects in a row

Handlers in $\lambda^{\mathsf{H/L}}$ always handle the first occurrence of a given effect in a row. This also leaves a place for potential generalisation. We can imagine a handle-like construct that handles the second or any other occurrence of the effect in the row. Such handlers are easily expressible in $\lambda^{\mathsf{H/L}}$ by means of an operation that permutes occurrences of the same effect in a row. In detail, $\mathsf{handle}^1_l\ e\ \{h\}$ that handles the second occurrence of effect l in a row can be macro-expressed as

$$\mathsf{handle}_l\ (\mathsf{swap}_l\ e)\ \{h\},$$

where $\mathsf{swap}_l\ e$ swaps first two occurrences of effect l in expression e. Defining this swap_l operation can be achieved using the same method as we used in the definition of $[-]^1_l$, that is, by building a thunk with the transformed effect. But in this case we need $[-]^1_l$ instead of the usual lift to coerce unhandled effects:

$$\begin{array}{lcl} \mathsf{swap}_l\ e & \equiv & [\ \mathsf{handle}_l\ e \\ & & \quad \{op_l\ x, r.\ \lambda\_.(\lambda y.[r\ y]^1_l\ ())\ [op_l\ x]_l \\ & & \quad ; \mathsf{return}\ x.\ \lambda\_.x \\ & & \quad \} \\ & &]^1_l\ (). \end{array}$$

We could show a property for swap analogous to the one discussed Section 4.4 with a similar proof based mostly on term reductions.

5 RELATED WORK

5.1 Calculi of algebraic effects and handlers

Different calculi for algebraic effects and handlers discussed in the literature are tailored for different purposes. They are either elaborated from A to Z in order to present a formal semantics of a fully-fledged programming language [Bauer and Pretnar 2015; Leijen 2017b; Lindley et al. 2017], or are trimmed down to focus on a specific aspect of programming with handlers [Bauer and Pretnar 2014; Hillerström et al. 2017; Kammar and Pretnar 2017]. Our case is the latter, since we focus on what we believe is a minimal calculus with a row-based type-and-effect system that supports well-behaved row polymorphism with multiple occurrences of the same effect in a row. Our calculus misses some elements indispensable in a full programming language, such as the usual base types other than the unit (booleans, integers, etc.), type polymorphism, or, perhaps, kinds. However, we suspect that adding these to the calculus would not pose much difficulty, since they seem orthogonal to the problem of relational interpretation of algebraic effects, and could be obtained using the standard relational machinery. Thus, to focus the presentation, they are intentionally omitted.

Managing multiple occurrences of the same effect, which we touch upon when considering row polymorphism (see Example 2.4), is an important practical problem, not necessarily related to the row-based approach to effects. For example, if one wants to use two different mutable memory cells, how to distinguish between the two different sets of *put*'s and *get*'s? The simplest approach would be to use two distinct effects, one for each cell. But this way we would forfeit modularity, since one would not be able to write a general handler for both effects. This issue is resolved, for example, in the language Eff [Bauer and Pretnar 2015] by effect instances or in Brady's [2013b] library by optional labels. In our calculus, the lift construct allows us to assign an operation to a different occurrence of the effect in the row, and the ordering of handlers allows us to handle an appropriate occurrence of the effect. While we acknowledge that some of these techniques may be less brittle in practical use, the semantics and models of such features are invariably more complex than what we propose in this paper, and their use would distract from the important theoretical properties we discuss here. We believe that the precise relative expressive power of dynamic effect instances, similar constructs in other languages, and our lift operator, as well as their interaction with other programming language constructs, such as existential types, forms an interesting avenue of future work.

Up to our knowledge, with the notable exception of Koka, all the calculi of algebraic effects and handlers described in the literature are based on Levy's [2004] call-by-push-value paradigm, which keeps an explicit distinction between value types and computation types, or on fine-grained call-by-value, which exhibits similar properties. It is a matter of future work to examine the relationship between call-by-push-value and call-by-value calculi with algebraic effects.

5.2 Reasoning about algebraic effects

Algebraic operations were first considered together with equations (see [Plotkin and Power 2004]) as a tool for algebraic understanding of monads used in programming, along the line of Moggi's [1991] category-theoretic semantics for the computational λ-calculus. The slogan 'notions of computation determine monads' [Plotkin and Power 2002] refers to the correspondence between algebraic theories and Lawvere theories on one side, and monads with rank on the other side; see [Hyland and Power 2007].

Reasoning about algebraic operations (without handlers) is discussed by Katsumata [2013] in the context of monadic lifting of logical relations, and by Plotkin and Power [2001a] in the context of relating denotational and operational semantics of languages with algebraic operations. Plotkin and Pretnar [Plotkin and Pretnar 2008] introduce a program logic to reason about equality in the computational λ-calculus with algebraic operations, later extended to incorporate handlers by Pretnar [2010].

Another approach to reasoning about algebraic operations and handlers is via dependent types. For a more practical account, see Brady's [2013b; 2014] implementation in the language Idris [Brady 2013a]. A more theoretical approach is given via fibred effects [Ahman 2017; Ahman et al. 2016].

5.3 Monadic interpretation

A part of the motivation for our approach to multiple occurrences of the same effect in a row comes from the monadic interpretation of algebraic effects [Plotkin and Power 2001b, 2002], used, for example, to embed algebraic effects in a pure language, such as Haskell [Kiselyov et al. 2013; Wu and Schrijvers 2015]. In this setting, operations are given in the continuation-passing style (see [Lindley 2014] for a discussion), that is, each operation is specified by the type of its parameter and a generalised *arity* given by an endofunctor. For example, a direct-style operation $ask_{\mathsf{R}} : 1 \to_{\mathsf{R}} \sigma$ corresponds to a continuation-passing operation with the arity given by the representable endofunctor $(\text{-})^{\sigma}$. A *signature* of an effect is an endofunctor obtained as a coproduct of operation signatures paired with the types of their respective parameters. Then, an effectful computation lives in the Kleisli category of the free monad induced by a signature, along the lines of Moggi's [1991] computational λ-calculus. In this setting, a multi-effect computation can be seen as a computation over the signature obtained as the coproduct of the respective signatures. Heavily abusing notation, one can say that a computation with the effect type given by a closed row $\langle l_0, l_1, l_2, \ldots \rangle$ correspond to a value of the free monad generated by the coproduct of signatures $l_0 + l_1 + l_2 + \cdots$.

Coproducts, at least in the category of (endofunctors on the category of) sets and functions, are given by (pointwise) disjoint unions, which means that their elements are labelled by the position of their signatures in the coproduct. (One usually writes *inl* and *inr* for the two labels in binary coproducts.) For instance, given an element of a coproduct of the shape $A + A$, the label reveals from which A the element comes from. In the syntax of $\lambda^{\mathsf{H/L}}$ and similar 'direct-style' calculi, there are no labels, and so, by default, each operation is assigned to the first occurrence of the corresponding effect in the coproduct, that is, the row. Additionally, in $\lambda^{\mathsf{H/L}}$ there is no label to denote a variable at the bottom of the free monad structure. So, given a signature Σ generating a free monad Σ^*, values of the free monad $\Sigma^*(\Sigma^* A)$, in which the variables are of the type $\Sigma^* A$, are treated in the same way as values of the type $\Sigma^* A$, since we cannot see where a variable 'begins' in the structure. In other words, for the lack of labels, some things that are easily distinguishable in the monadic approach are glued together in $\lambda^{\mathsf{H/L}}$. This issues can be addressed using the lift operator, which pushes occurrences of effects to the right in the row, making it possible to assign an operation to an arbitrary occurrence of the effect in the row.

The lift operator also allows us to represent $\Sigma^*(\Sigma^* A)$ as $(\Sigma + \Sigma)^* A$, useful in the following. In the monadic setting, handlers can be explained as transpositions of morphisms via the monadic free-forgetful adjunction $F \dashv U$ to the category of algebras for the signature endofunctors. That is, given a signature Σ and a Σ-algebra $\langle B, b : \Sigma B \to B \rangle$, a morphism $f : A \to U\langle B, b \rangle$ corresponds to an algebra homomorphism $\Sigma^* A \to B$. This homomorphism can be factored as $\Sigma^* A \xrightarrow{\Sigma^* f} \Sigma^* B \xrightarrow{\epsilon_{\langle B, b \rangle}} B$, where ϵ is the counit of the adjunction, and f represents the return clause. This is what happens in the example discussed in Section 4.3, in which we split the handler into two independent parts. However, if B is again an effectful computation, we do not want this handler to interpret the operations in B. This is what happens automatically in the monadic setting, as a label isolates B, while in Section 4.3 we have to explicitly mark it using the lift operator.

5.4 Row polymorphism for effect handlers

As we already remarked, in the recent years row-polymorphic systems have been used in the contexts of algebraic effects. In the following, we investigate the approaches adopted by the two most prominent exponents of this technique, Leijen's Koka [2017b], and Hillerström and Lindley's extension of Links [2016], through the lens of our relational interpretation of effects and effect rows. First, however, let us highlight their respective approaches.

The approach of Hillerström and Lindley is influenced by the fact that they extend an existing language, Links, and thus already have available row polymorphism, used for polymorphic variants, records and so on. This choice, however, ties their approach to the form of row quantification peculiar to the domain of records and variants: in keeping with the seminal work of Remy [1994], the names have to be unique within the row, and the universal quantifiers, constrained. In effect, the type $\forall \alpha.\ 1 \to_{\langle l | \alpha \rangle} 1$ is not well-formed: the system needs to ensure l will not be present in the row substituted for α, and the elimination rule for the polymorphic type must enforce this, giving us instead the type $\forall \alpha \notin \{l\}.\ 1 \to_{\langle l | \alpha \rangle} 1$.[9] In the context of records or variants, this kind of quantification is perfectly natural. However, as we argued in the preceding paragraph it is just as natural to allow multiple instances of the same effect in the row – indeed, as Example 2.4 argues, it can be positively beneficial. On the flip side, even though the quantification is more complex, the simplified shape of the rows can make the system easier to understand intuitively for the programmer.

The question remains, would an interpretation of effect rows defined along the lines of Section 3 also become simpler. We posit that this is indeed the case. Recall the interpretation of rows in Figure 8. If the type system ensures that effects in any row are disjoint, we can do away with the lifting of the interpretation of ε at l in the interpretation of $\langle l \mid \varepsilon \rangle$, and indeed with the maps themselves. Thus, in effect, we can interpret the row as a simple-minded set, as we have a static guarantee that every pair of effects in the row is disjoint. Crucially, this property is preserved under type substitution, due to the constrained universal quantification, which ensures soundness of this simplified approach. The lack of duplicate effects naturally renders lifts unnecessary, but this has little bearing for the remainder of the model, except for simpler definition of freeness. Thus, we can indeed view the system of Hillerström and Lindley as trading some expressive power for greater simplicity – a judgement justified by the natural simplifications in the model.

This is not so in the case of Leijen's calculus. The type system he presents does allow for multiple instances of the same effect label in the row, with the explicit proviso that $\langle l, l \rangle$ is different from $\langle l \rangle$, which the author considers harmless, if not particularly practical. Since the calculus has no counterpart of our lift operation, this seems probable on face value. First, let us consider how a row

[9] In the authors' formulation, this constraint is present in the kind of α, but its nature remains unchanged.

$\langle l, l\rangle$ can arise in the absence of lifts. A glance at the type system suffices to convince us that there are exactly two ways: by weakening, as a result of effect subsumption $\langle l\rangle \leq \langle l, l\rangle$, and through instantiation of a variable α in a row $\langle l \mid \alpha\rangle$ to a row $\langle l\rangle$. In the former case, the second occurrence of l is clearly a dummy, not referring to any actual algebraic operations. The latter case seems less clear-cut – but only until we consider how the row $\langle l \mid \alpha\rangle$ itself can be constructed. Again, we have to consider the effect subsumption rules, to find out that it can only arise through the subsumption $\langle l\rangle \leq \langle l \mid \alpha\rangle$,[10] thus again ensuring that the row α is not *actually* used, and so the terms that can have this effect are exactly the ones with effect $\langle l\rangle$.

Let us now consider how this approach would translate to our relational set-up. Note that in the interpretation of 'stuck' terms, we plug the expressions obtained from the interpretation of rows into *appropriately free* contexts – and in the absence of lifts, no contexts are n-free for $n > 0$. Thus, while we can *express* the interpretation of multiple effects, we are unable to actually *use* these terms. This observation points us towards a simplification of the interpretation, but at the cost of dropping the conceit that $\langle l\rangle$ and $\langle l, l\rangle$ are different – which the genesis of the row $\langle l, l\rangle$, explored above, already suggested.

In order to simplify the interpretation, we can use the insight that the tuples whose maps contain non-zero elements are useless, and that these are all the tuples where the lifting operation, $\uparrow$, acts nontrivially. Thus, instead of lifting, we can simply remove the affected tuples from the relation, ensuring that only the leftmost occurrence of each effect is taken into account. Overall, this seems to be a rather limited simplification: both the model and the calculus remain relatively complex, while the inability to refer to further occurrences of the effect requires the user to give up a significant amount of expressive power.

Let us note that Leijen suggests an operator akin to our lift, called *inject*, in [Leijen 2014] with stated purpose of preventing client-code's exceptions from interfering with exceptions raised by libraries. Moreover, [Leijen 2017a] indicates that some variant of this operator has been implemented. However, this operator is explicitly discarded from theoretical considerations, and little observation of its role in providing robust parametricity is made.

5.5 Logical relations, step-indexing and effects

Step-indexed logical relations are one of the go-to tools to reason about observational equivalence of programs in higher-order languages that feature more sophisticated constructs: from recursive types [Ahmed 2006] to higher-order store [Ahmed et al. 2009] and control operators [Dreyer et al. 2010]. While the step indices were initially rather explicit, in short time they were hidden under the 'later' modality, like for instance in Dreyer et al.'s LSLR [2011]. The notion of **COFE** as a universe of step-indexed spaces comes from Di Gianantonio and Miculan [2002], who introduced it as a more natural alternative for (a particular family of) ultrametric spaces that lends itself particularly well to formalisation.

The use of logical relations over type-and-effect systems in order to prove generic effect-based program equivalences goes back to Benton et al. [2007], who considered a language with first-order state and a region-based effect system. While we work in a similar context, the effects we consider have a rather different nature, and thus the relational interpretation of the effects is significantly different. An interesting point to note here is that the relational treatment of higher-order store, like that of Thamsborg and Birkedal [2011], required introduction of complex Kripke worlds, recursively dependent on semantic types: while the interpretation presented in this paper requires no such complexity, we conjecture that the techniques they used may be useful for extending our solution to systems that consider local declaration of algebraic effects or similar constructions.

[10] In the actual type system of Koka, this would correspond to an instance of the Open rule.

6 CONCLUSIONS AND FUTURE WORK

In this work, we have brought the technique of logical relations to a setting with row-polymorphic algebraic effects, thus providing a first semantic tool for reasoning about program equivalence in such languages. From the technical standpoint, this required a novel approach to the interpretation of effects and control-stuck expressions, which we believe can be useful also in the context of other flavours of control effects (e.g., shift/reset). Moreover, our research into relational interpretation of effects has helped us identify an interesting new programming construct, which allows the programmer to manage multiple occurrences of the same effect in an effect row. Finally, we have used the logical relation to prove some interesting and non-trivial program equivalences.

One of the potential directions of future work would be to extend our approach to features often found in programming languages that use algebraic effects, like effect instances or local definitions of effects. This would likely require a significantly more complex model, potentially along the lines of the models of higher-order state, with Kripke worlds modelling the effects accessible at a given point in the computation. Investigation of interactions of algebraic effects and abstraction (existential types, modules, etc.) is another direction that could prove fruitful both in terms of reasoning principles, and the effect on the design of the type-and-effect systems. Another potential direction of research is to investigate effect-based transformations that could be used as compiler optimisations, in order to lower the efficiency cost of programming with algebraic effects.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their thorough and insightful comments, which allowed us to clarify and amend a number of issues.

This material is based upon work supported by the National Science Centre of Poland under Grant No. 2014/15/B/ST6/00619.

REFERENCES

Samson Abramsky. 1990. The lazy lambda calculus. In *Research Topics in Functional Programming*, David A. Turner (Ed.). Addison-Wesley Longman Publishing Co., Inc., 65–116.

Danel Ahman. 2017. Handling fibred algebraic effects. (2017). https://danelahman.github.io/drafts/handling_fibred_algebraic_effects.pdf Unpublished draft.

Danel Ahman, Neil Ghani, and Gordon D. Plotkin. 2016. Dependent types and fibred computational effects. In *Foundations of Software Science and Computation Structures – 19th International Conference, FOSSACS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings (Lecture Notes in Computer Science)*, Bart Jacobs and Christof Löding (Eds.), Vol. 9634. Springer, 36–54. https://doi.org/10.1007/978-3-662-49630-5_3

Amal Ahmed, Derek Dreyer, and Andreas Rossberg. 2009. State-dependent representation independence. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, Zhong Shao and Benjamin C. Pierce (Eds.). ACM, 340–353. https://doi.org/10.1145/1480881.1480925

Amal J. Ahmed. 2006. Step-indexed syntactic logical relations for recursive and quantified types. In *Programming Languages and Systems, 15th European Symposium on Programming, ESOP 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 27-28, 2006, Proceedings (Lecture Notes in Computer Science)*, Peter Sestoft (Ed.), Vol. 3924. Springer, 69–83. https://doi.org/10.1007/11693024_6

Thorsten Altenkirch, James Chapman, and Tarmo Uustalu. 2014. Relative monads formalised. *Journal of Formalized Reasoning* 7, 1 (2014), 1–43. https://doi.org/10.6092/issn.1972-5787/4389

Andrew W. Appel and David A. McAllester. 2001. An indexed model of recursive types for foundational proof-carrying code. *ACM Transactions on Programming Languages and Systems* 23, 5 (2001), 657–683. https://doi.org/10.1145/504709.504712

Andrew W. Appel, Paul-André Melliès, Christopher D. Richards, and Jérôme Vouillon. 2007. A very modal model of a modern, major, general type system. In *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007*, Martin Hofmann and Matthias Felleisen (Eds.). ACM, 109–122. https://doi.org/10.1145/1190216.1190235

Andrej Bauer and Matija Pretnar. 2014. An effect system for algebraic effects and handlers. *Logical Methods in Computer Science* 10, 4 (2014), 1–29. https://doi.org/10.2168/LMCS-10(4:9)2014

Andrej Bauer and Matija Pretnar. 2015. Programming with algebraic effects and handlers. *Journal of Logic and Algebraic Methods in Programming* 84, 1 (2015), 108–123. https://doi.org/10.1016/j.jlamp.2014.02.001

Nick Benton, Andrew Kennedy, Lennart Beringer, and Martin Hofmann. 2007. Relational semantics for effect-based program transformations with dynamic allocation. In *Proceedings of the 9th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, July 14-16, 2007, Wrocław, Poland*, Michael Leuschel and Andreas Podelski (Eds.). ACM, 87–96. https://doi.org/10.1145/1273920.1273932

Małgorzata Biernacka, Dariusz Biernacki, and Olivier Danvy. 2005. An operational foundation for delimited continuations in the CPS hierarchy. *Logical Methods in Computer Science* 1, 2 (2005), 1–39. https://doi.org/10.2168/LMCS-1(2:5)2005

Dariusz Biernacki and Sergueï Lenglet. 2012. Normal form bisimulations for delimited-control operators. In *Functional and Logic Programming – 11th International Symposium, FLOPS 2012, Kobe, Japan, May 23-25, 2012. Proceedings (Lecture Notes in Computer Science)*, Tom Schrijvers and Peter Thiemann (Eds.), Vol. 7294. Springer, 47–61. https://doi.org/10.1007/978-3-642-29822-6_7

Dariusz Biernacki and Piotr Polesiuk. 2015. Logical relations for coherence of effect subtyping. In *13th International Conference on Typed Lambda Calculi and Applications, TLCA 2015, July 1-3, 2015, Warsaw, Poland (LIPIcs)*, Thorsten Altenkirch (Ed.), Vol. 38. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 107–122. https://doi.org/10.4230/LIPIcs.TLCA.2015.107

Edwin Brady. 2013a. Idris: general purpose programming with dependent types. In *Proceedings of the 7th Workshop on Programming Languages meets Program Verification, PLPV 2013, Rome, Italy, January 22, 2013*, Matthew Might, David Van Horn, Andreas Abel, and Tim Sheard (Eds.). ACM, 1–2. https://doi.org/10.1145/2428116.2428118

Edwin Brady. 2013b. Programming and reasoning with algebraic effects and dependent types, See [Morrisett and Uustalu 2013], 133–144. https://doi.org/10.1145/2500365.2500581

Edwin Brady. 2014. Resource-dependent algebraic effects. In *Trends in Functional Programming – 15th International Symposium, TFP 2014, Soesterberg, The Netherlands, May 26-28, 2014. Revised Selected Papers (Lecture Notes in Computer Science)*, Jurriaan Hage and Jay McCarthy (Eds.), Vol. 8843. Springer, 18–33. https://doi.org/10.1007/978-3-319-14675-1_2

Giuseppe Castagna and Andrew D. Gordon (Eds.). 2017. *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*. ACM. https://doi.org/10.1145/3009837

Derek Dreyer, Amal Ahmed, and Lars Birkedal. 2011. Logical step-indexed logical relations. *Logical Methods in Computer Science* 7, 2 (2011), 1–37. https://doi.org/10.2168/LMCS-7(2:16)2011

Derek Dreyer, Georg Neis, and Lars Birkedal. 2010. The impact of higher-order state and control effects on local relational reasoning. In *Proceeding of the 15th ACM SIGPLAN International Conference on Functional Programming, ICFP 2010, Baltimore, Maryland, USA, September 27-29, 2010*, Paul Hudak and Stephanie Weirich (Eds.). ACM, 143–156. https://doi.org/10.1145/1863543.1863566

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

Matthias Felleisen and Daniel P. Friedman. 1986. Control operators, the SECD machine, and the λ-calculus. In *Formal Description of Programming Concepts III*, Martin Wirsing (Ed.). Elsevier Science Publishers B.V. (North-Holland), 193–217.

Pietro Di Gianantonio and Marino Miculan. 2002. A unifying approach to recursive and co-recursive definitions. In *Types for Proofs and Programs, Second International Workshop, TYPES 2002, Berg en Dal, The Netherlands, April 24-28, 2002, Selected Papers (Lecture Notes in Computer Science)*, Herman Geuvers and Freek Wiedijk (Eds.), Vol. 2646. Springer, 148–161. https://doi.org/10.1007/3-540-39185-1_9

Daniel Hillerström and Sam Lindley. 2016. Liberating effects with rows and handlers. In *Proceedings of the 1st International Workshop on Type-Driven Development, TyDe@ICFP 2016, Nara, Japan, September 18, 2016*, James Chapman and Wouter Swierstra (Eds.). ACM, 15–27. https://doi.org/10.1145/2976022.2976033

Daniel Hillerström, Sam Lindley, Robert Atkey, and K. C. Sivaramakrishnan. 2017. Continuation passing style for effect handlers. In *2nd International Conference on Formal Structures for Computation and Deduction, FSCD 2017, September 3-9, 2017, Oxford, UK (LIPIcs)*, Dale Miller (Ed.), Vol. 84. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 18:1–18:19. https://doi.org/10.4230/LIPIcs.FSCD.2017.18

Martin Hyland and John Power. 2007. The category theoretic understanding of universal algebra: Lawvere theories and monads. *Electronic Notes in Theoretical Computer Science* 172 (2007), 437–458. https://doi.org/10.1016/j.entcs.2007.02.019

Ohad Kammar, Sam Lindley, and Nicolas Oury. 2013. Handlers in action, See [Morrisett and Uustalu 2013], 145–158. https://doi.org/10.1145/2500365.2500590

Ohad Kammar and Matija Pretnar. 2017. No value restriction is needed for algebraic effects and handlers. *Journal of Functional Programming* 27 (2017), e7. https://doi.org/10.1017/S0956796816000320

Shin-ya Katsumata. 2013. Relating computational effects by $\top\top$-lifting. *Information and Computation* 222 (2013), 228–246. https://doi.org/10.1016/j.ic.2012.10.014

Oleg Kiselyov, Amr Sabry, and Cameron Swords. 2013. Extensible effects: an alternative to monad transformers. In *Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell, Boston, MA, USA, September 23-24, 2013*, Chung-chieh Shan (Ed.). ACM, 59–70. https://doi.org/10.1145/2503778.2503791

Oleg Kiselyov and Kc Sivaramakrishnan. 2016. Eff directly in OCaml. (2016). ACM SIGPLAN Workshop on ML, September 2016, Nara, Japan.

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

Daan Leijen. 2014. Koka: Programming with row polymorphic effect types. In *Proceedings of the 5th Workshop on Mathematically Structured Functional Programming, MSFP@ETAPS 2014, Grenoble, France, 12 April 2014 (EPTCS)*, Paul Blain Levy and Neel Krishnaswami (Eds.), Vol. 153. 100–126. https://doi.org/10.4204/EPTCS.153.8

Daan Leijen. 2017a. Structured asynchrony with algebraic effects. In *Proceedings of the 2nd International Workshop on Type-Driven Development, TyDe@ICFP 2017, Oxford, UK, September 2017*, Sam Lindley and Brent Yorgey (Eds.). ACM, 16–29.

Daan Leijen. 2017b. Type directed compilation of row-typed algebraic effects, See [Castagna and Gordon 2017], 486–499. https://doi.org/10.1145/3009837

Paul Blain Levy. 2004. *Call-By-Push-Value: A Functional/Imperative Synthesis*. Semantics Structures in Computation, Vol. 2. Springer.

Sam Lindley. 2014. Algebraic effects and effect handlers for idioms and arrows. In *Proceedings of the 10th ACM SIGPLAN Workshop on Generic Programming, WGP 2014, Gothenburg, Sweden, August 31, 2014*, José Pedro Magalhães and Tiark Rompf (Eds.). ACM, 47–58. https://doi.org/10.1145/2633628.2633636

Sam Lindley, Conor McBride, and Craig McLaughlin. 2017. Do be do be do, See [Castagna and Gordon 2017], 500–514. https://doi.org/10.1145/3009837

Eugenio Moggi. 1991. Notions of computation and monads. *Information and Computation* 93, 1 (1991), 55–92. https://doi.org/10.1016/0890-5401(91)90052-4

James H. Morris. 1968. *Lambda Calculus Models of Programming Languages*. Ph.D. Dissertation. Massachusets Institute of Technology.

Greg Morrisett and Tarmo Uustalu (Eds.). 2013. *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA – September 25 - 27, 2013*. ACM.

Andrew Pitts and Ian Stark. 1998. Operational reasoning for functions with local state. In *Higher Order Operational Techniques in Semantics*, Andrew Gordon and Andrew Pitts (Eds.). Publications of the Newton Institute, Cambridge University Press, 227–273.

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 John Power. 2001a. Adequacy for algebraic effects. In *Foundations of Software Science and Computation Structures, 4th International Conference, FOSSACS 2001, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6, 2001, Proceedings (Lecture Notes in Computer Science)*, Furio Honsell and Marino Miculan (Eds.), Vol. 2030. Springer, 1–24. https://doi.org/10.1007/3-540-45315-6_1

Gordon D. Plotkin and John Power. 2001b. Semantics for algebraic operations. *Electronic Notes in Theoretical Computer Science* 45 (2001), 332–345. https://doi.org/10.1016/S1571-0661(04)80970-8

Gordon D. Plotkin and John Power. 2002. Notions of computation determine monads. In *Foundations of Software Science and Computation Structures, 5th International Conference, FOSSACS 2002, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002 Grenoble, France, April 8-12, 2002, Proceedings (Lecture Notes in Computer Science)*, Mogens Nielsen and Uffe Engberg (Eds.), Vol. 2303. Springer, 342–356. https://doi.org/10.1007/3-540-45931-6_24

Gordon D. Plotkin and Matija Pretnar. 2008. A logic for algebraic effects. In *Proceedings of the Twenty-Third Annual IEEE Symposium on Logic in Computer Science, LICS 2008, 24-27 June 2008, Pittsburgh, PA, USA*. IEEE Computer Society, 118–129. https://doi.org/10.1109/LICS.2008.45

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

Matija Pretnar. 2010. *Logic and Handling of Algebraic Effects*. Ph.D. Dissertation. University of Edinburgh, UK.

Matija Pretnar. 2015. An introduction to algebraic effects and handlers. Invited tutorial paper. *Electronic Notes in Theoretical Computer Science* 319 (2015), 19–35. https://doi.org/10.1016/j.entcs.2015.12.003

Didier Rémy. 1994. Type inference for records in natural extension of ML. In *Theoretical Aspects of Object-Oriented Programming*, Carl A. Gunter and John C. Mitchell (Eds.). MIT Press, 67–95.

John C. Reynolds. 1983. Types, abstraction and parametric polymorphism. In *Information Processing 83, Proceedings of the IFIP 9th World Computer Congress, Paris, France, September 19-23, 1983*, R. E. A. Mason (Ed.). North-Holland/IFIP, 513–523.

Davide Sangiorgi, Naoki Kobayashi, and Eijiro Sumii. 2011. Environmental bisimulations for higher-order languages. *ACM Transactions on Programming Languages and Systems* 33, 1 (2011), 5:1–5:69. https://doi.org/10.1145/1889997.1890002

Tom Schrijvers, Nicolas Wu, Benoit Desouter, and Bart Demoen. 2014. Heuristics entwined with handlers combined: From functional specification to logic programming implementation. In *Proceedings of the 16th International Symposium on Principles and Practice of Declarative Programming, Kent, Canterbury, United Kingdom, September 8-10, 2014*, Olaf Chitil, Andy King, and Olivier Danvy (Eds.). ACM, 259–270. https://doi.org/10.1145/2643135.2643145

Jacob Thamsborg and Lars Birkedal. 2011. A Kripke logical relation for effect-based program transformations. 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, 445–456. https://doi.org/10.1145/2034773.2034831

Nicolas Wu and Tom Schrijvers. 2015. Fusion for free – efficient algebraic effect handlers. In *Mathematics of Program Construction – 12th International Conference, MPC 2015, Königswinter, Germany, June 29 - July 1, 2015. Proceedings (Lecture Notes in Computer Science)*, Ralf Hinze and Janis Voigtländer (Eds.), Vol. 9129. Springer, 302–322. https://doi.org/10.1007/978-3-319-19797-5_15

$$
\begin{array}{rcll}
v & ::= & \lambda^{\rho} x.e \mid \Lambda^{\rho}.e \mid () \mid op_l \mid \theta & \text{(value)} \\
\rho & ::= & \{\} \mid \rho\{x \mapsto v\} & \text{(environment)} \\
\kappa & ::= & \bullet \mid \iota : \kappa & \text{(stack)} \\
\iota & ::= & e^{\rho} \mid v \mid * & \text{(stack frame)} \\
\pi & ::= & \bullet \mid \delta : \pi & \text{(meta-stack)} \\
\delta & ::= & (\mu, \kappa) & \text{(meta-stack frame)} \\
\mu & ::= & l \mid hr_l^{\rho} & \text{(meta-stack marker)} \\
\theta & ::= & \bullet \mid \delta : \theta & \text{(reified meta-stack)}
\end{array}
$$

$$
\begin{array}{ll}
\langle e \mid \rho \mid \kappa \mid \pi \rangle_{\text{eval}} & \text{(eval configuration)} \\
\langle \kappa \mid v \mid \pi \rangle_{\text{stack}} & \text{(stack configuration)} \\
\langle op_l \mid n \mid \kappa \mid \pi \mid v \mid \theta \rangle_{\text{op}} & \text{(operation configuration)} \\
\langle \theta \mid \kappa \mid \pi \mid v \rangle_{\text{res}} & \text{(resumption configuration)} \\
\langle \pi \mid v \rangle_{\text{mstack}} & \text{(meta-stack configuration)}
\end{array}
$$

Fig. 10. The abstract machine. The syntax.

A ABSTRACT MACHINE

In this section we present an abstract machine for the calculus $\lambda^{\mathsf{H/L}}$. The abstract machine provides a model implementation for the calculus, and it is based on the architecture of the definitional abstract machine for the control operators shift and reset [Biernacka et al. 2005]. The definitional abstract machine for shift and reset extends the CEK abstract machine [Felleisen and Friedman 1986], the canonical abstract machine for the call-by-value λ-calculus, with an additional layer of stack, called the meta-stack. The structure of the stack in the abstract machine for $\lambda^{\mathsf{H/L}}$ is richer in that it contains stack markers [Dybvig et al. 2007] corresponding to the lift expressions and handlers, that are dynamically explored in search of the right handler, whenever an operation is being handled.

The syntax of the abstract machine is presented in Figure 10. Values computed by the abstract machine include closures ($\lambda^{\rho} x.e$ and $\Lambda^{\rho}.e$), unit ($()$), operations (op_l), and reified continuations (θ). The machine uses the environment ρ that is a structure associating variables with values. The empty environment is written $\{\}$, updating an environment is written $\rho\{x \mapsto v\}$, and looking up a variable in an environment is written $\rho(x)$.

A stack κ is a list of stack frames, where $\bullet$ represents the empty stack, and $\iota : \kappa$ is the result of pushing ι on the stack κ. The stack frames e^{ρ} and v represent the operand (an expression coupled with its environment) and the operator (a value) in the call-by-value evaluation of expression application, whereas the stack frame $*$ represents the fake type argument for a type abstraction.

A meta-stack π is a list of meta-stack frames, where $\bullet$ represents the empty meta-stack, and $\delta : \pi$ is the result of pushing δ on the meta-stack π. A meta-stack frame (μ, κ) consists of a stack marker μ, i.e., either a lift marker l or a handler hr_l^{ρ} (a closure of the handler, really), and a stack κ. Since in $\lambda^{\mathsf{H/L}}$ we do not assume a top-level handler, it is not possible to represent the stack as a list of frames terminated with a marker, and the meta-stack as a list of such stacks. Instead we represent the complete stack as a pair κ_1 and $(\mu_1, \kappa_2) : \cdots : (\mu_n, \kappa_{n+1}) : \bullet$, where μ_i separates κ_i and κ_{i+1}.

The abstract machine operates in five modes. The modes eval, stack and mstack are standard – they interpret expressions, stacks, and meta-stacks, respectively. The configuration $\langle op_l \mid n \mid \kappa \mid \pi \mid v \mid \theta \rangle_{\text{op}}$ represents the process of searching the meta-stack π for the right handler for the operation op_l, counting the number of encountered lift markers and handlers for effect l, and accumulating

$$
\begin{array}{rcl}
e & \Rightarrow & \langle e \mid \{\} \mid \bullet \mid \bullet \rangle_{\mathsf{eval}} \\
\\
\langle x \mid \rho \mid \kappa \mid \pi \rangle_{\mathsf{eval}} & \Rightarrow & \langle \kappa \mid v \mid \pi \rangle_{\mathsf{stack}} \\
 & & \text{where } v = \rho(x) \\
\langle \lambda x.e \mid \rho \mid \kappa \mid \pi \rangle_{\mathsf{eval}} & \Rightarrow & \langle \kappa \mid \lambda^{\rho} x.e \mid \pi \rangle_{\mathsf{stack}} \\
\langle \Lambda.e \mid \rho \mid \kappa \mid \pi \rangle_{\mathsf{eval}} & \Rightarrow & \langle \kappa \mid \Lambda^{\rho}.e \mid \pi \rangle_{\mathsf{stack}} \\
\langle () \mid \rho \mid \kappa \mid \pi \rangle_{\mathsf{eval}} & \Rightarrow & \langle \kappa \mid () \mid \pi \rangle_{\mathsf{stack}} \\
\langle op_l \mid \rho \mid \kappa \mid \pi \rangle_{\mathsf{eval}} & \Rightarrow & \langle \kappa \mid op_l \mid \pi \rangle_{\mathsf{stack}} \\
\langle e_1\ e_2 \mid \rho \mid \kappa \mid \pi \rangle_{\mathsf{eval}} & \Rightarrow & \langle e_1 \mid \rho \mid e_2{}^{\rho} : \kappa \mid \pi \rangle_{\mathsf{eval}} \\
\langle e * \mid \rho \mid \kappa \mid \pi \rangle_{\mathsf{eval}} & \Rightarrow & \langle e \mid \rho \mid * : \kappa \mid \pi \rangle_{\mathsf{eval}} \\
\langle [e]_l \mid \rho \mid \kappa \mid \pi \rangle_{\mathsf{eval}} & \Rightarrow & \langle e \mid \rho \mid \bullet \mid (l, \kappa) : \pi \rangle_{\mathsf{eval}} \\
\langle \mathsf{handle}_l\ e\ \{hr\} \mid \rho \mid \kappa \mid \pi \rangle_{\mathsf{eval}} & \Rightarrow & \langle e \mid \rho \mid \bullet \mid (hr_l^{\rho}, \kappa) : \pi \rangle_{\mathsf{eval}} \\
\\
\langle \bullet \mid v \mid \pi \rangle_{\mathsf{stack}} & \Rightarrow & \langle \pi \mid v \rangle_{\mathsf{mstack}} \\
\langle e^{\rho} : \kappa \mid v \mid \pi \rangle_{\mathsf{stack}} & \Rightarrow & \langle e \mid \rho \mid v : \kappa \mid \pi \rangle_{\mathsf{eval}} \\
\langle * : \kappa \mid \Lambda^{\rho}.e \mid \pi \rangle_{\mathsf{stack}} & \Rightarrow & \langle e \mid \rho \mid \kappa \mid \pi \rangle_{\mathsf{eval}} \\
\langle \lambda^{\rho} x.e : \kappa \mid v \mid \pi \rangle_{\mathsf{stack}} & \Rightarrow & \langle e \mid \rho\{x \mapsto v\} \mid \kappa \mid \pi \rangle_{\mathsf{eval}} \\
\langle op_l : \kappa \mid v \mid \pi \rangle_{\mathsf{stack}} & \Rightarrow & \langle op_l \mid 0 \mid \kappa \mid \pi \mid v \mid \bullet \rangle_{\mathsf{op}} \\
\langle \theta : \kappa \mid v \mid \pi \rangle_{\mathsf{stack}} & \Rightarrow & \langle \theta \mid \kappa \mid \pi \mid v \rangle_{\mathsf{res}} \\
\\
\langle op_l \mid n \mid \kappa \mid (l, \kappa') : \pi \mid v \mid \theta \rangle_{\mathsf{op}} & \Rightarrow & \langle op_l \mid n+1 \mid \kappa' \mid \pi \mid v \mid (l, \kappa) : \theta \rangle_{\mathsf{op}} \\
\langle op_l \mid n \mid \kappa \mid (l', \kappa') : \pi \mid v \mid \theta \rangle_{\mathsf{op}} & \Rightarrow & \langle op_l \mid n \mid \kappa' \mid \pi \mid v \mid (l', \kappa) : \theta \rangle_{\mathsf{op}} \\
 & & \text{if } l \neq l' \\
\langle op_l \mid 0 \mid \kappa \mid (\{h; d\}_l^{\rho}, \kappa') : \pi \mid v \mid \theta \rangle_{\mathsf{op}} & \Rightarrow & \langle e \mid \rho\{x \mapsto v\}\{r \mapsto (\{h; d\}_l^{\rho}, \kappa) : \theta\} \mid \kappa' \mid \pi \rangle_{\mathsf{eval}} \\
 & & \text{where } op\ x, r.\ e \in h \\
\langle op_l \mid n \mid \kappa \mid (hr_l^{\rho}, \kappa') : \pi \mid v \mid \theta \rangle_{\mathsf{op}} & \Rightarrow & \langle op_l \mid n-1 \mid \kappa' \mid \pi \mid v \mid (hr_l^{\rho}, \kappa) : \theta \rangle_{\mathsf{op}} \\
 & & \text{if } n \neq 0 \\
\langle op_l \mid n \mid \kappa \mid (hr_{l'}^{\rho}, \kappa') : \pi \mid v \mid \theta \rangle_{\mathsf{op}} & \Rightarrow & \langle op_l \mid n \mid \kappa' \mid \pi \mid v \mid (hr_{l'}^{\rho}, \kappa) : \theta \rangle_{\mathsf{op}} \\
 & & \text{if } l \neq l' \\
\\
\langle \bullet \mid \kappa \mid \pi \mid v \rangle_{\mathsf{res}} & \Rightarrow & \langle \kappa \mid v \mid \pi \rangle_{\mathsf{stack}} \\
\langle (\mu, \kappa') : \theta \mid \kappa \mid \pi \mid v \rangle_{\mathsf{res}} & \Rightarrow & \langle \theta \mid \kappa' \mid (\mu, \kappa) : \pi \mid v \rangle_{\mathsf{res}} \\
\\
\langle (l, \kappa) : \pi \mid v \rangle_{\mathsf{mstack}} & \Rightarrow & \langle \kappa \mid v \mid \pi \rangle_{\mathsf{stack}} \\
\langle (\{h; \mathsf{return}\ x.\ e\}_l^{\rho}, \kappa) : \pi \mid v \rangle_{\mathsf{mstack}} & \Rightarrow & \langle e \mid \rho\{x \mapsto v\} \mid \kappa \mid \pi \rangle_{\mathsf{eval}} \\
\langle \bullet \mid v \rangle_{\mathsf{mstack}} & \Rightarrow & v
\end{array}
$$

Fig. 11. The abstract machine. The transitions.

the traversed meta-stack (in reversed order) in θ. The configuration $\langle \theta \mid \kappa \mid \pi \mid v \rangle_{\mathsf{res}}$ resumes the reified meta-stack θ, recursively concatenating it with the current stack represented by κ and π.

The transitions of the abstract machine are presented in Figure 11. The evaluation of an expression e starts the machine in the initial configuration $\langle e \mid \{\} \mid \bullet \mid \bullet \rangle_{\mathsf{eval}}$, whereas the result v of evaluation is unloaded from the final configuration $\langle \bullet \mid v \rangle_{\mathsf{mstack}}$.

When the machine is in the op-mode, it searches the first handler for a given operation op_l in the meta-stack for which the counter n is equal 0. Whenever a lift marker for effect l is encountered, the counter is incremented, and whenever a handler for l is encountered but the counter is not equal 0, it is decremented. When the search process runs, the traversed meta-context is being accumulated and finally it is stored in the environment as the resume argument of the operation handler. Notice that given a complete stack represented by κ_1 and $(\mu_1, \kappa_2) : \cdots : (\mu_n, \kappa_{n+1}) : \pi$, where μ_n is the right handler, the captured meta-stack is $(\mu_n, \kappa_n) : \cdots : (\mu_1, \kappa_1) : \bullet$, whereas the remaining complete stack is formed by κ_{n+1} and π.

When the machine is in the res-mode, the captured meta-stack $(\mu_n, \kappa_n) : \cdots : (\mu_1, \kappa_1) : \bullet$ is recursively pushed frame by frame on the current complete stack given by κ and π, yielding a new complete stack formed by κ_1 and $(\mu_1, \kappa_2) : \cdots : (\mu_n, \kappa) : \pi$. The protocol of capturing and resuming delimited continuations that our abstract machine implements is as realistic as, e.g., the one presented in [Hillerström and Lindley 2016], where captured stacks are not reversed and, therefore, can be simply prepended to the current stack, when resumed. In this scenario, even if one represents the stack with a mutable data structure allowing for efficient concatenation, unless the captured continuations are one-shot, or linear, they have to be copied when resumed anyway, and in general algebraic effects require multi-shot delimited continuations, e.g., to express backtracking.

We do not provide a correctness proof of the abstract machine in this article, but it could be done the standard way by decompiling it into the calculus and establishing a simulation relation with the reduction semantics of Section 2.2.

B PRECISE DEFINITION OF THE LOGICAL RELATION

In this section we provide the detailed definition of the logical in order to show, how the complex mutually recursive definitions of Section 3 can be solved. Additionally, the definitions presented here are more explicit: we do not hide parameters Σ and Δ.

Evaluation closure.

$$\begin{aligned}
\mathcal{E} &: \mathbf{Type} \to_{\mathrm{c}} \mathbf{Eff} \to_{\mathrm{n}} \mathbf{URel}(\mathrm{Exp}) \\
\mathcal{K} &: \mathbf{Type} \to_{\mathrm{c}} \mathbf{Eff} \to_{\mathrm{n}} \mathbf{URel}(\mathrm{ECont}) \\
\mathcal{S} &: \mathbf{Type} \to_{\mathrm{c}} \mathbf{Eff} \to_{\mathrm{n}} \mathbf{URel}(\mathrm{Exp})
\end{aligned}$$

$$\begin{aligned}
(e_1, e_2) \in \mathcal{E}(\mu)(\Xi) &\iff \forall (E_1, E_2) \in \mathcal{K}(\mu)(\Xi).\ (E_1[e_1], E_2[e_2]) \in \mathbf{Obs} \\
(E_1, E_2) \in \mathcal{K}(\mu)(\Xi) &\iff \forall (v_1, v_2) \in \mu.\ (E_1[v_1], E_2[v_2]) \in \mathbf{Obs} \wedge \\
&\qquad \forall (e_1, e_2) \in \mathcal{S}(\mu)(\Xi).\ (E_1[e_1], E_2[e_2]) \in \mathbf{Obs} \\
(E_1[e_1], E_2[e_2]) \in \mathcal{S}(\mu)(\Xi) &\iff \exists \rho_1, \rho_2, \nu.\ (e_1, e_2, \rho_1, \rho_2, \nu) \in \Xi \wedge \rho_1 - \mathrm{free} E_1 \wedge \rho_2 - \mathrm{free} E_2 \wedge \\
&\qquad \forall (e_1', e_2') \in \nu.\ (E_1[e_1'], E_2[e_2']) \in \triangleright \mathcal{E}(\mu)(\Xi)
\end{aligned}$$

Effect interpretation.

$$\mathcal{F} : (\Sigma; \Delta \vdash - \longrightarrow \mathbf{Eff}^{\Delta} \to_{\mathrm{n}} \mathbf{Type}) \to_{\mathrm{c}} \Sigma; \Delta \vdash - \longrightarrow \mathbf{Eff}^{\Delta} \to_{\mathrm{n}} \mathbf{Eff}$$

$$\mathcal{F}(T)(\Sigma;\Delta \vdash \langle\rangle)\ \eta \equiv \emptyset$$
$$\mathcal{F}(T)(\Sigma;\Delta \vdash \alpha)\ \eta \equiv \eta(\alpha)$$
$$\mathcal{F}(T)(\Sigma;\Delta \vdash \langle l \mid \varepsilon\rangle)\ \eta \equiv \mathcal{L}(T)(l) \cup \mathcal{F}(T)(\Sigma;\Delta \vdash \varepsilon) \uparrow l$$

$$\mathcal{L}(T)\ l \equiv \left\{(op_l\ v_1, op_l\ v_2, \rho_l, \rho_l, \triangleright\mu_2) \;\middle|\; op : \tau_1 \to \tau_2 \in \Sigma(l) \wedge (v_1, v_2) \in \triangleright\mu_1\right\}$$
$$\text{where } \rho_l \equiv [l \mapsto 0]$$
$$\mu_1 \equiv T(\Sigma;\cdot \vdash \tau_1)(\cdot)$$
$$\mu_2 \equiv T(\Sigma;\cdot \vdash \tau_2)(\cdot)$$
$$\Xi \uparrow l \equiv \{(e_1, e_2, \rho_1 \uparrow l, \rho_2 \uparrow l, \mu) \mid (e_1, e_2, \rho_1, \rho_2, \mu) \in \Xi\}$$
$$\text{where } \rho \uparrow l \equiv \begin{cases} \rho[l \mapsto \rho(l)+1] & \text{if } l \in \mathrm{dom}(\rho) \\ \rho & \text{otherwise} \end{cases}$$

Type interpretation.

$$\boxed{\mathcal{T} : (\Sigma;\Delta \vdash - \to \mathbf{Eff}^{\Delta} \to_{\mathrm{n}} \mathbf{Type}) \to_{\mathrm{c}} \Sigma;\Delta \vdash - \to \mathbf{Eff}^{\Delta} \to_{\mathrm{n}} \mathbf{Type}}$$

$$(v_1, v_2) \in \mathcal{T}(T)(\Sigma;\Delta \vdash 1)\eta \iff v_1 = v_2 = ()$$
$$(v_1, v_2) \in \mathcal{T}(T)(\Sigma;\Delta \vdash \tau_1 \to_\varepsilon \tau_2)\eta \iff \forall (u_1, u_2) \in \mu_1.\ (v_1\ u_1, v_2\ u_2) \in \mathcal{E}(\mu_2)(\Xi)$$
$$\text{where } \mu_1 \equiv \mathcal{T}(T)(\Sigma;\Delta \vdash \tau_1)(\eta)$$
$$\mu_2 \equiv \mathcal{T}(T)(\Sigma;\Delta \vdash \tau_2)(\eta)$$
$$\Xi \equiv \mathcal{F}(T)(\Sigma;\Delta \vdash \varepsilon)(\eta)$$
$$(v_1, v_2) \in \mathcal{T}(T)(\Sigma;\Delta \vdash \forall\alpha.\tau)\eta \iff \forall R \in \mathbf{Eff}.\ (v_1\ *, v_2\ *) \in \mathcal{E}(\mu)(\emptyset)$$
$$\text{where } \mu \equiv \mathcal{T}(T)(\Sigma;\Delta, \alpha \vdash \tau)(\eta[\alpha \mapsto R])$$

Complete closed-term relations.

$$[\![\Sigma;\Delta \vdash \tau]\!]_\eta \equiv \mathrm{fix}(\mathcal{T})(\Sigma;\Delta \vdash \tau)(\eta)$$
$$[\![\Sigma;\Delta \vdash \varepsilon]\!]_\eta \equiv \mathcal{F}([\![-]\!])(\Sigma;\Delta \vdash \varepsilon)\ \eta$$
$$[\![\Sigma;\Delta \vdash \Gamma]\!]_\eta \equiv \left\{(\gamma_1, \gamma_2) \;\middle|\; \begin{array}{l} \mathrm{dom}(\gamma_1) = \mathrm{dom}(\gamma_2) = \mathrm{dom}(\Gamma) \wedge \\ \forall x \in \mathrm{dom}(\Gamma).\ (\gamma_1(x), \gamma_2(x)) \in [\![\Sigma;\Delta \vdash (\Gamma(x))]\!]_\eta \end{array}\right\}$$
$$\mathcal{E}[\![\Sigma;\Delta \vdash \tau\ /\ \varepsilon]\!]_\eta \equiv \mathcal{E}([\![\Sigma;\Delta \vdash \tau]\!]_\eta)([\![\Sigma;\Delta \vdash \varepsilon]\!]_\eta)$$
$$\mathcal{K}[\![\Sigma;\Delta \vdash \tau\ /\ \varepsilon]\!]_\eta \equiv \mathcal{K}([\![\Sigma;\Delta \vdash \tau]\!]_\eta)([\![\Sigma;\Delta \vdash \varepsilon]\!]_\eta)$$
$$\mathcal{S}[\![\Sigma;\Delta \vdash \tau\ /\ \varepsilon]\!]_\eta \equiv \mathcal{S}([\![\Sigma;\Delta \vdash \tau]\!]_\eta)([\![\Sigma;\Delta \vdash \varepsilon]\!]_\eta)$$
$$\mathcal{H}[\![\Sigma;\Delta \vdash \tau\ /\ \varepsilon]\!]^l_\eta \equiv \left\{(h_1, h_2) \;\middle|\; \begin{array}{l} \forall (op : \tau_1 \to \tau_2) \in \Sigma(l).\ \forall (op\ x_i, r_i.\ e_i) \in h_i\ |_{i\in\{1,2\}}\ . \\ \quad (\lambda x_1, r_1.e_1, \lambda x_2, r_2.e_2) \in [\![\Sigma;\Delta \vdash \tau_1 \to (\tau_2 \to_\varepsilon \tau) \to_\varepsilon \tau]\!]_\eta \end{array}\right\}$$