Binders by Day, Labels by Night

Effect Instances via Lexically Scoped Handlers

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

Handlers of algebraic effects aspire to be a practical and robust programming construct that allows one to define, use, and combine different computational effects. Interestingly, a critical problem that still bars the way to their popular adoption is how to combine different uses of *the same* effect in a program, particularly in a language with a static type-and-effect system. For example, it is rudimentary to define the "mutable memory cell" effect as a pair of operations, put and get, together with a handler, but it is far from obvious how to use this effect a number of times to operate a number of memory cells in a single context. In this paper, we propose a solution based on lexically scoped effects in which each use (an "instance") of an effect can be singled out by name, bound by an enclosing handler and tracked in the type of the expression. Such a setting proves to be delicate with respect to the choice of semantics, as it depends on the explosive mixture of effects, polymorphism, and reduction under binders. Hence, we devise a novel approach to Kripke-style logical relations that can deal with open terms, which allows us to prove the desired properties of our calculus. We formalise our core results in Coq, and introduce an experimental surface-level programming language to show that our approach is applicable in practice.

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

Additional Key Words and Phrases: algebraic effects, effect handlers, logical relations

ACM Reference Format:
Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. 2020. Binders by Day, Labels by Night: Effect Instances via Lexically Scoped Handlers. *Proc. ACM Program. Lang.* 4, POPL, Article 48 (January 2020), 29 pages. https://doi.org/10.1145/3371116

1 INTRODUCTION

Built on firm theoretical foundations [Plotkin and Power 2004; Plotkin and Pretnar 2013], algebraic effects are the recent big hitter in the area of programming with computational effects [Bauer and Pretnar 2015]. The key idea is to split a definition of an effect into two parts: a set of operations and their handler. The programmer uses the former to construct effectful expressions; for example, the operation throw to raise an exception, put and get to modify the content of a memory cell, or choose and fail to model nondeterminism. Such operations, however, have no set meaning by

Authors' addresses: Dariusz Biernacki, Institute of Computer Science, University of Wrocław, Joliot-Curie 15, Wrocław, 53-206, Poland, dabi@cs.uni.wroc.pl; Maciej Piróg, Institute of Computer Science, University of Wrocław, Joliot-Curie 15, Wrocław, 53-206, Poland, mpirog@cs.uni.wroc.pl; Piotr Polesiuk, Institute of Computer Science, University of Wrocław, Joliot-Curie 15, Wrocław, 53-206, Poland, ppolesiuk@cs.uni.wroc.pl; Filip Sieczkowski, Institute of Computer Science, University of Wrocław, Joliot-Curie 15, Wrocław, 53-206, Poland, efes@cs.uni.wroc.pl.

2475-1421/2020/1-ART48
https://doi.org/10.1145/3371116

themselves – it is an enclosing handler that specifies their concrete semantics. This approach opens new avenues of modular effectful programming. If one wants to construct a computation that makes use of two different effects, they simply use operations from both effects in the code, and then enclose it with two handlers, one for each effect. Moreover, the programmer can define custom handlers, yielding lightweight, bespoke effectful abstractions.

To classify these new modes of expression, languages with effect handlers usually come equipped with a type-and-effect system [Bauer and Pretnar 2014; Hillerström and Lindley 2016], which gives static guarantees on the effects that each expression can perform. This way, the programmer is always aware of what can happen behind the scenes: whether a function is pure, does any I/O, throws a particular kind of exceptions – and if all of them are always caught. Such systems can be quite complex, and allow for features like effect polymorphism [Leijen 2014], existential effects [Biernacki et al. 2019] (useful for combining algebraic effects with built-in effects), or linearity [Leijen 2018] (useful for physical resource management).

With all this, algebraic effects and handlers are well on their way to becoming a de facto standard of dealing with computational effects in non-pure applicative programming, especially since they have already proven their worth in the wild [Bingham et al. 2018; Dolan et al. 2015; Leijen 2017a,b]. One of the final hurdles is the problem that we tackle in this paper: while programming with *distinct* effects simultaneously is easy (because each handler knows which effect it should take care of), we often want to program with different *instances* of the same effect, for example a number of mutable memory cells, different sources of random values, and so on; see Section 2 for examples and a more technical overview. In this paper, we propose a mechanism that would allow the programmer to match operations to the intended handler more directly.

While this problem might seem innocent at first glance, its importance and difficulty is witnessed by the fact that most of the existing implementations of languages with effect handlers address it one way or another, but none of the solutions is entirely satisfactory. For example, an earlier iteration of Eff [Bauer and Pretnar 2015] used to be equipped with instances as first-class values paired with operation names and handlers. This gives the programmer a lot of expressiveness, but it is not possible to track such instances using a type-and-effect system, since it is impossible to statically determine which instances will be handled and when. Biernacki et al. [2019] introduced *effect coercions*, which are similar to the slightly more expressive *adaptors* proposed later by Convent et al. [2019]. These two solutions boil down to recognising each use of an effect by its position in the row of effects, resulting in something akin to de Bruijn indices on the level of types, which is rather expressive and well-behaved on the theoretical side, but quite impractical in real life. As another example, the Links language [Hillerström and Lindley 2016] avoids the issue altogether by explicitly disallowing two handlers for the same effect in a single context as a type-level constraint. This severely limits expressivity, but provides the desired static guarantees.

From a technical standpoint, the problems with instances stem from the fiddly semantics of effect handlers, which involves reifying evaluation contexts as first-class values; see Section 2 for a more detailed overview. In addition, the more polymorphism a language allows, the more involved the type-and-effect system must be in order to track the effects in such values, as it creates more opportunities to smuggle effects around by instantiating universal or existential quantifiers.

In this paper, we propose a solution that is simple, practical, and expressive – and which still enjoys a static type-and-effect discipline. Building on the insight that coercions/adaptors correspond very closely to the structural rules of intuitionistic logic, we use variables to denote the various instances of the effects visible in a given evaluation context. The use of variables, complete with lexical scoping rules, provides a notion of instances that is easy to grasp intuitively and fits well with the functional programming paradigm. Section 3 defines the syntax and type system of our calculus.

We furnish our calculus with two operational semantics, which we call *open* and *generative* respectively. The first one, defined in Section 4, closely follows the understanding of instances as variables, bound by a handler or a λ-abstraction, and performs reductions under variable-binding handlers using standard capture-avoiding substitution. The second one, defined in Section 5, is closer in spirit to the dynamically allocated instances of Eff. It turns out that each approach has strengths and weaknesses. In particular, while the first semantics is more intuitive, it is difficult to implement efficiently, and does not scale to some advanced features of the type-and-effect systems actually used for programming with algebraic effects, namely polymorphic effect signatures. On the other hand, the dynamically allocated instances are difficult to think about in intuitive terms, even though they scale well and are easy to implement efficiently. Thus, in Section 6, we *relate* the two: we show that in the absence of the advanced modes of polymorphism, the two semantics are equivalent. This means that we may often reason about our programs intuitively using the open semantics, and actually compute using the more robust and efficient generative semantics.

Since this equivalence does not hold in general, we use types to ensure that the two semantics coincide for programs that are well-typed without the advanced modes of polymorphism mentioned above. To this end, we introduce a novel Kripke-style logical relation that bridges the gap between the two semantics, building on the work of Biernacki et al. [2018]. An interesting technical challenge stems from the fact that one of the semantics reduces open terms (which might be given open types), thus proscribing the usual tactic of closing-off of the free variables with arbitrary matching substitutions. In order to solve this problem and provide a correct relational interpretation, we use a somewhat complex Kripke structure that makes it possible to track the evolution of free variables, and ensure that they are appropriately closed by the evaluation contexts, thus allowing us to relate computations in the open semantics to computations in the generative semantics. To the best of our knowledge, this is the first logical relation that uses the Kripke worlds in such a way, which by itself is a technical contribution of this paper. We also provide a logical relation that justifies parametricity of the fully-fledged polymorphic calculus under the generative semantics; this is more standard, following the approach to delimited control of Biernacki et al. [2018], but certain aspects of the relation are simpler and more uniform than in the previous attempts.

A matter that we consider with equal attention is the practical angle of the proposed calculus, that is, if it is possible to build a usable programming language on top of it. Since instances have proven elusive and hinging on minor, seemingly unrelated details of the type system, such a logical-relation-oriented design of a programming language appears to be crucial to provide the coveted strong guarantees, especially in the nontrivial aspect of coupling effects and polymorphism. Meanwhile, one should not forget that the primary goal of instances is to give the programmer a handy tool to manage effects in a program, so the question remains if a language based on the proposed calculus can actually be more convenient than, say, working with coercions/adaptors. In practice, such issues can be evaluated only by going through a number of examples. For this, we implement an experimental programming language and show a couple of examples in Section 7. The goal of our implementation is to provide a system of instances with a minimal overhead, in which the management of instances is kept implicit as much as possible, especially if there is only one instance of a particular effect in scope.

Our contributions offer some more general insights into operational semantics and type-and-effect systems for algebraic effects and handlers. For example, we do not need row types in our calculus, as finite sets of (instances of) effects are sufficient – in our view, such an approach is easier to grasp by the programmer and is more modular. The results can also be extended to alternative approaches to effect handlers. A thorough inspection of our logical relation reveals more permissive conditions on where one can allocate an instance of an effect – in particular, it does not have to be a handler, as long as there are no effect abstractions between the binder and the handler. This

observation readily allows one to generalise our results to languages with shallow handlers, such as Frank [Convent et al. 2019], in which it is the recursive structure of a function that should carry an instance annotation rather than the short-lived handler inside the function.

We note that the concept of lexically scoped effects is not new in itself, as it goes back at least to Birkedal et al.'s [1996] regions for mutable store. In the context of effect handlers, Brachthäuser et al. [2018] use lexical scoping in their Java library, which, however, is not equipped with a type-and-effect system. Moreover, they concentrate solely on the implementation, and do not provide a core calculus or any metatheoretic results. Zhang and Myers [2019] solve a related problem of ensuring parametricity of effect polymorphism, which is subsumed by our solution. However, Zhang and Myers' calculus relies on certain restrictions on types and effects that prevent it from scaling directly to type-and-effect systems used in practice.

The paper is accompanied by a surface-level language, based on the implementation by Biernacki et al. [2019].[1] Moreover, we formalise our calculus and meta results in Coq.[2]

Contributions

- We introduce a core calculus (syntax and a type-and-effect system) with instances of algebraic effects, in which an instance is effectively a lexically scoped variable. It is a first such calculus in "direct style" (that is, similar to the core calculi of Koka [Leijen 2017c] and Links [Hillerström and Lindley 2016]), as opposed to the previous "handler-passing" approach [Brachthäuser et al. 2018; Zhang and Myers 2019].
- We compare two styles of operational semantics for lexically scoped effects: *open*, which is more natural, but requires reduction under instance-variable-binding handlers, and *generative*, in which no such reductions occur, but the semantics is much less intuitive.
- We show that if we restrict type and effect polymorphism to expressions (disallowing polymorphic operations), the two semantics are equivalent for well-typed programs.
- We introduce a novel approach to Kripke-style logical relations that allows us to reason about the open semantics, and prove type soundness and the equivalence of the two semantics.
- Using more standard techniques, we prove parametricity under the generative semantics.
- We formalise our calculus, the two operational semantics, and the logical relations in Coq.
- We introduce an experimental programming language that allows one to program with effect instances in a practical and type-safe manner.

2 BACKGROUND: WHICH EFFECT A HANDLER HANDLES?

In this section, we give an informal overview of the operational semantics of (deep) effect handlers. Then, we explain why we believe that the problem of instances is not trivial. For a more slow-paced but in-depth introduction to handlers, consult, for example, a tutorial by Pretnar [2015].

Roughly speaking, effect handlers are a form of delimited control operators generalising exception handlers, in particular those with the additional continuation branch [Benton and Kennedy 2001]. Indeed, effect handlers are closely related to the shift_0/reset control operators [Forster et al. 2017; Piróg et al. 2019]. The main difference is that in the case of handlers, the control effect is interpreted by the delimiter and not the point where the continuation is captured.

In this paper, we consider the left-to-right call-by-value setting. Operations can be used as single-argument functions, but applying them to a value makes the evaluation stuck, except when it happens within a handler that can interpret this particular operation. For illustrative purposes, we write throw, ask, etc. for operations in this section, while later on in the paper, we use a more

[1] Available at https://bitbucket.org/pl-uwr/helium/src/popl20.
[2] Available at https://bitbucket.org/pl-uwr/aleff-lex.

condensed notation. The syntax for a handler is **handle** e $\{h; r\}$, where e is the handled expression, which can use the operations from the effect interpreted by the handler, r is a return clause of the form **return** $x.\ e_r$, which binds a variable x inside the expression e_r, while h is a list of clauses of the form op $x, k.\ e_h$, each interpreting one operation op from the effect, where the variable x stands for the value of the argument of the operation, k is the captured continuation, while e_h interprets the entire handled expression. The following reduction rules explain the basic operational semantics of handlers, where E is an evaluation context, v is a value, and e is an expression:

$$\textbf{handle}\ v\ \{h; \textbf{return}\ x.\ e_r\} \to e_r\{v\,/\,x\}$$

$$\textbf{handle}\ E[\mathrm{op}\ v]\ \{h; r\} \to e_h\{v\,/\,x\}\{\boldsymbol{\lambda}\, z.\ \textbf{handle}\ E[z]\ \{h; r\}\,/\,k\} \quad \text{where} \quad \mathrm{op}\ x, k.\ e_h \in h$$

Thus, the result of handling a value is specified by the return clause, which can depend on this value. The result of handling an expression with an operation at the evaluation position is given by the value of the expression e_h defined in the appropriate clause in the handler, which can use the value of the argument of the operation and the captured continuation (a "resumption"). The continuation is delimited by the handler, and accepts values to be put in place of the operation call.

The expression e_h may use the resumption, but it does not have to. For example, consider an effect with one operation throw, modelling a single exception. Then, to obtain the behaviour of exceptions, we can define a handler that simply discards the resumption and replaces the entire handled computation with a value:

$$\textbf{handle}\ 2 + \mathsf{throw}\ ()\ \{\mathsf{throw}\ (), k.\ 42; \textbf{return}\ x.\ x\} \to 42$$

We can also continue the evaluation using the resumption. For example, consider the Reader effect, in which the operation ask is used to retrieve some ambient value. For this, we use $h \triangleq \mathsf{ask}\ (), k.\ k\ 42$ and $r_{\mathrm{id}} \triangleq \textbf{return}\ x.\ x$:

$$\textbf{handle}\ \mathsf{ask}\ () + \mathsf{ask}\ ()\ \{h; r_{\mathrm{id}}\} \to^* \textbf{handle}\ 42 + \mathsf{ask}\ ()\ \{h; r_{\mathrm{id}}\} \to^* 84$$

We can also use the resumption a number of times, for example to model nondeterminism equipped with the operation pick, which takes as its argument a list of possible values. We can define a handler that collects all possible results of the entire computation on a list as follows:

$$\textbf{handle}\ \mathsf{pick}\ [1,2] + \mathsf{pick}\ [10,40]\ \{\mathsf{pick}\ xs, k.\ \mathsf{concatMap}\ k\ xs; \textbf{return}\ x.\ [x]\} \to^* [11, 41, 12, 42]$$

Another standard example is the effect of a single mutable memory cell. Its signature consists of two operations, put and get, which can be interpreted using the following handler:

$$\{\mathsf{get}\ (), k.\ \boldsymbol{\lambda}\, s.\ k\ s\ s; \mathsf{put}\ s', k.\ \boldsymbol{\lambda}\, s.\ k\ ()\ s'; \textbf{return}\ x.\ \boldsymbol{\lambda}\, s.\ x\}$$

To program with a number of *distinct* effects, we simply use the operations that we need, and put the expression in two nested handlers:

$$\textbf{handle handle}\ \mathsf{ask}\ () + \mathsf{throw}\ ()\ \{\mathsf{ask}\ (), k.\ k\ 42; r_{\mathrm{id}}\}\ \{\mathsf{throw}\ (), k.\ 43; r_{\mathrm{id}}\} \to^* 43$$

As specified by the reduction rules, each handler deals with the operations that it knows how to interpret. The situation becomes more ambiguous if we use *the same* effect twice:

$$\textbf{handle handle}\ \mathsf{ask}\ () + \mathsf{ask}\ ()\ \{\mathsf{ask}\ (), k.\ k\ 42; r_{\mathrm{id}}\}\ \{\mathsf{ask}\ (), k.\ k\ 43; r_{\mathrm{id}}\}$$

According to the reduction rules above, the behaviour of this program is not deterministic, as each handler can claim the ask operations. To make the semantics deterministic and more systematic, virtually all implementations of existing languages assume that by default it is the innermost handler that interprets each matching operation. It is quite arbitrary, and (without some additional language constructs) it makes it difficult to work with two distinct ambient values in the example above, and different instances of the same effect in general.

One ad hoc solution would be to use two different effects with similar signatures. But this can be done only if all the instances can be statically determined, which is not possible, for example, when writing a library code, which should work with all possible clients, no matter how many instances they need; see Section 7 for an example of a tiny library for the mutable state cell effect. Thus, we need a way to distinguish between different instances of the same effect, and this paper explores the simple idea of naming each instance, both the handler and the operations. With this, one can define the following program:

$$\mathbf{handle}_a\ \mathbf{handle}_b\ \mathsf{ask}_a\ ()\ +\ \mathsf{ask}_b\ ()\ \{\mathsf{ask}\ (), k.\ k\ 42; r_{\mathrm{id}}\}\ \{\mathsf{ask}\ (), k.\ k\ 43; r_{\mathrm{id}}\} \longrightarrow^* 85$$

Using names, however, turns out to be surprisingly delicate. Consider the following example, which uses an effect with a single operation op, where E is an evaluation context, r is arbitrary, and $h \triangleq \mathrm{op}\ x, k.\ k\ (\boldsymbol{\lambda}\ ().\ k\ (\boldsymbol{\lambda}\ ().\ ()))$:

$$\mathbf{handle}_a\ E[\mathrm{op}_a\ ()\ ()]\ \{h; r\} \longrightarrow^* \mathbf{handle}_a\ E[\mathbf{handle}_a\ E[()]\ \{h; r\}]\ \{h; r\}$$

Since the resumption captures the entire evaluation context together with the handler, we managed to duplicate it. Thus, if a is an entity that is static or dynamically-generated earlier in the program, we seem to be back where we started: we have two nested handlers, each for the same effect and the same instance.

The solution that we adopt in this paper is to name instances with variables, which are bound by handlers. This way, operations are lexically scoped, so no ambiguities occur, but there is a price: the naive extension of the operational semantics (that is, the open semantics defined in Section 4) reduces expressions under binders. This, especially in the presence of parametric polymorphism, is a walk on thin ice. To illustrate the subtleties, let us reveal that if we employ the naive semantics, the ice breaks when we allow for polymorphic types of operations. This can be alleviated by a more involved operational semantics (the generative semantics in Section 5), but the equivalence of the two semantics in the basic setting seems to require rather heavyweight relational machinery.

3 CORE CALCULUS: SYNTAX AND TYPES

In this section, we introduce the syntax and type-and-effect system of our core calculus. We actually introduce two calculi: one with type and effect polymorphism available only in expressions, and its extension in which polymorphism is available also in signatures specifying types of operations. The latter form of polymorphism is not only useful in real-life programming (the most basic example is the throw operation, which in the latter system can be given the signature $\forall\, \alpha :: \mathsf{T}.\ \mathsf{unit} \Rightarrow \alpha$), but it also adds significant power of expression [Piróg et al. 2019], which means that it is not a feature that can always be safely omitted for the sake of brevity of presentation. Our two calculi have the same syntax of terms and types, and the difference between them is that the latter has a couple of additional typing rules, marked with ($\star$).

To avoid clutter in the presentation, we assume that each effect consists of only one operation, called **do**, which is not an uncommon formulation [Piróg et al. 2019; Zhang and Myers 2019]. In our case, the operational semantics couples an operation with the right handler using an instance, so the name of the operation does not really matter, while having a number of operations in an effect is obviously desired in the surface-level language, but is usually orthogonal to the features that one wants to capture in a core calculus; see the discussion in [Piróg et al. 2019].

The peculiarity of our calculus is the presence of instances. Each occurrence of the operation is tagged with a variable that specifies the handler in the evaluation context that is supposed to interpret the operation. Handlers, on the other hand, are binders, so that an instance is local to the handled expression. In the case of library code that is not in a handler, one can use an instance

$$
\begin{array}{rll}
\mathrm{TVar} \ni \alpha, \ldots & & \text{(type variables)} \\
\mathrm{IVar} \ni a, b, \ldots & & \text{(instance variables)} \\
\mathrm{Var} \ni f, k, x, y, \ldots & & \text{(variables)} \\
\\
\mathrm{Kind} \ni \kappa & ::= \mathsf{T} \mid \mathsf{E} \mid \mathsf{S} & \text{(kinds)} \\
\mathrm{Typelike} \ni \sigma, \tau, \varepsilon & ::= \alpha \mid \mathsf{unit} \mid \tau \to_{\varepsilon} \tau \mid \forall\, \alpha :: \kappa.\ \tau \mid \forall\, a : \sigma.\ \tau & \text{(types, effects, signatures)} \\
& \quad \mid \iota \mid \varepsilon \cdot \varepsilon \mid a \mid \tau \Rightarrow \tau & \\
\\
\mathrm{Val} \ni u, v & ::= x \mid () \mid \mathbf{fun}\, f\, x.\ e \mid \Lambda\, \alpha.\ e \mid \boldsymbol{\lambda}\, a.\ e \mid & \text{(values)} \\
\mathrm{Exp} \ni e & ::= v \mid \mathbf{let}\, x = e\, \mathbf{in}\, e \mid v\ v \mid v * \mid v\ a \mid \mathbf{do}_a\ v \mid \mathbf{handle}_a\ e\ \{h; r\} & \text{(expressions)} \\
h & ::= x, k.\ e & \\
r & ::= \mathbf{return}\ x.\ e &
\end{array}
$$

Fig. 1. Syntax of the calculus

abstraction, which can be instantiated with any instance bound by a handler later on, when the library code is actually used.

Syntax. We present the syntax of terms and types in Figure 1. We assume infinite sets of variables that stand for types, effects, and signatures (TVar), instances (IVar), and values in terms (Var). Note that instance variables form a separate syntactic category from term-level variables, and by themselves do not form expressions. This entails that even though one can abstract over instances and pass them as arguments to functions, they never mix with actual expressions, hence one cannot perform any run-time computation on instances.

There are three kinds in the calculus: types (T), effects (E), and signatures (S). In the core calculus, one does not declare effects (in contrast to, e.g., [Biernacki et al. 2019]), and signatures play a role similar to type schemes in the Hindley–Milner type system: kept in the context, they describe the (possibly polymorphic) type of a particular instance, which is instantiated every time the operation is used. Note that there is no kind for instances: in the syntax and type system of the core calculus, instances exist only as variables coupling operations and handlers, and are never instantiated with anything else than other instance variables (this is no longer the case in the generative operational semantics in Section 5, which needs some additional syntactic elements).

There are nine constructors of type-level objects. We keep them in a single syntactic category for efficiency of presentation; for example, the universal quantifier for typelikes, $\forall\, \alpha :: \kappa.\ \tau$, is used both in types and signatures. By convention, the variable we use provides a hint of the object's kind – we use τ for types and typelikes of any kind, ε for effects, and σ for signatures – while the well-formedness relation is made precise in Figure 2, where it is defined in two contexts: Δ, which assigns kinds to type variables, and X, which is a set of available instance variables.

The proper types are constructed from: type variables of kind T, the base type unit, arrow type $\tau \to_{\varepsilon} \tau$ (where ε is the effect that may occur when the function is applied to an argument), universal quantifier for typelikes $\forall\, \alpha :: \kappa.\ \tau$ (which provides both type and effect polymorphism in expressions), and instance quantifier $\forall\, a : \sigma.\ \tau$ (which describes expressions that are parametric in an instance of the given signature). The effects we considered are formed by the empty effect ι (which is the effect of pure computations), effect composition $\varepsilon \cdot \varepsilon$, instance variables and type variables of kind E. While the syntax forms binary trees, we implicitly work up to a type equivalence

Well-formedness of types. $\boxed{\Delta; X \vdash \tau :: \kappa}$

$$\frac{\alpha :: \kappa \in \Delta}{\Delta; X \vdash \alpha :: \kappa} \qquad \frac{}{\Delta; X \vdash \mathsf{unit} :: \mathsf{T}} \qquad \frac{\Delta; X \vdash \tau_1 :: \mathsf{T} \qquad \Delta; X \vdash \varepsilon :: \mathsf{E} \qquad \Delta; X \vdash \tau_2 :: \mathsf{T}}{\Delta; X \vdash \tau_1 \to_\varepsilon \tau_2 :: \mathsf{T}}$$

$$\frac{\Delta, \alpha :: \kappa; X \vdash \tau :: \mathsf{T}}{\Delta; X \vdash \forall\, \alpha :: \kappa.\ \tau :: \mathsf{T}} \qquad \frac{\Delta; X \vdash \sigma :: \mathsf{S} \qquad \Delta; X \uplus \{a\} \vdash \tau :: \mathsf{T}}{\Delta; X \vdash \forall\, a : \sigma.\ \tau :: \mathsf{T}}$$

$$\frac{}{\Delta; X \vdash \iota :: \mathsf{E}} \qquad \frac{\Delta; X \vdash \varepsilon_1 :: \mathsf{E} \qquad \Delta; X \vdash \varepsilon_2 :: \mathsf{E}}{\Delta; X \vdash \varepsilon_1 \cdot \varepsilon_2 :: \mathsf{E}} \qquad \frac{a \in X}{\Delta; X \vdash a :: \mathsf{E}}$$

$$\frac{\Delta; X \vdash \tau_1 :: \mathsf{T} \qquad \Delta; X \vdash \tau_2 :: \mathsf{T}}{\Delta; X \vdash \tau_1 \Rightarrow \tau_2 :: \mathsf{S}} \qquad \frac{\Delta, \alpha :: \kappa; X \vdash \tau :: \mathsf{S}}{\Delta; X \vdash \forall\, \alpha :: \kappa.\ \tau :: \mathsf{S}}\ (\star)$$

Fig. 2. Well-formedness of types

relation that ensures that $(\mathsf{E}, \cdot, \iota)$ is a free idempotent, commutative monoid – in other words, a finite set. We choose this presentation, rather than an explicit collection of instances (which would be more akin to, say [Talpin and Jouvelot 1994]) due to the presence of effect variables, which themselves stand for entire sets of instances. Nonetheless, the notion of set-like effects makes our calculus much closer to those considered in other areas of effectful programming than the row-based systems commonly found in effect handler literature [Hillerström and Lindley 2016; Leijen 2014]. In particular, the effect $a \cdot \alpha \cdot \beta$, where a is an instance variable, while α and β are effect variables is a well-formed effect, while rows allow at most one effect (row) variable. Signatures are constructed using the bold arrow $\tau \Rightarrow \tau$ and the type variables of kind S, possibly preceded by the universal quantifier for type-level objects in the $(\star)$ variant of the calculus.

Term-level values are given by variables, the base type value $\mathbf{()}$, and three different abstractions. The first one, $\mathbf{fun}\ f\ x.\ e$, is a recursive function f with a formal argument x and a body e. We need recursive functions, because otherwise the type system would make both operational semantics always terminate, but we still use the usual notation $\boldsymbol{\lambda}\, x.\ e$ as syntactic sugar for non-recursive functions. Next, we use $\boldsymbol{\Lambda}\, \alpha.\ e$ to abstract over typelikes of any kind. Finally, we have the instance abstraction, $\boldsymbol{\lambda}\, a.\ e$. Note that since instance variables form a separate syntactic category, instance abstractions are always distinguishable from the sugar for λ-abstractions. Other expressions include let-expressions, applications corresponding to the respective abstractions (we use type and effect instantiation in the style of Ahmed [2006]), the **do** operation, and handlers. Arguments in applications and operations are values, in the style of fine-grained call-by-value [Levy et al. 2003]. Since there is only one operation, the body of each handler consists of one clause for the operation and the return clause, that is, it is of the shape $\{x, k.\ e; \mathbf{return}\ x.\ e'\}$. Both operations and handlers are decorated with variables coming from IVar. Importantly, a handler binds this variable inside the handled expression, that is, in the expression $\mathbf{handle}_a\ e\ \{h; r\}$, the instance variable a is bound in e, where it can occur as a subscript of the operation or as an argument to an instance abstraction. Note that we also consider non-binding handlers in Section 5, but we formulate them as a separate syntactic form.

Subtyping. The set-like structure of effects induces the natural notion of extension ordering, which corresponds to set inclusion. This induces a subtyping relation on types, effects and signatures, which we write $\Delta; X \vdash \tau <: \tau$. The extension ordering mentioned above states that $\Delta; X \vdash \varepsilon_1 <: \varepsilon_1 \cdot \varepsilon_2$ for any ε_2 well-formed in Δ and X. We close this rule with respect to type and effect formation rules, with the usual contravariance on the left-hand side of the arrow, as well as the type equivalence relation described above. By design, we do not allow (nontrivial) subtyping of signatures. As the rules are fairly standard, we elide them in this presentation.

Signature instantiation. Effect signatures can be instantiated as a pair of a type of the argument and a result type of the operation. This is captured by the relation $\Delta; X \vdash \sigma \rightsquigarrow \tau_1 \Rightarrow \tau_2$, which is defined below. As with the well-formedness, $(\star)$ marks the rule that we add when considering the calculus with polymorphic signatures. Note that without it, the relation becomes trivial, but overall this relation gives us a uniform and judgemental way of expressing the two cases.

$$\frac{}{\Delta; X \vdash \tau_1 \Rightarrow \tau_2 \rightsquigarrow \tau_1 \Rightarrow \tau_2} \qquad \frac{\Delta; X \vdash \tau :: \kappa \qquad \Delta; X \vdash \sigma\{\tau \,/\, \alpha\} \rightsquigarrow \tau_1 \Rightarrow \tau_2}{\Delta; X \vdash \forall\, \alpha :: \kappa.\ \sigma \rightsquigarrow \tau_1 \Rightarrow \tau_2}\ (\star)$$

Typing relations. We give the typing rules of our calculus in Figure 3. The relations are defined in three contexts: Δ assigns kinds to type variables that are in scope, much like in the well-formedness relation, Θ assigns signatures to instance variables, and Γ assigns proper types to term variables. Note that, in contrast to the previous judgements, we need to know what signature is associated with a given instance variable: when this knowledge is not necessary (as in the premises that mention well-formedness, subtyping and signature instantiation), we take $\mathrm{dom}(\Theta)$ as the set of instance variables in scope. Following the fine-grained call-by-value approach, a value is assigned a type, while an arbitrary expression has both a type and an effect. Handlers are annotated with a signature that they handle, as well as the type and effect of the resulting computation.

The rules for the constructs that do not really deal with effects are as usual in polymorphic lambda-calculi with kinds, except for the rather obvious propagation of effects, so we do not discuss them in detail. As for the constructs specific to effects, the type of an instance abstraction is, not too surprisingly, the quantifier for instances. Note that this yields a very limited form of a dependent product, since the instance variable a, even though not a value itself, may occur both as a part of the expression e, and as a part of its type τ. We extend the purity restriction convention (which generalises the staple value restriction, cf. the rule for type abstraction) to this case, to avoid potential issues with instance variables escaping their scope.

The type of $\mathbf{do}_a\ v$ is given by instantiating the signature associated with the instance variable a. The handling construct is typed using the judgement for handlers: note how the signature σ mediates between the typing of the handler and the instance a introduced for the scope of the effectful expression, and that a is clearly disjoint from ε, as it may not appear in Θ. Finally, the rules for typing the handlers are rather innocent: the rule for the simple signatures is standard, while the rule for the polymorphic signatures matches the standard rule for polymorphic abstraction.

4 EVALUATION UNDER BINDERS: THE OPEN SEMANTICS

We begin the discussion of the dynamic semantics of our calculus with the *open* semantics, where we treat handlers as binders and reduce accordingly, applying capture-avoiding substitution as needed. We believe that this is an intuitively natural approach to the operational semantics, as no additional semantic artefacts are necessary. At the same time, the semantics is bound to be nontrivial, since reductions under handlers are now reductions under binders, and the binding occurrences can be captured as resumptions. Note also that bound variables might escape their scope and become *globally free* as a result of reduction of the handler (both in case of **return** and

Typing values. $\boxed{\Delta; \Theta; \Gamma \vdash_{\mathrm{v}} v : \tau}$

$$\frac{x : \tau \in \Gamma}{\Delta; \Theta; \Gamma \vdash_{\mathrm{v}} x : \tau} \qquad \frac{}{\Delta; \Theta; \Gamma \vdash_{\mathrm{v}} () : \mathsf{unit}} \qquad \frac{\Delta; \mathrm{dom}(\Theta) \vdash \sigma :: \mathsf{S} \qquad \Delta; \Theta, a : \sigma; \Gamma \vdash e : \tau \;/\; \iota}{\Delta; \Theta; \Gamma \vdash_{\mathrm{v}} \boldsymbol{\lambda}\, a.\, e : \forall\, a : \sigma.\, \tau}$$

$$\frac{\Delta; \mathrm{dom}(\Theta) \vdash \tau_1 \to_\varepsilon \tau_2 :: \mathsf{T} \qquad \Delta; \Theta; \Gamma, f : \tau_1 \to_\varepsilon \tau_2, x : \tau_1 \vdash e : \tau_2 \;/\; \varepsilon}{\Delta; \Theta; \Gamma \vdash_{\mathrm{v}} \mathbf{fun}\, f\, x.\, e : \tau_1 \to_\varepsilon \tau_2} \qquad \frac{\Delta, \alpha :: \kappa; \Theta; \Gamma \vdash e : \tau \;/\; \iota}{\Delta; \Theta; \Gamma \vdash_{\mathrm{v}} \boldsymbol{\Lambda}\, \alpha.\, e : \forall\, \alpha :: \kappa.\, \tau}$$

Typing expressions. $\boxed{\Delta; \Theta; \Gamma \vdash e : \tau \;/\; \varepsilon}$

$$\frac{\Delta; \Theta; \Gamma \vdash_{\mathrm{v}} v : \tau}{\Delta; \Theta; \Gamma \vdash v : \tau \;/\; \iota} \qquad \frac{\Delta; \Theta; \Gamma \vdash e_1 : \tau_1 \;/\; \varepsilon \qquad \Delta; \Theta; \Gamma, x : \tau_1 \vdash e_2 : \tau_2 \;/\; \varepsilon}{\Delta; \Theta; \Gamma \vdash \mathbf{let}\, x = e_1\, \mathbf{in}\, e_2 : \tau_2 \;/\; \varepsilon}$$

$$\frac{\Delta; \Theta; \Gamma \vdash_{\mathrm{v}} u : \tau_1 \to_\varepsilon \tau_2 \qquad \Delta; \Theta; \Gamma \vdash_{\mathrm{v}} v : \tau_1}{\Delta; \Theta; \Gamma \vdash u\, v : \tau_2 \;/\; \varepsilon} \qquad \frac{\Delta; \Theta; \Gamma \vdash_{\mathrm{v}} v : \forall\, \alpha :: \kappa.\, \tau \qquad \Delta; \mathrm{dom}(\Theta) \vdash \tau' :: \kappa}{\Delta; \Theta; \Gamma \vdash v * : \tau\{\tau' \,/\, \alpha\} \;/\; \iota}$$

$$\frac{\Delta; \Theta; \Gamma \vdash_{\mathrm{v}} v : \forall\, a : \sigma.\, \tau \qquad b : \sigma \in \Theta}{\Delta; \Theta; \Gamma \vdash v\, b : \tau\{b \,/\, a\} \;/\; \iota} \qquad \frac{\Delta; \Theta; \Gamma \vdash_{\mathrm{v}} v : \tau_1 \qquad \Delta; \mathrm{dom}(\Theta) \vdash \Theta(a) \rightsquigarrow \tau_1 \Rightarrow \tau_2}{\Delta; \Theta; \Gamma \vdash \mathbf{do}_a\, v : \tau_2 \;/\; a}$$

$$\frac{\begin{array}{c}\Delta; \mathrm{dom}(\Theta) \vdash \sigma :: \mathsf{S} \qquad \Delta; \mathrm{dom}(\Theta) \vdash \tau' :: \mathsf{T} \\ \Delta; \Theta, a : \sigma; \Gamma \vdash e : \tau' \;/\; a \cdot \varepsilon \qquad \Delta; \Theta; \Gamma \vdash h : \sigma \triangleright \tau \;/\; \varepsilon \qquad \Delta; \Theta; \Gamma, x : \tau' \vdash e_r : \tau \;/\; \varepsilon\end{array}}{\Delta; \Theta; \Gamma \vdash \mathbf{handle}_a\, e\, \{h; \mathbf{return}\, x.\, e_r\} : \tau \;/\; \varepsilon}$$

$$\frac{\Delta; \Theta; \Gamma \vdash e : \tau \;/\; \varepsilon \qquad \Delta; \Theta \vdash \tau <: \tau' \qquad \Delta; \Theta \vdash \varepsilon <: \varepsilon'}{\Delta; \Theta; \Gamma \vdash e : \tau' \;/\; \varepsilon'}$$

Typing handlers. $\boxed{\Delta; \Theta; \Gamma \vdash h : \sigma \triangleright \tau \;/\; \varepsilon}$

$$\frac{\Delta; \Theta; \Gamma, x : \tau_1, r : \tau_2 \to_\varepsilon \tau \vdash e : \tau \;/\; \varepsilon}{\Delta; \Theta; \Gamma \vdash x, r.\, e : \tau_1 \Rightarrow \tau_2 \triangleright \tau \;/\; \varepsilon} \qquad \frac{\Delta, \alpha :: \kappa; \Theta; \Gamma \vdash h : \sigma \triangleright \tau \;/\; \varepsilon}{\Delta; \Theta; \Gamma \vdash h : \forall\, \alpha :: \kappa.\, \sigma \triangleright \tau \;/\; \varepsilon}\ (\star)$$

Fig. 3. Typing relations

do reductions). As an example, consider the following term:[3]

$$\mathbf{handle}_a\, \boldsymbol{\lambda}\, ().\, \mathbf{do}_a\, ()\, \{x, r.\, r\, x; r_{\mathrm{id}}\}$$

It reduces in one step to $\boldsymbol{\lambda}\, ().\, \mathbf{do}_a\, ()$, which clearly contains an unbound variable a. Due to hygienic convention, the instance variables that are freed by the reductions become inert: operations annotated by these variables can never be handled, thus behaving somewhat like dangling pointers.

The grammar of the evaluation contexts (represented inside-out) and resumptions (represented outside-in) is presented in Figure 4. Note that both the evaluation contexts and resumptions *bind*

[3]While this term does not have a type in the system we consider, it presents the general pattern in which a reduction can capture a variable under a binder. It can also be used to construct a well-typed expression with the same runtime property by hiding the type $\mathsf{unit} \to_a \mathsf{unit}$ under an existential quantifier.

$$\mathrm{ECont} \ni E ::= \Box \mid E[\mathbf{let}\, x = \Box\, \mathbf{in}\, e] \mid E[\mathbf{handle}_a\, \Box\, \{h; r\}] \qquad \text{(evaluation contexts)}$$
$$\mathrm{RCont} \ni R ::= \Box \mid \mathbf{let}\, x = R\, \mathbf{in}\, e \mid \mathbf{handle}_a\, R\, \{h; r\} \qquad \text{(resumptions)}$$

$$\frac{}{(\mathbf{fun}\, f\, x.\, e)\, v \mapsto e\{\mathbf{fun}\, f\, x.\, e\,/\,f\}\{v\,/\,x\}} \qquad \frac{}{(\Lambda\, \alpha.\, e)\, * \mapsto e} \qquad \frac{}{(\boldsymbol{\lambda}\, a.\, e)\, b \mapsto e\{b\,/\,a\}}$$

$$\frac{}{\mathbf{handle}_a\, v\, \{h; \mathbf{return}\, x.\, e\} \mapsto e\{v\,/\,x\}} \qquad \frac{R = \mathbf{handle}_a\, R'\, \{x : \tau_1, y : \tau_2.\, e; r\}}{R[\mathbf{do}_a\, v] \mapsto e\{v\,/\,x\}\{\boldsymbol{\lambda}\, z.\, R[z]\,/\,y\}}$$

$$\frac{}{\mathbf{let}\, x = v\, \mathbf{in}\, e \mapsto e\{v\,/\,x\}} \qquad \frac{e_1 \mapsto e_2}{E[e_1] \to E[e_2]}$$

Fig. 4. Open semantics

the instance variables that are handled by these contexts. We consider all evaluation contexts to be complete program contexts; to keep track of the binding structure of the contexts in the rest of the paper, it is useful to stratify them into sets ECont^X of evaluation contexts that bind precisely the variables found in a set X. We can extend this stratification to expressions, values and handlers, writing Exp^X to mean the set of expressions that are potentially open in instance variables, whereof the ones contained in X should be handled by an evaluation context, while all the rest are considered globally free. In particular, all free instance variables of $\mathrm{Exp}^\emptyset$ are globally free — and this is the set of complete programs. Thus, we can set the type of plugging an expression into the evaluation context, $-[-]$ as $\mathrm{ECont}^X \to \mathrm{Exp}^X \to \mathrm{Exp}^\emptyset$, binding all variables in X and leaving those outside X as globally free.

By convention, we equate expressions of Exp^X up to renaming of bound variables, as well as free variables *outside* of X: this is benign, as these are inert, and, if used, lead to an error independent of the choice of names. While this plays little active role in the operational semantics, we believe that — in this case, where binders are so dynamic — it is rather important to keep this notion precise.

We treat values and handlers analogous to expressions — but what of the resumptions? It turns out that it is useful to annotate them with two sets: the set Y of the variables that we expect to be bound by an evaluation context, and the set X that is bound by the handlers within R. Thus we take sets RCont^X_Y, and the type of the natural plugging operation as $\mathrm{RCont}^X_Y \to \mathrm{Exp}^{X \uplus Y} \to \mathrm{Exp}^Y$.

Now we can see how the variables become globally free: since, by the last rule, a reduction is a contraction within an evaluation context, we need to consider a contraction on expressions with the "interesting" free instance variables given by some set X that matches the variables bound by the evaluation context E. Most contractions do not interact with this set, and are rather standard; the rule for reduction of the application of instance lambda to an instance also does not introduce any new free instance variables. Unsurprisingly, the only two rules that can make a variable globally free are the contractions of a handler. If we see a value within the handler, we use the expression provided by the return clause, as explained in Section 2. However, note that the variable a may appear within v (say, in a body of a returned function) and, by hygienic convention, is not a member of X. Thus, it may become globally free, and thus inert, after the reduction. A similar effect, but on a somewhat larger scale, may occur in the contraction rule for the operation: assume that $R \in \mathrm{RCont}^Y_X$, i.e., that it binds some set of variables Y (which includes a in particular), which, by hygiene, is disjoint from X. Clearly, we have $v \in \mathrm{Exp}^{X \uplus Y}$, but at the same time $e \in \mathrm{Exp}^X$, since it is

located on the outside of the resumption. Thus, the reduction needs to set the variables of Y as globally free — and implicitly rename their bound occurrences in the reified resumption, $\boldsymbol{\lambda}\, z.\ R[z]$ — if our assumptions of hygiene are to be maintained.

While this semantics meshes well with the intuitive understanding of effect instances as lexically scoped, there are two significant limitations. Firstly, the fact that the argument of the operation escapes from the scope of the handlers in the resumption makes it unsound in the presence of the universal quantifiers in the signature. Consider the identity handler, $h_{\mathrm{id}} \triangleq x, r.\ r\ x$, which simply passes its arguments to the resumption. Clearly, we have

$$\Delta; \Theta; \Gamma \vdash h_{\mathrm{id}} : (\forall\, \alpha :: \mathsf{T}.\ \alpha \Rightarrow \alpha) \triangleright \tau\ /\ \varepsilon,$$

for any contexts, result types, and effects. Now, consider the following expression:

$$\mathbf{handle}_a\ \mathbf{handle}_b\ (\mathbf{do}_a\ (\boldsymbol{\lambda}\,().\ \mathbf{do}_b\ ()))\ ()\ \{h_{\mathrm{id}};\, r_{\mathrm{id}}\}\ \{h_{\mathrm{id}};\, r_{\mathrm{id}}\},$$

where r_{id} is the identity return clause. The signatures associated with the two instances are both polymorphic, but while the type of the operation's argument associated with b is instantiated with unit, the one associated with a is instantiated by the operation with the effectful type $\mathsf{unit} \to_b \mathsf{unit}$. Still, the entire expression typechecks with return type unit and no remaining effects. However, consider how the open semantics reduces this term: when the outer **do** operation is encountered, the matching handler is found in the evaluation context, and the argument of the operation (the suspended call to $\mathbf{do}_b$) is passed, along with the resumption, as an argument to the body of the handler. Thus, we obtain the following reduct:

$$(\boldsymbol{\lambda}\, z.\ \mathbf{handle}_a\ \mathbf{handle}_b\ z\ ()\ \{h_{\mathrm{id}};\, r_{\mathrm{id}}\}\ \{h_{\mathrm{id}};\, r_{\mathrm{id}}\})\ (\boldsymbol{\lambda}\,().\ \mathbf{do}_b\ ()).$$

Note how b has escaped its lexical scope: with the following beta reduction, it will re-enter the scope of the handler, but the hygienic convention will force us to rename the binding occurrence to some other variable, say, b' — and after another step, the reduction will get stuck. Note that the use of resumptions is crucial here: mere exceptions, which may not use the resumption, would never lead to such a problem, even if we allow polymorphic quantification.

Since polymorphic signatures are an important aspect of programming with effect handlers (for example, the operations throw and pick from Section 2 both have natural polymorphic types), this semantics is clearly restrictive. Moreover, and this is the second limitation, the necessary refreshing of variables bound by resumptions makes it at best unclear whether this semantics can be efficiently implemented. Nonetheless, we believe that this semantics is still very useful as a specification, or a model that one can take advantage of when not using polymorphic signatures; hence our attempt to establish a correspondence with the generative semantics, to which we now turn.

5 TURNING BINDERS INTO LABELS: THE GENERATIVE SEMANTICS

We now investigate an alternative semantics for our calculus, one that does not suffer from the problems encountered in the preceding section. To this end, we generate *runtime* instances that replace the binders rather than reducing under them. The idea is somewhat similar to the generative exceptions of the ML language family or certain region calculi: when a handler is encountered in an evaluation position, we pick a globally fresh runtime instance and substitute it for the instance variable. These runtime instances are global labels without any binder structure, which ensures that the evaluation of complete programs always reduces closed expressions, more in line with classic programming language semantics. The definition is presented in Figure 5.

As we can see from the definition, we assume a set of runtime instance labels, Inst, and three new forms of runtime expressions, as is usual in calculi that generate objects at runtime. In this case, instance application and operation are merely variants of the forms found in the source calculi,

$$\begin{aligned}
\text{Inst} \ni l &\ldots && \text{(instance labels)}\\
\text{Exp} \ni e &::= \ldots \mid \mathbf{handle}_l\ e\ \{h; r\} \mid v\ l \mid \mathbf{do}_l\ v\\
\text{ECont} \ni E &::= \square \mid E[\mathbf{let}\, x = \square\, \mathbf{in}\, e] \mid E[\mathbf{handle}_l\ \square\ \{h; r\}]\\
\text{RCont} \ni R &::= \square \mid \mathbf{let}\, x = R\, \mathbf{in}\, e \mid \mathbf{handle}_l\ R\ \{h; r\}
\end{aligned}$$

$$\frac{}{(\mathbf{fun}\, f\, x.\, e)\ v \mapsto e\{\mathbf{fun}\, f\, x.\, e\,/\,f\}\{v\,/\,x\}} \qquad \frac{}{(\Lambda\, \alpha.\, e)\ * \mapsto e} \qquad \frac{}{(\boldsymbol{\lambda}\, a.\, e)\ l \mapsto e\{l\,/\,a\}}$$

$$\frac{}{\mathbf{handle}_l\ v\ \{h; \mathbf{return}\ x.\, e\} \mapsto e\{v\,/\,x\}} \qquad \frac{R = \mathbf{handle}_l\ R'\ \{x, k.\, e; r\} \qquad \text{free}(l, R')}{R[\mathbf{do}_l\ v] \mapsto e\{v\,/\,x\}\{\boldsymbol{\lambda}\, z.\, R[z]\,/\,k\}}$$

$$\frac{}{\mathbf{let}\, x = v\, \mathbf{in}\, e \mapsto e\{v\,/\,x\}}$$

$$\frac{e_1 \mapsto e_2}{E[e_1] \to E[e_2]} \qquad \frac{\text{fresh}(l)}{E[\mathbf{handle}_a\ e\ \{h; r\}] \to E[\mathbf{handle}_l\ e\{l\,/\,a\}\ \{h; r\}]}$$

Fig. 5. Generative semantics

but using instance labels rather than instance variables. The final one, a new form of handler, is different in that it no longer binds an instance, but rather is tagged with a label — similar to calculi with static handler identities. The reduction rules for this runtime handler match closely with those found in earlier calculi: an operation finds the nearest enclosing handler with its label (through the simple free predicate) and continues the evaluation in the handler's body. The reduction rule for the instance application is also rather unsurprising: the runtime instance l is simply substituted for the instance variable a in the body of the function. What brings new power to the semantics is the fact that these instances are generated at runtime, in the final rule of the semantics.

In this rule, we generate an instance l, fresh with respect to the entire program, and substitute it throughout the expression for the bound occurrences of a; we also replace the lexically scoped handler of our source calculus with the more standard runtime handler. Note also that only the runtime handler and let expression generate evaluation contexts (or resumptions), and thus we can never reduce under binders.

In order to observe the behaviour of this calculus in more detail, let us consider the example from the previous section, of the program that is well-typed in the calculus that allows polymorphic signatures, but is ill-behaved in the open semantics. Recall we have

$$\mathbf{handle}_a\ \mathbf{handle}_b\ (\mathbf{do}_a\ (\boldsymbol{\lambda}\, ().\, \mathbf{do}_b\ ()))\ ()\ \{h_{\text{id}}; r_{\text{id}}\}\ \{h_{\text{id}}; r_{\text{id}}\},$$

where h_{id} and r_{id} are identity handlers and resumptions, respectively. The only reduction we can perform is turning the outer handler into a runtime handler, getting

$$\mathbf{handle}_{l_1}\ \mathbf{handle}_b\ (\mathbf{do}_{l_1}\ (\boldsymbol{\lambda}\, ().\, \mathbf{do}_b\ ()))\ ()\ \{h_{\text{id}}; r_{\text{id}}\}\ \{h_{\text{id}}; r_{\text{id}}\},$$

where l_1 is a runtime instance. The runtime handler now constitutes an evaluation context, allowing us to generate a fresh instance for the inner handler — after which the do–handle reduction may fire. Its result is

$$(\boldsymbol{\lambda}\, z.\, \mathbf{handle}_{l_1}\ \mathbf{handle}_{l_2}\ z\ ()\ \{h_{\text{id}}; r_{\text{id}}\}\ \{h_{\text{id}}; r_{\text{id}}\})\ (\boldsymbol{\lambda}\, ().\, \mathbf{do}_{l_2}\ ()),$$

which is similar to the expression obtained in the open semantics — but, crucially, without the problematic status of the instance found at the suspended **do** operation. Thus, after a series of beta reductions, we can safely reduce the second operation and proceed to return the unit value.

Not only does this semantics work well with signature polymorphism (cf. Section 6), it is also quite amenable to implementation, since it is not difficult to generate globally fresh identifiers whenever necessary. We do, however, lose much of the simplicity of the open semantics by this move, and it is not clear at face value whether this semantics is even correct. The reason for this uncertainty is that the problem discussed in Section 2 could potentially return. By equipping handlers in our calculus with binding structure, we have ensured that no two handlers in an enclosing context could be mistaken: they would be referred to by distinct variables. When using the generative semantics, this useful invariant is lost: as discussed before, we can duplicate resumptions, and thus handlers decorated with *the same* label — and we have no coercions or adaptors to distinguish them! Since we can only duplicate *the same* handler in this way, it is intuitively clear that this would not lead to well-typed programs that get outright stuck; however, this makes the mental model of the calculus much more complex. Moreover, it is not necessarily clear that this always leads to the intended semantics: in essence, that the calculus equipped with this semantics enjoys parametricity.

In fact, the sensible behaviour of the semantics hinges on the properties of the type system. What's more, in the case of programs without signature polymorphism, the types enforce the equivalence of the two semantics: a highly nontrivial property, given that we know, from the example in the preceding paragraphs, that when not restricted, the two semantics can be distinguished. We believe this equivalence to be a highly desirable property, as it allows us, in many cases, to think about the open semantics, while running our programs using the more robust and easier to implement efficiently generative one. Before turning to the semantic machinery that we use to build sound logical relations that ensure parametricity of the generative semantics and the equivalence of the two in the absence of signature polymorphism, let us note in passing that since this equivalence hinges on the properties of a rich type system, we find it unlikely that simpler methods, such as direct simulation arguments, would be sufficient to establish this property.

6 LOGICAL RELATIONS

We now turn to the properties of our calculus with its two semantics. There are two properties that we seek to establish: firstly, equivalence of the semantics in the absence of polymorphic effect signatures, and secondly, parametricity of the calculus equipped with the generative semantics, particularly in presence of polymorphic effect signatures. The latter property gives us confidence that the generative semantics is well-behaved, and thus can be considered as a sensible foundation of an implementation, while the former (beyond being an interesting theoretical observation) serves to convince us that – at least in the restricted setting – the duplication of labels in the evaluation context, which is possible in the generative semantics, does not lead to matching different handlers than those we expect (as per the open semantics). We establish both these properties by constructing appropriate step-indexed logical relations [Appel and McAllester 2001], building directly on [Biernacki et al. 2018].

We begin with the model of the calculus with signature polymorphism under the generative semantics, as it is a rather straightforward extension of Biernacki et al.'s model, and then we turn to the more complex Kripke-style model that allows us to relate the two semantics. As with the model we build on, we work implicitly in the category **COFE** of complete ordered families of equivalences [Gianantonio and Miculan 2002] and use the *later* operator, written $\triangleright$, to enforce a reduction of the step index [Appel et al. 2007; Dreyer et al. 2011]. We define recursive predicates using the unique fixed-point operator that ensures well-foundedness through the later operator, although we elide the details for clarity of presentation.

6.1 Parametricity of the Generative Semantics

While this model is closely related to the model of Biernacki et al., we still have a many-kinded system, and thus we must begin by defining the spaces that will be used as interpretations of the kinds. Thus, we define semantic types, written **Type**, effects, **Effect**, signatures, **Sig**, and computations **Comp**. We also set the former three as the interpretations of the appropriate kinds, and define the space of semantic *instances*, $\mathfrak{I}$, which we use to connect the instance variables with their interpretations and runtime counterparts:

$$\begin{aligned}
\mathbf{Comp} &\triangleq \mathbf{UPred}(\mathrm{Exp}^2) \\
[\![\mathsf{T}]\!] \triangleq \mathbf{Type} &\triangleq \mathbf{UPred}(\mathrm{Val}^2) \\
[\![\mathsf{E}]\!] \triangleq \mathbf{Effect} &\triangleq \left\{ R \in \mathbf{UPred}(\mathrm{Exp}^2 \times \mathcal{P}_{\mathrm{fin}}(\mathrm{Inst})^2 \times \mathbf{Comp}) \;\middle|\; |R|_{\mathrm{Inst}} \text{ is finite} \right\} \\
[\![\mathsf{S}]\!] \triangleq \mathbf{Sig} &\triangleq \mathbf{UPred}(\mathrm{Val}^2 \times \mathbf{Type}) \\
\mathfrak{I} &\triangleq \mathbf{Sig} \times \mathrm{Inst}^2
\end{aligned}$$

$\mathbf{UPred}(X)$ denotes the space of (step-indexed) predicates over a (non-indexed) set X. Since we work over operational semantics, the notions of semantic types and computations is standard: they are (step-indexed) relations over, respectively, closed values and closed expressions. The semantic signatures and effects follow the ideas of Biernacki et al., although somewhat factorised: the semantic signature relates a pair of values related as an *argument* of the **do** operation, together with a semantic type that describes the expected *answers*. The semantic effect relates two expressions that model control-stuck parts of a program, with a semantic computation that denotes the related answers — and two sets of instances that denote at which instance a program may be stuck. The side condition requires that any semantic effect uses only a finite set of instances; a technical measure that allows us to generate a fresh instance whenever necessary. This notion of semantic effects is close to Biernacki et al.; however, while they use finite maps to distinguish multiple uses of the same effect, we may simply refer to particular instances by name, thus simplifying the setup. On the other hand, we now require a device that links instance variables (which are a subset of open types) with runtime instances, which may appear in the expressions. To this end, we use semantic instances, which simply consist of a signature and a pair of runtime instances. This technique is quite standard for many generative aspects of programming languages, we discuss some of the related work in Section 8.

Interpretation of types, signatures, and effects. We now define an interpretation for any type (of a given kind κ) that is well-defined in some context Δ, and uses a set X of free instance variables. In order to interpret such a type, we need an interpretation for a type-variable context Δ, as well as an interpretation for instance variables contained in X: then, the denotation of the type ought to belong to $[\![\kappa]\!]$. For Δ, we can pick a simple pointwise interpretation, setting $\delta \in [\![\Delta]\!] \iff \forall \alpha.\, \delta(\alpha) \in [\![\Delta(\alpha)]\!]$. For free instance variables we may pick any semantic instance, i.e., any map of type $X \to \mathfrak{I}$ is a sensible interpretation of the set X of free instance variables. This leads us to the following type of the interpretation map for open types:

$$[\![-]\!] : [\![\Delta]\!] \to (X \to \mathfrak{I}) \to [\![\kappa]\!].$$

The definition is presented in Figure 6, and we consider constructs of each kind in turn. Observe that the interpretations of the unit and arrow types, as well as the universal quantifier over types and the type variables match the standard definitions for polymorphic lambda calculus. Note also, that – in line with our design – the effect structure is set-like: we interpret the pure effect as an empty semantic effect, and the effect composition as a sum.

$$
\begin{aligned}
[\![\alpha]\!]^{\delta,\vartheta} &\triangleq \delta(\alpha)\\[1em]
[\![\mathsf{unit}]\!]^{\delta,\vartheta} &\triangleq \{((),())\}\\
[\![\tau_1 \to_\varepsilon \tau_2]\!]^{\delta,\vartheta} &\triangleq \left\{(v_1,v_2) \;\middle|\; \forall (u_1,u_2) \in [\![\tau_1]\!]^{\delta,\vartheta}.\ (v_1\ u_1, v_2\ u_2) \in \mathcal{E}[\![\tau_2 \ /\ \varepsilon]\!]^{\delta,\vartheta}\right\}\\
[\![\forall\, \alpha :: \kappa.\ \tau]\!]^{\delta,\vartheta} &\triangleq \left\{(v_1,v_2) \;\middle|\; \forall \mu \in [\![\kappa]\!].\ (v_1\ *, v_2\ *) \in \mathcal{E}[\![\tau \ /\ \iota]\!]^{\delta[\alpha\mapsto\mu],\vartheta}\right\}\\
[\![\forall\, a : \sigma.\ \tau]\!]^{\delta,\vartheta} &\triangleq \left\{(v_1,v_2) \;\middle|\; \forall l_1, l_2, I.\ I = ([\![\sigma]\!]^{\delta,\vartheta}, l_1, l_2) \Rightarrow (v_1\ l_1, v_2\ l_2) \in \mathcal{E}[\![\tau \ /\ \iota]\!]^{\delta,\vartheta[a\mapsto I]}\right\}\\[1em]
[\![\iota]\!]^{\delta,\vartheta} &\triangleq \emptyset\\
[\![\varepsilon_1 \cdot \varepsilon_2]\!]^{\delta,\vartheta} &\triangleq [\![\varepsilon_1]\!]^{\delta,\vartheta} \cup [\![\varepsilon_2]\!]^{\delta,\vartheta}\\
[\![a]\!]^{\delta,\vartheta} &\triangleq \left\{(\mathbf{do}_{l_1}\ v_1, \mathbf{do}_{l_2}\ v_2, \{l_1\}, \{l_2\}, \mu) \;\middle|\; \exists \nu.\ \vartheta(a) = (\nu, l_1, l_2) \wedge (v_1, v_2, \mu) \in \nu\right\}\\[1em]
[\![\tau_1 \Rrightarrow \tau_2]\!]^{\delta,\vartheta} &\triangleq \left\{(v_1, v_2, [\![\tau_2]\!]^{\delta,\vartheta}) \;\middle|\; (v_1,v_2) \in [\![\tau_1]\!]^{\delta,\vartheta}\right\}\\
[\![\forall\, \alpha :: \kappa.\ \sigma]\!]^{\delta,\vartheta} &\triangleq \left\{(v_1, v_2, \mu) \;\middle|\; \exists \mu' \in [\![\kappa]\!].\ (v_1, v_2, \mu) \in [\![\sigma]\!]^{\delta[\alpha\mapsto\mu'],\vartheta}\right\}
\end{aligned}
$$

Fig. 6. Interpretation of types, effects and signatures. Throughout, δ gives interpretation of free type variables, and ϑ — the interpretation of free instance variables.

This leaves us with the features that have to do with instances and signatures. Let us first consider the singleton effect that consists of an instance variable a: clearly, ϑ gives us a semantic instance for a, which consists of a semantic signature ν and two runtime instances, l_1 and l_2. Thus, the appropriate control-stuck terms are of the form $\mathbf{do}_{l_i}\ v_i$ for some values v_i — but since we have a semantic signature for a, these values, together with the relation on the answers, should belong to that signature — ν. Let us now consider the universal quantifier over instances: we do know the signature σ, and its interpretation should clearly be associated with a in the interpretation of τ. But what runtime instances should we use? The answer, appropriate for the universal quantifier, is any two, as long as we are consistent: thus, we extend ϑ with the interpretation of σ and any pair of instances l_1 and l_2, but also require that the values are related by this environment when applied to these particular instances.

Finally, consider the interpretation of signatures. The arrow signature requires that the arguments are related at the left-hand-side type, and specifies the interpretation of the right-hand-side type as the desired relation for the result. This, unsurprisingly, corresponds closely to the interpretation of a single effect by Biernacki et al. The surprising rule is the one for polymorphic signatures, which interprets a universal quantifier via an existential one. One intuitive reading is that it is the signatures' "elimination forms" (i.e., the handlers) that need to be parametric — and thus that the nature of a parametric signature is closer to an existential quantifier than to a universal one.

Closure operators and the logical relation. We use biorthogonality [Pitts and Stark 1998] to define the closure operator for computations, with the additional closure for control-stuck expressions introduced in [Biernacki et al. 2018]; the definitions are presented in Figure 7. Following the latter work, we test evaluation contexts both with values and with *control-stuck* expressions – that is, intuitively, operations wrapped in resumptions that do not handle them. Thus, the evaluation

$$
\begin{aligned}
(e_1, e_2) \in \mathcal{E}[\![\tau \mathbin{/} \varepsilon]\!]^{\delta,\vartheta} &\iff \forall (E_1, E_2) \in \mathcal{K}[\![\tau \mathbin{/} \varepsilon]\!]^{\delta,\vartheta}.\ (E_1[e_1], E_2[e_2]) \in \mathbf{Obs} \\
(E_1, E_2) \in \mathcal{K}[\![\tau \mathbin{/} \varepsilon]\!]^{\delta,\vartheta} &\iff \forall (v_1, v_2) \in [\![\tau]\!]^{\delta,\vartheta}.\ (E_1[v_1], E_2[v_2]) \in \mathbf{Obs} \wedge \\
&\qquad \forall (e_1, e_2) \in \mathcal{S}[\![\tau \mathbin{/} \varepsilon]\!]^{\delta,\vartheta}.\ (E_1[e_1], E_2[e_2]) \in \mathbf{Obs} \\
(R_1[e_1], R_2[e_2]) \in \mathcal{S}[\![\tau \mathbin{/} \varepsilon]\!]^{\delta,\vartheta} &\iff \exists L_1, L_2, \mu.\ (e_1, e_2, L_1, L_2, \mu) \in [\![\varepsilon]\!]^{\delta,\vartheta} \\
&\qquad \wedge\ (\forall l \in L_i.\ \mathrm{free}(l, R_i))_{i \in \{1,2\}} \\
&\qquad \wedge\ \forall (e_1', e_2') \in \mu.\ (R_1[e_1'], R_2[e_2']) \in \triangleright \mathcal{E}[\![\tau \mathbin{/} \varepsilon]\!]^{\delta,\vartheta} \\
(x, k.\ e_1, x, k.\ e_2) \in \mathcal{H}[\![\sigma \triangleright \tau \mathbin{/} \varepsilon]\!]^{\delta,\vartheta} &\iff \forall u_1, u_2, v_1, v_2, \mu.\ (u_1, u_2, \mu) \in [\![\sigma]\!]^{\delta,\vartheta} \Rightarrow \\
&\qquad \left(\forall (w_1, w_2) \in \mu.\ (v_1\ w_1, v_2\ w_2) \in \mathcal{E}[\![\tau \mathbin{/} \varepsilon]\!]^{\delta,\vartheta}\right) \Rightarrow \\
&\qquad (e_1\{u_1 \mathbin{/} x\}\{v_1 \mathbin{/} y\}, e_2\{u_2 \mathbin{/} x\}\{v_2 \mathbin{/} y\}) \in \mathcal{E}[\![\tau \mathbin{/} \varepsilon]\!]^{\delta,\vartheta}
\end{aligned}
$$

$$
(e_1, e_2) \in \mathbf{Obs} \iff (e_1 = () \wedge e_2 \to^* ()) \vee \left(\exists (e_1', e_2') \in \triangleright \mathbf{Obs}.\ e_1 \to e_1' \wedge e_2 \to^* e_2'\right)
$$

Fig. 7. Closure operators for expressions, evaluation contexts, control-stuck expressions, and handlers

contexts should provide the interpretations to the "stuck" operations, which makes it a crucial part of the relation for contexts. To formalise the notion of control-stuck expressions, we use the interpretation of the effect, which describes the possible stuck forms. Note that even though the definitions seem circular, the use of the later operator in the interpretation of control-stuck terms ensures that the recursion is guarded, and thus well-defined. Also, note that in the relation for handlers we pick any *semantic* type μ for the arguments of the resumption, thus allowing for handlers where σ is a universally quantified signature. Thus, the assertion that v_1 and v_2 are related reified resumptions restates the definition for the arrow type — but with the argument type μ, rather than an interpretation of a particular syntactic type.

The only component we lack for the definition of the logical approximation relation is the interpretation of term- and instance-variable contexts. The former is standard, provided we have interpretations for type and instance variables; the latter behaves in a pattern similar to that observed in the interpretation of the instance quantifier: we take the interpretations of appropriate signatures, together with any runtime instances — but we ensure that the chosen instances will be substituted for appropriate instance variables.

$$
\begin{aligned}
(\vartheta, \eta_1, \eta_2) \in [\![\Theta]\!]^{\delta} &\iff \forall a \in \mathrm{dom}(\Theta).\ \vartheta(a) = ([\![\Theta(a)]\!]^{\delta,\vartheta}, \eta_1(a), \eta_2(a)) \\
(\gamma_1, \gamma_2) \in [\![\Gamma]\!]^{\delta,\vartheta} &\iff \forall x \in \mathrm{dom}(\Gamma).\ (\gamma_1(x), \gamma_2(x)) \in [\![\Gamma(x)]\!]^{\delta,\vartheta}
\end{aligned}
$$

Finally, we can define the logical approximation relation:[4]

$$
\begin{aligned}
&\Delta; \Theta; \Gamma \vDash e_1 \preccurlyeq e_2 : \tau \mathbin{/} \varepsilon \iff \\
&\qquad \forall \delta \in [\![\Delta]\!], (\vartheta, \eta_1, \eta_2) \in [\![\Theta]\!]^{\delta}, (\gamma_1, \gamma_2) \in [\![\Gamma]\!]^{\delta,\vartheta}.\ (\eta_1(\gamma_1(e_1)), \eta_2(\gamma_2(e_2))) \in \mathcal{E}[\![\tau \mathbin{/} \varepsilon]\!]^{\delta,\vartheta}
\end{aligned}
$$

Now we can proceed in the usual way, showing that logical approximation is a compatible congruence, which gives us the fundamental property of the logical relations, and hence soundness (given the natural definition of contextual approximation).

[4] In the following we concentrate on expressions, eliding analogous definitions and statements for values and handlers.

Theorem 1 (Fundamental). *Any well-typed expression e logically approximates itself, i.e., if $\Delta; \Theta; \Gamma \vdash e : \tau \mid \varepsilon$, then we have $\Delta; \Theta; \Gamma \vDash e \preccurlyeq e : \tau \mid \varepsilon$.*

Theorem 2 (Type Soundness). *For any well-typed program e its evaluation does not get stuck, i.e., if $\cdot; \cdot; \cdot \vdash e : \tau \mid \iota$ and $e \to^* e' \not\to$, then $e' \in \mathrm{Val}$.*

Theorem 3 (Soundness). *Logical approximation entails contextual approximation, i.e., if $\Delta; \Theta; \Gamma \vDash e_1 \preccurlyeq e_2 : \tau \mid \varepsilon$, then $\Delta; \Theta; \Gamma \vdash e_1 \preceq e_2 : \tau \mid \varepsilon$.*

Note that in the definition we did not rely on the tight connection between the handlers and generation of instances. In fact, the only aspect of the proof where the generativity and the attendant freshness of the instance is important is the compatibility of the inference rule for the **handle** construct. There, it is used to enforce the fact that the runtime instance picked for the instance variable bound by the handler is distinct from any others that could appear in the remainder of the effect, ε. This ensures that we handle precisely these operations that the type system expects to be handled. However, this notion need not be baked into the handling construct itself. In particular, this approach should readily generalise to the case of shallow handlers, such as those found in Frank [Convent et al. 2019], where instances could be provided by recursive function definitions rather than the handlers themselves. At the same time, we cannot decouple the two in general, as the separation required by the handler would not be preserved by all the constructs in our language – particularly by polymorphic abstraction. While the precise conditions required for decoupling instance generation and handlers warrant further investigation, we do not pursue this line further in the current work.

6.2 Equivalence of the Two Semantics

We now turn to the more novel logical relation that we use to prove that our two semantics are equivalent (in the absence of polymorphic signatures). To this end, we need to consider an interesting problem: in the previous section, our relations operated on closed values, expressions, and handlers — we have substituted away all term variables with closed values, and all instance variables with runtime instances, retaining a link between the latter two, since free instance variables did appear in *types* that we needed to interpret. However, the open semantics never substitutes anything for instance variables, and computes using them under (some) binders. Thus, one of the usual invariants of our methodology no longer holds: we cannot simply define a relation for closed terms and lift it to open terms via substitution. How can we proceed?

The idea behind our solution is to use a Kripke structure to track the free variable sets of the open semantics — and of their evolution under weakening, substitution or opening. These sets of variables are then used to ensure that the closing evaluation contexts give interpretation, via handlers, to precisely these variables, preserving the essence of the biorthogonal approach. There is, however, a discrepancy: while the runtime sets of available free instance variables may fluctuate with substitution and evaluation, the one in the world of types is static and never varies with time. Thus, our worlds are indexed with the *starting* set of instance variables: more precisely, we set

$$\mathbf{World}^X \triangleq \Sigma_{Y \subseteq_{\mathrm{fin}} \mathrm{IVar}} X \rightharpoonup Y$$

for any finite $X \subseteq \mathrm{IVar}$. We write our worlds (Y, ρ) when precision is needed, but often omit the set if its clear from context (as the codomain of ρ). Note that our maps are partial: this is so, since the evolution of a program may set some instance variables *globally free* (cf. Section 4): these are no longer "interesting", and surely may not be interpreted by the evaluation context.

For any pair of maps that have the same domains, we define a notion of mediation via ρ, a map between their codomains, by composition: $\rho_1 \sqsubseteq^{\rho} \rho_2 \iff \rho \circ \rho_1 = \rho_2$. This, of course, induces a

preorder on $\mathbf{World}^X$: $\rho_1 \sqsubseteq \rho_2 \iff \exists \rho.\ \rho_1 \sqsubseteq^{\rho} \rho_2$. Thus, we can treat $\mathbf{World}^X$ as a Kripke structure — as long as we can define spaces that are appropriately closed under world evolution.

We do this in two steps: first, consider a family F of sets indexed by finite sets of instance variables that is functorial with respect to partial maps between sets of instance variables.[5] Now, for any step-indexed predicate $P : \mathbf{UPred}(F(Y))$ we can retract it via a map $\rho : X \rightharpoonup Y$ obtaining a predicate over $F(X)$, as follows:

$$e \in P \downarrow \rho \iff F(\rho)(e) \in P.$$

This retraction is precisely what we need to define *world-indexed* predicates, which are closed under evolution of the world. Formally, we take $\mathbf{WIPred}^X(F)$ for some finite set of instance variables X and a functorial family of sets F to be a $\mathbf{World}^X$-indexed family R of step-indexed predicates over F at the given world that are closed under future-world transitions:

$$\mathbf{WIPred}^X(F) \triangleq \left\{ R \in \Pi_{(Y,\rho)\in\mathbf{World}^X} \mathbf{UPred}(F(Y)) \;\middle|\; \forall \rho_1 \sqsubseteq^{\rho'} \rho_2.\ R(\rho_1) \subseteq R(\rho_2) \downarrow \rho' \right\}.$$

Note how the use of retraction ensures that the relation is closed without specifying much about F.

These world-indexed predicates are at the core of our definition: one useful operator that we can define for them is shifting the starting world of a predicate $\mu : \mathbf{WIPred}^X(F)$ along a map $\rho : X \rightharpoonup Y$, as follows:

$$(\mu \Uparrow \rho)(\rho') \triangleq \mu(\rho' \circ \rho),$$

where $\rho' : Y \rightharpoonup Z$, since composition preserves closure under world evolution.

Semantic domains and interpretation of kinds. With much of the machinery defined, we can retrace the steps taken in the previous sections, and build appropriate universes of semantic types, computations, signatures and effects, and use those as interpretations of kinds in our calculus. In contrast to that definition, however, our universes are again indexed by the sets of free instance variables. We set:

$$\begin{aligned}
\mathbf{Comp}^X &\triangleq \mathbf{WIPred}^X(Y \mapsto \mathrm{Exp} \times \mathrm{Exp}_Y) \\
[\![\mathsf{T}]\!]^X \triangleq \mathbf{Type}^X &\triangleq \mathbf{WIPred}^X(Y \mapsto \mathrm{Val} \times \mathrm{Val}_Y) \\
[\![\mathsf{E}]\!]^X \triangleq \mathbf{Effect}^X &\triangleq \left\{ R \in \mathbf{WIPred}^X(Y \mapsto \mathrm{Exp} \times \mathrm{Exp}_Y \times \mathcal{P}_{\mathrm{fin}}(\mathrm{Inst}) \times \mathbf{Comp}^Y) \;\middle|\; |R|_{\mathrm{Inst}} \text{ is finite} \right\} \\
[\![\mathsf{S}]\!]^X \triangleq \mathbf{Sig}^X &\triangleq \mathbf{WIPred}^X(Y \mapsto \mathrm{Val} \times \mathrm{Val}_Y \times \mathbf{Type}^Y) \\
\mathfrak{I}^X &\triangleq \mathbf{Sig}^X \times \mathrm{Inst},
\end{aligned}$$

again defining semantic instances as a package that contains a semantic signature — although this time, only a single runtime instance is required, since only one of the semantics uses them. Note that our relations work across semantics: we have closed terms of the *instance* semantics on the left, and terms of the open semantics that are closed in term variables but that can contain free instance variables on the right-hand side. Finally, note that we use semantic computations and types, respectively, as parts of the family in the definition of semantic effects and signatures. This is well-defined, as world-indexed predicates behave functorially with respect to our maps, with the action on maps given by the $\Uparrow$ operator.

[5] This is a technical requirement; all the families we consider, in particular terms of the program syntax, are clearly well-behaved, so we only mention it explicitly where non-obvious.

Interpretation of types, signatures and effects. As before, we can now define an interpretation for types of kind κ that are well-formed in a type-variable context Δ and use free instance variables from a set X. Since our logical relation uses Kripke worlds, we need it to be parameterised by such a world — particularly, one indexed by X. All the other elements of our interpretation need to be taken at this world, which, intuitively, describes the evolution of variables from X to its codomain. Thus, we need the interpretation of Δ to be parameterised by a set of type variables, giving us $\delta \in [\![\Delta]\!]^W \iff \forall \alpha.\, \delta(\alpha) \in [\![\Delta(\alpha)]\!]^W$. This leads to the following type of the interpretation map:

$$[\![-]\!] : \Pi_{(W,\rho)\in \mathbf{World}^X} [\![\Delta]\!]^W \to (X \to \mathfrak{V}^W) \to [\![\kappa]\!]^W(\mathrm{id}).$$

There are some things to note about this type: first, we can clearly see the distinction between the variables that may appear in the type (given by X), and those that may appear in the terms in our interpretation (given by W). This is particularly visible in the map that gives interpretation to variables in X as semantic instances over W. Second, the result of our interpretation is not the appropriate semantic space, but rather its underlying predicate. This is largely a matter of taste, but we believe that it results in a cleaner presentation, with less pollution caused by having to consider multiple worlds.

The definition itself, presented in Figure 8, is very reminiscent of the one we considered in previous section, albeit with additional elements. Readers familiar with Kripke-style logical relations will not be surprised that the interpretation of the arrow type needs to be explicitly closed under future worlds — this is due to a contravariant occurrence of the world in the interpretation of the argument type. Note, however, that since δ and ϑ are themselves world-indexed, we need to shift them to the new world: thus we interpret both the argument and result types in the world $\rho' \circ \rho$ and use $\delta \Uparrow \rho'$ and $\vartheta \Uparrow \rho'$ as the appropriate substitutions. Moreover, this evolution of the world needs to be replicated at the term level, leading to the application of ρ' to v_2. This is a pattern repeated throughout the definition. The general shape of the interpretation of the universal quantifier is standard. However, in contrast to most Kripke-style logical relations, we require an explicit closure under future worlds. This is caused by the fact that even the interpretation of kinds is indexed by a set of instance variables, and we need to allow for the evolution of the world, even here. Leaving the instance quantification aside for the moment, we note that the interpretation of effects is mostly carried over from the relation for the generative semantics: they still behave very much set-like. The only important difference is the treatment of instance variables: for the open expressions we now use the world ρ to give the appropriate substitution for a — on the generative side, we pick the runtime instance provided by ϑ, as before. Finally, for the effect signatures we again match the interpretation we have seen before, the difference being that we now need to construct a semantic type (and thus, a world-indexed family of predicates) from the interpretation of τ_2 — which is the result of our choice of the type of the interpretation map, and poses no technical problems.

Let us now return to the instance quantifier. As before, we need to quantify over the future world to allow for evolution. However, this time the situation is complicated by the fact that τ has an additional free instance variable: a, which can evolve to end up within the set W' — or a globally free variable. Thus, we pick an element $w \in W'^{\perp}$, where the choice of bottom corresponds to the empty partial map from $W' \uplus \{a\}$ to W', and thus to setting a as globally free. From this point on, the definition matches the one for the generative semantics: we take a label l and the semantic instance composed of l and the interpretation of σ in the world $\rho' \circ \rho$ (since a is not free in σ), again, building an appropriate world-indexed predicate. Now it suffices to apply v_1 to l, move v_2 forward by ρ', and apply it to a moved forward to the set W' by the chosen element w. Note that if $w = \perp$, then we keep a as the chosen variable — but its status changes to globally free, and thus never handled by the evaluation context. This pair of terms has to belong to the interpretation of τ at a world that moves $X \uplus \{a\}$ to W', given by $(\rho' \circ \rho)[a \mapsto w]$ and appropriately shifted δ and ϑ.

$$
\begin{aligned}
[\![\alpha]\!]^{\delta,\vartheta}_{\rho} &\triangleq \delta(\alpha)(\mathrm{id}) \\
[\![\mathsf{unit}]\!]^{\delta,\vartheta}_{\rho} &\triangleq \{((),())\} \\
[\![\tau_1 \to_{\varepsilon} \tau_2]\!]^{\delta,\vartheta}_{\rho} &\triangleq \left\{(v_1,v_2) \;\middle|\; \forall \rho'.\, \forall (u_1,u_2) \in [\![\tau_1]\!]^{\delta \Uparrow \rho', \vartheta \Uparrow \rho'}_{\rho' \circ \rho}.\, (v_1\ u_1, \rho'(v_2)\ u_2) \in \mathcal{E}[\![\tau_2 \;/\; \varepsilon]\!]^{\delta \Uparrow \rho', \vartheta \Uparrow \rho'}_{\rho' \circ \rho} \right\} \\
[\![\forall\, \alpha :: \kappa.\, \tau]\!]^{\delta,\vartheta}_{\rho} &\triangleq \left\{(v_1,v_2) \;\middle|\; \forall \rho'.\, \forall \mu \in [\![\kappa]\!]^{\mathrm{cod}(\rho')}.\, (v_1\ *, \rho'(v_2)\ *) \in \mathcal{E}[\![\tau \;/\; \iota]\!]^{(\delta \Uparrow \rho')[\alpha \mapsto \mu], \vartheta \Uparrow \rho'}_{\rho' \circ \rho} \right\} \\
[\![\forall\, a : \sigma.\, \tau]\!]^{\delta,\vartheta}_{\rho} &\triangleq \left\{(v_1,v_2) \;\middle|\; \begin{array}{c} \forall W', \rho' : W \rightharpoonup W', w : W'^{\perp}.\, \forall l, I.\, I = (\rho' \mapsto [\![\sigma]\!]^{\delta \Uparrow \rho', \vartheta \Uparrow \rho'}_{\rho' \circ \rho}, l) \Rightarrow \\ (v_1\ l, \rho'(v_2)\ a\{w \,/\, a\}) \in \mathcal{E}[\![\tau \;/\; \iota]\!]^{\delta \Uparrow \rho', \vartheta[a \mapsto I] \Uparrow \rho'}_{(\rho' \circ \rho)[a \mapsto w]} \end{array} \right\} \\
[\![\iota]\!]^{\delta,\vartheta}_{\rho} &\triangleq \emptyset \\
[\![\varepsilon_1 \cdot \varepsilon_2]\!]^{\delta,\vartheta}_{\rho} &\triangleq [\![\varepsilon_1]\!]^{\delta,\vartheta}_{\rho} \cup [\![\varepsilon_2]\!]^{\delta,\vartheta}_{\rho} \\
[\![a]\!]^{\delta,\vartheta}_{\rho} &\triangleq \left\{(\mathbf{do}_l\ v_1, \mathbf{do}_{\rho(a)}\ v_2, \{l\}, \mu) \;\middle|\; \exists \nu.\, \vartheta(a) = (\nu, l) \wedge (v_1, v_2, \mu) \in \nu(\mathrm{id}) \right\} \\
[\![\tau_1 \Rightarrow \tau_2]\!]^{\delta,\vartheta}_{\rho} &\triangleq \left\{(v_1, v_2, \rho' \mapsto [\![\tau_2]\!]^{\delta \Uparrow \rho', \vartheta \Uparrow \rho'}_{\rho' \circ \rho}) \;\middle|\; (v_1,v_2) \in [\![\tau_1]\!]^{\delta,\vartheta}_{\rho} \right\}
\end{aligned}
$$

Fig. 8. Interpretation of types, effects, and signatures in the inter-semantic logical relation. We assume $\rho : X \rightharpoonup W$ throughout.

Closure operators and the logical relation. We can now define the closure operators, analogous to those defined for the generative semantics; these are presented in Figure 9. The closure for expressions is a standard biorthogonal definition, although expanded with the explicit future-world closure we have seen in the interpretation of types. Note that since the observation relation works on complete — and thus, closed — programs, it is not indexed with any particular world. The relation for evaluation contexts follows the pattern of Biernacki et al., with two observations, one for values, and one for control-stuck expressions. The final two operators are more involved. In the relation for control-stuck expressions, recall that the resumption R_2 may well bind some variables *in addition* to the set W given by the world; let's call this set X — and thus have $R_2 \in \mathrm{RCont}^X_W$. Thus, we need to interpret the effect ε in a *larger* world with a codomain of $W \uplus X$, which we can achieve by composing ρ with an inclusion map: we denote this composition with ρ_R. We then need to take some map $\rho' : W \rightharpoonup W'$ to ensure future-world closure — but the response type μ is defined at $W \uplus X$. Thus, we take $\rho'_R : W \uplus X \rightharpoonup W' \uplus X$ to be the obvious lifting of ρ'. This allows us to take expressions related by μ, which we can now plug back into resumptions. As before, the later operator ensures the entire definition is well-formed.

Before turning to the relation for handlers, let us consider the observation relation, as it is not the standard notion of approximation as used in step-indexed relations. Usually, one builds an asymmetric observation relation, which gives rise to a logical approximation relation — like we did for the generative semantics. If one wants to reason about equivalence, it then suffices to take a symmetric closure of the logical approximation. However, this is not the case here — as our relation works across two semantics. What's more, at the present we merely want to show that the two semantics are equivalent. Thus, we introduce a *weaker* relation, that requires that the two programs proceed in lock-step (at least for the reductions that actually require step-indexing).

$$
\begin{aligned}
(v_1, v_2) \in \mathbf{Arrow}(\mu, \nu) &\iff \forall \rho.\, \forall (u_1, u_2) \in \mu(\rho).\, (v_1\ u_1, \rho(v_2)\ u_2) \in \nu(\rho) \\
(e_1, e_2) \in \mathcal{E}[\![\tau \,/\, \varepsilon]\!]^{\delta,\vartheta}_{\rho} &\iff \forall \rho'.\, \forall (E_1, E_2) \in \mathcal{K}[\![\tau \,/\, \varepsilon]\!]^{\delta \Uparrow \rho', \vartheta \Uparrow \rho'}_{\rho' \circ \rho}.\, (E_1[e_1], E_2[e_2]) \in \mathbf{Obs} \\
(E_1, E_2) \in \mathcal{K}[\![\tau \,/\, \varepsilon]\!]^{\delta,\vartheta}_{\rho} &\iff \forall (v_1, v_2) \in [\![\tau]\!]^{\delta,\vartheta}_{\rho}.\, (E_1[v_1], E_2[v_2]) \in \mathbf{Obs} \\
&\qquad \wedge\, \forall (e_1, e_2) \in \mathcal{S}[\![\tau \,/\, \varepsilon]\!]^{\delta,\vartheta}_{\rho}.\, (E_1[e_1], E_2[e_2]) \in \mathbf{Obs} \\
(R_1[e_1], R_2[e_2]) \in \mathcal{S}[\![\tau \,/\, \varepsilon]\!]^{\delta,\vartheta}_{\rho} &\iff \exists L, \mu.\, (e_1, e_2, L, \mu) \in [\![\varepsilon]\!]^{\delta \Uparrow \rho_R, \vartheta \Uparrow \rho_R}_{\rho_R \circ \rho} \wedge (\forall l \in L.\ \mathrm{free}(l, R_1)) \\
&\qquad \wedge\, \forall \rho'.\, \forall (e_1', e_2') \in \mu(\rho_R').\, (R_1[e_1'], \rho'(R_2)[e_2']) \in \triangleright \mathcal{E}[\![\tau \,/\, \varepsilon]\!]^{\delta \Uparrow \rho', \vartheta \Uparrow \rho'}_{\rho' \circ \rho} \\
(x, k.\ e_1, x, k.\ e_2) \in \mathcal{H}[\![\sigma \triangleright \tau \,/\, \varepsilon]\!]^{\delta,\vartheta}_{\rho} &\iff \forall \rho', \rho^{\uparrow}, \rho^{\downarrow}.\, \rho^{\downarrow} \circ \rho^{\uparrow} = \mathrm{id} \Rightarrow \\
&\qquad \forall (u_1, u_2, \mu) \in [\![\sigma]\!]^{\delta \Uparrow \rho^{\uparrow} \circ \rho', \vartheta \Uparrow \rho^{\uparrow} \circ \rho'}_{\rho^{\uparrow} \circ \rho' \circ \rho}.\, \exists \mu'. \mu = \mu' \Uparrow \rho^{\uparrow} \\
&\qquad \wedge\, \forall (v_1, v_2) \in \mathbf{Arrow}(\mu', \rho'' \mapsto \mathcal{E}[\![\tau \,/\, \varepsilon]\!]^{\delta \Uparrow \rho'' \circ \rho', \vartheta \Uparrow \rho'' \circ \rho'}_{\rho'' \circ \rho' \circ \rho}). \\
&\qquad (e_1\{u_1 \,/\, x\}\{v_1 \,/\, y\}, \rho'(e_2)\{\rho^{\downarrow}(u_2) \,/\, x\}\{v_2 \,/\, y\}) \in \mathcal{E}[\![\tau \,/\, \varepsilon]\!]^{\delta \Uparrow \rho', \vartheta \Uparrow \rho'}_{\rho' \circ \rho} \\
(e_1, e_2) \in \mathbf{Obs} &\iff (e_1 = () \wedge e_2 \to^* ()) \vee (e_1 \to^* () \wedge e_2 = ()) \\
&\qquad \vee \left(\exists e_1', e_2'.\, e_1 \to e_1' \wedge e_2 \to e_2' \wedge (e_2, e_2') \in \triangleright \mathbf{Obs}\right)
\end{aligned}
$$

Fig. 9. Closure operators for expressions, evaluation contexts, control-stuck expressions, and handlers. In the relation for control-stuck expressions ρ_R denotes the inclusion map induced by R_2, and ρ_R' — the lifting of ρ' by this inclusion.

This is sufficient for our purpose, as the two semantics can reduce on both sides save for the label-generating reduction in the generative semantics, which is simple from the reduction behaviour point of view. If, on the other hand, we took the traditional approach, we would have to define virtually the entire relation twice, for very little gain, at least given our goal.

Let us finally consider the relation for handlers. Note, first, that until now there was no point where some technical obstacle could prevent us from extending the calculus — and the relation — with signature polymorphism. The relation for handlers is the final place where something could prevent us from successfully relating polymorphic signatures: and indeed, this is the case. The problem stems from a similar issue as the one we explored with control-stuck terms. Now, the arguments of the handler, u_1 and u_2, passed by the **do** operation, come from within some resumption — and thus, from a *larger* world. Moreover, this is also the case for the response type μ. However, the reified resumptions, v_1 and v_2, need to be interpreted at the smaller world, even though their arguments should be related by μ. How to square this circle? We require that u_1, u_2 and μ come from a world given by a composition of maps, the last of which is *invertible* on the left — which matches the intuition that it stems from an *inclusion* induced by the resumption. Then, for the handlers to be related, we require that μ is equal to some semantic type moved forward by $\rho^{\uparrow}$. This is the case if σ is a simple signature (of the form $\tau_1 \Rightarrow \tau_2$), but not if it were polymorphic — as in that case we are free to use the variables introduced by $\rho^{\uparrow}$ in the instantiation of the quantifier. Once we have μ' on hand, we can assume that the reified resumptions are related at the appropriate type — for the sake of clarity, we introduce an auxiliary definition of "semantic arrows" — and prove that the bodies of the handlers are related, given the related arguments and resumptions.

Like in the simpler relation, we need to give the interpretations of the term- and instance-variable contexts, which differs little from the previous case. The only distinction is the fact that we only

```
signature State (X : Type) =
| put : X    => Unit
| get : Unit => X

let update f = put (f (get ()))
```

```
handle ‘x in handle ‘y in
    put ‘y False;
    update ‘x (fn s => if get ‘y () then s - 6 else s + 29);
    get ‘x ()
with hState True with hState 13
```

Fig. 10. Multiple state cells using lexically scoped handlers

need one substitution of runtime instances for instance variables, for the generative component of our relation. We get:

$$[\![\Theta]\!]^{\delta}_{\rho} \triangleq \left\{ (\vartheta, \eta) \;\middle|\; \forall a \in \mathrm{dom}(\Theta).\ \vartheta(a) = ([\![\Theta(a)]\!]^{\delta,\vartheta}_{\rho}, \eta(a)) \right\}$$

$$[\![\Gamma]\!]^{\delta,\vartheta}_{\rho} \triangleq \left\{ (\gamma_1, \gamma_2) \;\middle|\; \forall x \in \mathrm{dom}(\Gamma).\ (\gamma_1(x), \gamma_2(x)) \in [\![\Gamma(x)]\!]^{\delta,\vartheta}_{\rho} \right\},$$

and finally we can define the logical relation (as previously, we only give the relation for the expressions, eliding values and handlers):

$$\begin{aligned} \Delta; \Theta; \Gamma \vDash e_1 \approx e_2 : \tau \;/\; \varepsilon \iff \forall (W, \rho) \in \mathbf{World}^{\mathrm{dom}(\Theta)}.\ & \\ \forall \delta \in [\![\Delta]\!]^{W}, (\vartheta, \eta) \in [\![\Theta]\!]^{\delta}_{\rho}, (\gamma_1, \gamma_2) \in [\![\Gamma]\!]^{\delta,\vartheta}_{\rho}.\ & \\ (\eta(\gamma_1(e_1)), \gamma_2(e_2)) \in \mathcal{E}[\![\tau \;/\; \varepsilon]\!]^{\delta,\vartheta}_{\rho} & \end{aligned}$$

We can now show that our relation is a compatible congruence, getting the fundamental theorem of logical relations, and, as its corollary (together with adequacy of our observation), the equivalence of the two semantics:

Theorem 4 (Fundamental). *Any well-typed source expression e is logically related with itself across the two semantics, i.e., if $\Delta; \Theta; \Gamma \vdash e : \tau \;/\; \varepsilon$, then we have $\Delta; \Theta; \Gamma \vDash e \approx e : \tau \;/\; \varepsilon$.*

Theorem 5 (Equivalence). *For any well-typed, closed, pure program e, its evaluation in open semantics terminates iff its evaluation in the generative semantics terminates, i.e., if $\cdot; \cdot; \cdot \vdash e : \tau \;/\; \iota$, then $(\exists v.\ e \to_o^* v) \iff (\exists v.\ e \to_g^* v)$.*

7 PROGRAMMING WITH SCOPED EFFECTS

Having established the sensible behaviour of our generative semantics, we have implemented an experimental programming language based on our calculus. Thus, we can offer some preliminary observations regarding the practicality of our approach. In this section, we consider some simple examples that highlight how programming with lexically scoped effects can be cleaner and more modular, if somewhat more restrictive than the full power of dynamically scoped effects.

We begin with a simple example that uses two cells of mutable state, presented in Figure 10. The definition of the signature for state is standard: it consists of two operations, put and get, respectively setting and getting the state cell. We then define a function update, which calls its argument with the current value of the state cell, and stores the result. Note that since this function only uses a single instance of state, we never need to mention it: it is clear from the context that update should introduce an implicit instance of the state effect, and call both get and put with this instance. The type of update could be given, in the syntax of our calculus, as $\forall\, \alpha :: \mathsf{T}.\ \forall\, \beta :: \mathsf{E}.\ \forall\, a : \mathrm{State}\ \alpha.\ (\alpha \to_{\beta} \alpha) \to_{a \cdot \beta} \mathsf{unit}$. Note that the instantiation of β, and thus the potential interaction between the modified cell of state and the function passed as an argument is left to the callee. In the client code snippet that follows, we introduce two instance variables, ‘x and ‘y (a backtick before an identifier makes it an instance variable), for two cells of state, and

```
signature rec CMT (E : Effect) =
| fork  : (forall ('cmt : CMT E),
             Unit ->['cmt, E] Unit) => Unit
| yield : Unit => Unit

data rec Process (E : Effect) =
| Proc of (Unit -> Queue (Process E) ->[E] Unit)

let continue pq =
  match Queue.pop pq with
  | None => ()
  | Some(pq, Proc proc) => proc () pq
  end
```

```
let hCMT () =
  let rec hCMT_rec pq =
    handler
    | fork proc => fn pq =>
      handle proc () with
        hCMT_rec (Queue.push (Proc resume) pq)
    | yield ()  => fn pq =>
        continue (Queue.push (Proc resume) pq)
    | return () => fn pq => continue pq
    | finally f => f pq
    end
  in
  hCMT_rec Queue.empty
```

Fig. 11. Simple cooperative multithreading via effect handlers

use both operations of the state effect, and update – but this time necessarily giving the instance arguments explicitly. Since in this case, under a pair of handlers, the instance used would not necessarily be clear from context, our typechecking algorithm would protest if we omitted the instance arguments.[6] Note also that the argument passed to the update function itself uses a state – but we can be certain that the two cells do not interfere by virtue of their different names bound by the two handlers: thus, with the standard definition of handlers for state, hState, discussed in Section 1, the snippet evaluates to 42.

As a more sophisticated example, let us consider a simple cooperative concurrency effect, adapted from [Convent et al. 2019], presented in Figure 11. The signature is parameterised by the type of effects that the concurrent processes can raise (which allows for communication channels), and consists of two operations: fork, which takes a thunk (itself parameterised with an instance of the concurrency effect) as an argument, and yield, which allows the scheduler to interfere. The scheduler is implemented using a queue of suspended processes, which are started in a breadth-first pattern and stopped at yield signals. Note that the somewhat sophisticated handler is not annotated with signatures, since, once again, the single instance considered in its definition is clear from context. This form of concurrency can be used to implement sophisticated patterns when used in conjunction with other effects: Convent et al. show an implementation of communicating actors, and one could easily encode groups of processes that can be terminated simultaneously and other similar patterns.

8 RELATED WORK

8.1 Managing Effects in Different Settings

The problem of managing effects is not specific to handlers of algebraic effects. It arises whenever one deals with user-defined effects or any form of instances of predefined effects. A well-known example are generative polymorphic exceptions in SML [Milner et al. 1997]. To ensure type safety, each time a declaration of an exception is evaluated, it actually gives a new type of exceptions, quite similarly to our generative semantics.

[6] A sufficiently sophisticated algorithm might be able to determine that the two cells of state have incompatible types, and the instances could potentially be deduced automatically. Part of the appeal of the instance approach is that irrespective of the algorithm, this remains a mere optimisation: we can always fall back to passing the instances explicitly when this would increase readability or the type-checker cannot ensure that the instances are clear from context.

The problem of managing effects was often taken into account when considering type-and-effect systems for particular built-in effects [Lucassen and Gifford 1988; Talpin and Jouvelot 1994]. One example is region-based mutable store, in which one wants to be able to dynamically allocate, deallocate, and refer to regions, while statically keeping track of their liveness via a type-and-effect system. The fact that liveness can be tracked via lexically scoped region variables is explored, for instance, by Birkedal et al. [1996].

Another approach to effects is given by monads, first used to give a denotational semantics of effects by Moggi [1991], and later popularised as a programming abstraction by Wadler [1990; 1992]. The usual way to achieve modularity in programming with monads is rather ad hoc by using monad transformers [Liang et al. 1995]. One way to think about a stack of monad transformers is that it corresponds to a row of effects in a row-based type-and-effect system. The n-th layer of the stack can be accessed via an n-fold composition of the `lift` function. One can also access the first "instance" of a particular effect using type classes that couple particular operations and particular monads using the *á la carte* technique [Swierstra 2008]; see also [Gibbons and Hinze 2011]. To manage multiple occurrences of the same monad on the stack, Snyder and Alexander [2010] introduced a technique based on global names implemented with phantom types, while Schrijvers and Oliveira [2011] introduced monad views that allow functionality similar to effect subtyping. Recently, Devriese [2019] proposed a technique of managing effects in Haskell based on effect polymorphism and explicit type-class dictionary applications to distinguished between instances of effects. We leave a more detailed comparison with this approach as future work.

8.2 Managing Algebraic Effects

Row-based type-and-effect systems distinguish "instances" of effects by their position in the effect row. One way to manage them is to use coercions [Biernacki et al. 2019] or adaptors [Convent et al. 2019], which, however, place the burden of keeping track of which operation is meant for which position, as well as manually permuting the row to agree the effects of two subcomputations, on the programmer. Simply using names, as in our approach, makes the code much more readable and modular. It is also more amenable for efficient implementation, as coercions/adaptors have nontrivial computational content. On the flip side, our instances are a bit less expressive; for example, it is not immediately clear how to use them to implement the encoding of state via reader and writer, which relies on dynamic switching of handlers [Biernacki et al. 2018].

We also note that polymorphism in ML-style type systems, especially in the context of value and purity restrictions, is an active area of research [Kammar and Pretnar 2017; Sekiyama and Igarashi 2019]. In our calculus, we employ the purity restriction, as described in Section 3.

Except for the languages with built-in effect handlers mentioned in the introduction, we should also note an implementation of a DSL embedded in Idris by Brady [2013], which supports effect tracking through dependent types. In order to make it possible to program consciously with different instances of the same effect, the DSL allows for static labelling of the effects listed in the type of a given expression. Such a solution is applicable in simple cases, but suffers from the problems described in Section 2.

8.3 Lexically Scoped Effects

As mentioned in the introduction, lexically scoped effects are not a novel idea in general, but they have been employed to effect handlers only recently. The examples that we are aware of are a Java library by Brachthäuser et al. [2018] and the calculus for effect tunnelling by Zhang and Myers [2019]. Both these solutions are given in a "handler-passing" style, in which a handler lives inside the delimited context, making these calculi similar in feeling to the shift_0/reset control operators rather than effect handlers as usually presented in the literature.

Zhang and Myers solve a problem of parametricity of effect polymorphism [Biernacki et al. 2018, Example 2.4]. Our results subsume their solution, and the general technical ideas in both papers are actually very much alike: to use lexically scoped variables to couple operations and handlers. The difference is that Zhang and Myers's calculus (equipped with an operational semantics akin to our open semantics) has a rather restricted type system. In particular, their effect signatures can be neither effectful nor polymorphic. They also construct a logical relation for their calculus, which is much simpler than the one we introduce to deal with the open semantics, but their relation seems to rely on the type restrictions in a nontrivial manner. Thus, it does not seem to be extendable to more advanced but practical language features. In the Coq formalisation of their results, however, Zhang and Myers use a semantics that is closely related to our generative semantics: as discussed in Sections 4 and 5, this leads to a discrepancy between the semantics formalised in the development and the one presented in the paper in the untyped case. Since the logical relation works over untyped terms, we fear that this means the formalisation is not adequate with respect to the presentation in the paper – although by itself this does not invalidate the results as presented.

8.4 Logical Relations for Generative Semantics

As already mentioned, generative semantics has found applications in the study of polymorphic exceptions and dynamically allocated mutable references. More recently, generative semantics has been defined, and investigated, for polymorphic blame calculi [Ahmed et al. 2017; Toro et al. 2019] that marry polymorphic static typing with dynamic typing. In such semantics it is type instances that are dynamically generated and have to be kept track of in the logical relation that is meant to ensure parametricity of the type system. Such relations, unsurprisingly, are given in the Kripke style and their overall underlying structure resembles that of the logical relations for ML-style mutable references [Ahmed et al. 2010, 2009] as well as of the one we present in Section 6.1.

9 CONCLUSION

In this paper, we propose a solution to the practical problem of managing instances in languages with effect handlers. The advantage of using lexically scoped effects is that the resulting calculus is simple and intuitive on the level of syntax and types – we dare say simpler than calculi with row-based type systems. Obstacles begin with considerations on possible operational semantics, and the main theme of this paper is exploring the space of design choices, inhabited by (perhaps among others) the open and generative semantics. Importantly, we try to take into account both the theoretical properties and practical aspects of programming with lexically scoped instances.

We put much emphasis on understanding parametric polymorphism in the presence of effects. Polymorphism is known to be a major troublemaker, hence the need for the value restriction, generative exceptions, and similar restraints in non-pure functional languages, which might not be obvious to the programmer at first, but are necessary to maintain type safety. We hope that our novel approach to Kripke-style logical relations will allow us to reason about the marriage of polymorphism and effects in other contexts as well. An obvious example would be reasoning about region-based store in which the name or address of a deallocated region is not presumed definitively dead, but can be reused, just like an instance variable name in our open semantics.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for helpful comments regarding the presentation and related work. We are also grateful to the participants of the NII Shonan Meeting "Programming and reasoning with algebraic effects" and of the ChoCoLa seminar at ENS Lyons for useful discussions that led to this work. This work was supported by the National Science Centre, Poland under grant no. 2018/31/D/ST6/03951 and grant no. 2016/23/D/ST6/01387.

REFERENCES

Amal Ahmed, Andrew W. Appel, Christopher D. Richards, Kedar N. Swadi, Gang Tan, and Daniel C. Wang. 2010. Semantic foundations for typed assembly languages. *ACM Trans. Program. Lang. Syst.* 32, 3 (2010), 7:1–7:67. https://doi.org/10.1145/1709093.1709094

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 Ahmed, Dustin Jamner, Jeremy G. Siek, and Philip Wadler. 2017. Theorems for free for free: parametricity, with and without types. *PACMPL* 1, ICFP (2017), 39:1–39:28. https://doi.org/10.1145/3110283

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

Andrew W. Appel and David A. McAllester. 2001. An indexed model of recursive types for foundational proof-carrying code. *ACM Trans. Program. Lang. Syst.* 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*. 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, 4:9 (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. *J. Log. Algebr. Meth. Program.* 84, 1 (2015), 108–123. https://doi.org/10.1016/j.jlamp.2014.02.001

Nick Benton and Andrew Kennedy. 2001. Exceptional Syntax. *Journal of Functional Programming* 11, 4 (2001), 395–410. https://doi.org/10.1017/S0956796801004099

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

Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. 2019. Abstracting Algebraic Effects. *PACMPL* 3, POPL, Article 6 (2019), 28 pages. https://doi.org/10.1145/3290319

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

Lars Birkedal, Mads Tofte, and Magnus Vejlstrup. 1996. From Region Inference to von Neumann Machines via Region Representation Inference. In *Conference Record of POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, St. Petersburg Beach, Florida, USA, January 21-24, 1996*, Hans-Juergen Boehm and Guy L. Steele Jr. (Eds.). ACM Press, 171–183. https://doi.org/10.1145/237721.237771

Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. 2018. Effect handlers for the masses. *PACMPL* 2, OOPSLA (2018), 111:1–111:27. https://doi.org/10.1145/3276481

Edwin Brady. 2013. Programming and reasoning with algebraic effects and dependent types. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, Greg Morrisett and Tarmo Uustalu (Eds.). ACM, 133–144. https://doi.org/10.1145/2500365.2500581

Manuel M. T. Chakravarty, Zhenjiang Hu, and Olivier Danvy (Eds.). 2011. *Proceeding of the 16th ACM SIGPLAN International Conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011*. ACM.

Lukas Convent, Sam Lindley, Conor McBride, and Craig McLaughlin. 2019. Doo bee doo bee doo. (2019). Draft.

Dominique Devriese. 2019. Modular Effects in Haskell Through Effect Polymorphism and Explicit Dictionary Applications: A New Approach and the μVeriFast Verifier As a Case Study. In *Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell (Haskell 2019)*. ACM, New York, NY, USA, 1–14. https://doi.org/10.1145/3331545.3342589

Stephen Dolan, Leo White, KC Sivaramakrishnan, Jeremy Yallop, and Anil Madhavapeddy. 2015. Effective Concurrency through Algebraic Effects. (2015). OCaml Users and Developers Workshop, September 2015, Vancouver, Canada.

Derek Dreyer, Amal Ahmed, and Lars Birkedal. 2011. Logical Step-Indexed Logical Relations. *Logical Methods in Computer Science* 7, 2 (2011). https://doi.org/10.2168/LMCS-7(2:16)2011

Yannick Forster, Ohad Kammar, Sam Lindley, and Matija Pretnar. 2017. On the expressive power of user-defined effects: effect handlers, monadic reflection, delimited control. *PACMPL* 1, ICFP (2017), 13:1–13:29. https://doi.org/10.1145/3110257

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*. 148–161. https://doi.org/10.1007/3-540-39185-1_9

Jeremy Gibbons and Ralf Hinze. 2011. Just do it: simple monadic equational reasoning, See [Chakravarty et al. 2011], 2–14. https://doi.org/10.1145/2034773.2034777

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

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

Daan Leijen. 2014. Koka: Programming with Row Polymorphic Effect Types. In *Proceedings 5th Workshop on Mathematically Structured Functional Programming, MSFP@ETAPS 2014, Grenoble, France, 12 April 2014.* 100–126. https://doi.org/10.4204/EPTCS.153.8

Daan Leijen. 2017a. Implementing Algebraic Effects in C - "Monads for Free in C". In *Programming Languages and Systems - 15th Asian Symposium, APLAS 2017, Suzhou, China, November 27-29, 2017, Proceedings (Lecture Notes in Computer Science)*, Bor-Yuh Evan Chang (Ed.), Vol. 10695. Springer, 339–363. https://doi.org/10.1007/978-3-319-71237-6_17

Daan Leijen. 2017b. Structured asynchrony with algebraic effects. In *Proceedings of the 2nd International Workshop on Type-Driven Development, TyDe@ICFP 2017, Oxford, UK, September 2017.*

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

Daan Leijen. 2018. *Algebraic Effect Handlers with Resources and Deep Finalization.* Technical Report MSR-TR-2018-10. Microsoft Research.

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

Sheng Liang, Paul Hudak, and Mark P. Jones. 1995. Monad Transformers and Modular Interpreters. In *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23-25, 1995*, Ron K. Cytron and Peter Lee (Eds.). ACM Press, 333–343. https://doi.org/10.1145/199448.199528

John M. Lucassen and David K. Gifford. 1988. Polymorphic Effect Systems. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, San Diego, California, USA, January 10-13, 1988.* 47–57. https://doi.org/10.1145/73560.73564

Robin Milner, Mads Tofte, and David Macqueen. 1997. *The Definition of Standard ML.* MIT Press, Cambridge, MA, USA.

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

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

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. http://www.inf.ed.ac.uk/~stark/operfl.html

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

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

Matija Pretnar. 2015. An Introduction to Algebraic Effects and Handlers. Invited tutorial paper. *Electr. Notes Theor. Comput. Sci.* 319 (2015), 19–35. https://doi.org/10.1016/j.entcs.2015.12.003

Tom Schrijvers and Bruno C. d. S. Oliveira. 2011. Monads, zippers and views: virtualizing the monad stack, See [Chakravarty et al. 2011], 32–44. https://doi.org/10.1145/2034773.2034781

Taro Sekiyama and Atsushi Igarashi. 2019. Handling Polymorphic Algebraic Effects. In *Programming Languages and Systems - 28th European Symposium on Programming, ESOP 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings (Lecture Notes in Computer Science)*, Luís Caires (Ed.), Vol. 11423. Springer, 353–380. https://doi.org/10.1007/978-3-030-17184-1_13

Mark Snyder and Perry Alexander. 2010. Monad Factory: Type-Indexed Monads. In *Trends in Functional Programming - 11th International Symposium, TFP 2010, Norman, OK, USA, May 17-19, 2010. Revised Selected Papers (Lecture Notes in Computer Science)*, Rex L. Page, Zoltán Horváth, and Viktória Zsók (Eds.), Vol. 6546. Springer, 198–213. https://doi.org/10.1007/978-3-642-22941-1_13

Wouter Swierstra. 2008. Data types à la carte. *J. Funct. Program.* 18, 4 (2008), 423–436. https://doi.org/10.1017/S0956796808006758

Jean-Pierre Talpin and Pierre Jouvelot. 1994. The Type and Effect Discipline. *Inf. Comput.* 111, 2 (1994), 245–296. https://doi.org/10.1006/inco.1994.1046

Matías Toro, Elizabeth Labrada, and Éric Tanter. 2019. Gradual parametricity, revisited. *PACMPL* 3, POPL (2019), 17:1–17:30. https://doi.org/10.1145/3290330

Philip Wadler. 1990. Comprehending Monads. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming (LFP '90)*. ACM, 61–78. https://doi.org/10.1145/91556.91592

Philip Wadler. 1992. Monads for functional programming. In *Proceedings of the Marktoberdorf Summer School on Program Design Calculi*.

Yizhou Zhang and Andrew C. Myers. 2019. Abstraction-safe effect handlers via tunneling. *PACMPL* 3, POPL (2019), 5:1–5:29. https://doi.org/10.1145/3290318