Abstracting Algebraic Effects

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

Proposed originally by Plotkin and Pretnar, algebraic effects and their handlers are a leading-edge approach to computational effects: exceptions, mutable state, nondeterminism, and such. Appreciated for their elegance and expressiveness, they are now progressing into mainstream functional programming languages. In this paper, we introduce and examine programming language constructs that back adoption of programming with algebraic effects on a larger scale in a modular fashion by providing mechanisms for abstraction. We propose two such mechanisms: existential effects (which hide the details of a particular effect from the user) and local effects (which guarantee that no code coming from the outside can interfere with a given effect). The main technical difficulty arises from the dynamic nature of coupling an effectful operation with the right handler during execution, but, as we show in this paper, a carefully designed type system can ensure that this will not break the abstraction. Our main contribution is a novel calculus for algebraic effects and handlers, called λ^{HEL}, equipped with local and existential algebraic effects, in which the dynamic nature of handlers is kept in check by typed runtime coercions. As a proof of concept, we present an experimental programming language based on our calculus, which provides strong abstraction mechanisms via an ML-style module system.

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

Additional Key Words and Phrases: algebraic effect, row polymorphism, existential type

ACM Reference Format:
Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. 2018. Abstracting Algebraic Effects. *Proc. ACM Program. Lang.* 1, 1, Article 1 (January 2018), 30 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The notion of algebraic effects first appeared in the works of Plotkin and Power [2004; 2001; 2002], who studied computational effects in terms of *operations*, such as `put` and `get` in programming with mutable state, or `throw` in programming with exceptions. More recently, Plotkin and Pretnar [2013] proposed a framework in which effects understood as sets of operations come in tandem with *handlers*: constructs that give semantics to effectful computations, generalizing the usual exception handlers. A clear advantage and a novel feature of this approach is a separation of syntax and semantics of effects, which turns out to be rather expressive and elegant in practical examples. Most notably, it allows for constructing computations that rely on multiple different effects at a time, and for precise control over how these effects interact. As a programming feature, algebraic effects and handlers appear in a number of experimental languages, such as Eff [Bauer and Pretnar 2015], Frank [Lindley et al. 2017], and Koka [Leijen 2014], or as extensions of existing languages [Hillerström and Lindley 2016; Kammar et al. 2013].

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.

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

In this paper, we tackle the issue of abstraction in languages with algebraic effects, in the sense provided, for instance, by a module system. In a large-scale system, modules serve to provide interfaces for abstract data types and hide the implementation details of associated functions and the concrete representation of data. This approach is crucial, as it can ensure non-interference of conceptually separate subparts of a large program, even when the implementation changes over time. At the same time, by providing abstract interfaces, modules empower the programmers to think and design at the higher levels of abstraction, which facilitates creating large systems.

In the same vein as the usual data abstraction, when designing a system in a language with algebraic effects, one might want to declare an effect abstract, to hide implementation details from the library's users and only expose a limited, abstract interface. This has been recognised in a recent technical report by Leijen 2018. We study a natural notion of exposing *abstract effects* through existential quantification, and explain that this issue turns out to be not as simple as hiding the definition of an effect from the user, since the execution of effectful programs rely on a dynamic process of matching an operation to the appropriate handler. This process, if implemented naively, can break the abstraction by "stealing" an effect from the inside of what was supposed to be a black box. We show how to avoid this by a rigorous type discipline and a novel construct, *effect coercions*, which extend a notion of coercions used by Saleh et al. in [2018] to coercions that have significance at runtime.

In Section 2, we introduce our key contribution: a core calculus of abstract algebraic effects, called λ^{HEL}. It is equipped with a polymorphic row-based type-and-effect system, in the style of the core languages of Koka [Leijen 2017] and Links [Hillerström and Lindley 2016]. As a novel feature, it allows for existential quantification over both types and (rows of) effects. It also provides the mentioned effect coercions and *local effects*, which allow the programmer to define an effect that is visible only within the scope of an expression. Operationally, λ^{HEL} is given a call-by-value reduction semantics in the style of Biernacki *et al.*'s [2018] $\lambda^{\mathsf{H/L}}$-calculus. In particular, an operation is matched to its handler by the n-freeness relation, which can be explicitly controlled using effect coercions. Indeed, $\lambda^{\mathsf{H/L}}$'s *lift* is one of the available effect coercions in λ^{HEL}.

As we believe, to thoroughly study this kind of abstraction, one needs to take into consideration the practical applicability of the constructs provided by the language, which can be assessed only by going through a number of examples of increasing complexity, which we discuss in Section 6. To ensure that λ^{HEL} is a reasonable core calculus for a programming language with abstract algebraic effects, we also provide an implementation: an experimental programming language called Helium, which provides abstraction for both types and effects via an ML-style module system. We discuss it briefly in Section 5.

We also look at another aspect of λ^{HEL} as a core calculus, namely, execution. The reduction semantics of λ^{HEL} is type-directed at certain points, and it relies on the complex n-freeness relation. Thus, as a step towards an efficient implementation, in Section 3, we show a lower-level language, which is no longer decorated with types, except for labels that assign operations and handlers to a particular effect. We define a translation from λ^{HEL} to the untyped calculus, and prove its correctness. In Section 4, we show a CEK-like abstract machine that executes programs in the untyped language. Indeed, this machine is used as the execution model in our Helium interpreter.

1.1 Concrete and Abstract Algebraic Effects

We now proceed to introduce some motivating examples of both concrete and abstract algebraic effects. In order to keep the presentation readable, we use syntactic sugar to make the expressions of our calculus look more familiar and omit any potential coercions where they could be easily inferred from the context. Also, note that the calculus we work with is pure, save for the algebraic effects, and thus any state-like behaviour would have to be implemented via an appropriate effect.

Algebraic Effects and Handlers. A simple effect can be defined as follows:

```
effect Reader A = { ask : Unit => A }
```

It defines an effect constructor called `Reader`, which is parameterized by a type `a`. The `Reader` effect consists of a single operation `ask`, which can be used similarly to a function `Unit -> a`. We can put an expression that uses the `ask` operation in a handler, which gives semantics to the effect. For example:

```
handle "Hello " ++ ask () ++ ". How are you doing, " ++ ask () ++ "?" with
| ask ()   => resume "Dave"
| return x => x
end
```

To evaluate the expression above, we first try to evaluate the expression in between the `handle` and `with` keywords. When we need a value of an application of `ask`, the handler takes over. Each time, it resumes with the string `"Dave"`, which means that it goes back to evaluating the expression, but with the string substituted for the particular occurrence of the operation. The `return` clause indicates that if no operation needs evaluating in the handled expression, that is, we handle a pure value, we simply use this value as the overall result of the handler.

In this case, the handled expression is given the type `String`, but it is also given a row of effects, `[Reader String]`. Such a row is a list of effects that can be invoked by a given expression. In this case, it has only one effect.

The power of algebraic effects lies in the fact that we can easily combine different effects, and that handlers can make use of the entire handled contexts. The latter can be illustrated with the following two examples:

```
effect Error =
  { error : type T. Unit => T }

let herr = handle
| error () => 0
| return x => x
end
```

```
effect NonDet =
  { flip : Unit => Bool
  ; fail : Unit => Unit }

let hnondet = handle
| return x => [x]
| fail ()  => []
| flip ()  => append (resume True)
                     (resume False)
end
```

The handler `herr` simply throws the entire context away, since it answers with `0`, but it does not resume. That is, the entire context is replaced with `0`. The handler `hnondet` is a handler for a nondeterministic computation that stores all available answers on a list. Handling the operation `flip` uses the context two times, one for each possible result of the nondeterministic choice. We can freely mix the two effects within one expression, and handle it by placing two handlers:

```
handle
  handle
    if flip () then 7 else (error () + 1)
  with herr
with hnondet
```

First, we evaluate `flip ()`, so the `hnondet` handler takes over, while `herr` is used only in the second `resume` of the operation `flip`. Thus, the overall result is the list `[7, 0]`. The effect associated with the inner expression is given by the row `[Error, Nondet]` (in this case, the order of the effects in the row does not matter).

Our calculus supports polymorphism over rows, which means that an expression can be given a row of effects that can be extended with additional effects depending on the context. A polymorphic row ends with a variable, and is denoted as, for example, `[Error, Nondet | r]`. The fact that a function performs effects is visible in its type, in which the arrow is decorated with the row of effects. For example, a polymorphic `iterate` function could be given the following type:

```
Int -> (a ->[|r] a) -> a ->[|r] a
```

Now, we give two motivating examples for the two features that provide abstraction mechanisms in programming with algebraic effects.

Existential Effects. Algebraic effects are appreciated for the separation of the interface of an effect and its semantics given by handlers. However, we do not always want the interface of an effect to be the interface provided by a module or a library, especially when we want to abstract away the information what effect is really in use, or we want to allow the client to use the effect only in some specific way. Leijen shows an example of this in [2018], where he marks a "filesystem" effect as abstract, in order to hide its component operations from the clients, and ensure they only use the built-in handler. Similarly, we can think of effects such as I/O, which are handled by the runtime system of a language, as abstract effects *without* a provided handler, which the client can only call, but never handle.

As a more complex illustration of the power of abstract effects, consider the following signature for a Union-Find data structure, inspired by SML's UREF signature:

```
type  Set : type -> type
effect UF : type -> effect

val new    : a ->[UF a] Set a
val find   : Set a ->[UF a] a
val union  : (a -> a ->[|r] a) -> Set a -> Set a ->[UF a | r] Unit
val withUF : (Unit ->[UF a | r] b) ->[|r] b
```

Here, we specify that in addition to the abstract type `Set` of disjoint sets, we also provide an abstract *effect* `UF`, and that the usual operations of `new`, `find` and `union` generate this effect. Note that since in our calculus an arrow without annotation signifies a *pure* function, `union` indeed must have some effect annotation, since its final result is always trivial. Note that this signature *abstracts* from the implementation details of `UF` – we have no information how many operations there are in the effect, and what are their types. We only know that the Union-Find functions introduce such an effect.

Since the client cannot write a handler for `UF`, not knowing its definition, the library also has to provide an *abstract* handler. This is the role of `withUF`, which takes a computation that performs the effect `UF`, and removes it by interpreting the underlying operations, which remain unexposed to the user. Note that the functions `new`, `find` and `union` are not simply operations of `UF` – they may be arbitrarily complex; indeed, with the given signature, `union` could not be expressed as an operation of an algebraic effect either in our calculus, or any other algebraic effect calculi that we are aware of.

The fact that `union` and `withUF` are polymorphic in the row of effects is important, since `union` takes as its first argument a function that is used to select a new representative of the two sets, and we want to be able to back this process with some additional effects. We elaborate on this example in Section 6, where we detail a concrete use-case.

Local Effects. The general practice of programming with effects is that we usually want to keep the effects local (except for some top-level effects handled by the runtime system, like I/O). This means that we use the effects in one part of the program for efficiency or to structure the code in a better way, but we do not want them to affect other parts of the program. Thus, we seek programming language constructs that ensure locality of an effect. Some guarantees are given by the type system alone, which keeps track of the effects used in an expression. However, this is not enough in the presence of effect polymorphism.

As an illustration, we revisit Example 2.4 in [Biernacki et al. 2018], and show an alternative solution that uses local effects. Assume that the context includes an effect `Tick` with one operation `tick : Unit => Unit`, and, for some types `T1` and `T2`, a function `val f : (T1 ->[|r] T2) ->[|r] Unit`, which is polymorphic in the row of effects `r`. Now, we try to define a new function, `cnt_f`, which counts the number of times `f` uses its argument. One approach would be as follows:

```
let cnt_f g =
  handle f (fn x => tick (); g x) with
  | tick ()  => fn n => resume () (n+1)
  | return _ => fn n => n
  end 0
```

Indeed, every time `f` calls `g x`, the `tick` operation is called first, which causes the counter in the handler to increment. This function works as expected, until we use it with a `g` that has the `Tick` effect in its row. In such a case, the `tick` operations in the definition of `g` are handled by the handler given in the body of `cnt_f`, which is obviously not what we intended. What we want is to treat the `tick` operation in the argument of `f` locally. In Helium, we can explicitly say it as follows:

```
let cnt_f g =
  effect Tick = { tick : Unit => Unit } in
  handle f (fn x => tick (); g x) with
  | tick ()  => fn n => resume () (n+1)
  | return _ => fn n => n
  end 0
```

This guarantees that whether there is a `Tick` effect declared in the global context or not, the `Tick` effect in the definition of `cnt_f` is local, hence it cannot occur in the row of `g`.

1.2 Implementing Abstract Effects

Now, we discuss the problem that might occur in a naive implementation of modules that enable abstract effects. Usually, in a language with a strong type system, one can erase all the type information, and still be sure that the program behaves well at runtime. For example, the type system guarantees that a constructor of an algebraic data type is always paired with a `match` for the same type, so one does not have to remember a constructor's type. In particular, it is enough to identify a constructor by its index within the algebraic data type, while `match` can be implemented as a single 'switch'. In such cases, existential types can be simply erased as well.

However, if the language provides algebraic effects, one cannot statically decide which handler will be needed for a given expression. Thus, one common way to implement algebraic effects is for an operation to be decorated with a piece of type information: a label that makes it possible to pair the operation with the right handler. The fact that not all types are erased makes implementation of existential types problematic. Consider the following example, assuming we have the `Reader` effect constructor in the context. First, we define a signature of a module `M`:

```
effect E
val my_ask    : Unit ->[E] Int
val my_handle : (Unit ->[E|r] a) ->[|r] a
```

We implement the module `M` as follows:

```
effect E := Reader Int
let my_ask      = ask
let my_handle t = handle t () with | ask ()   => resume 1
                                   | return x => x end
```

We use it in the following expression:

```
handle
  handle ask () + M.my_ask () with | ask ()   => resume 5
                                   | return x => x end
with M.my_handle
```

In the expression `ask() + M.my_ask ()`, there are two effects in play: `Reader Int` and `E`, which comes from the module `M`. Although both effects are in reality `Reader Int`, the latter is abstract, so, in order to enforce abstraction barriers, we treat them as two separate effects. Hence, the value of the call of `M.my_handle` in the last line is `6`. But if we simply erase the types (except for effect labels), there is no distinction between the top-level `ask` operation and the `ask` operation given by `M.my_ask`, because they are both associated with `Reader Int`. In such a case, the handler defined in the last line would take care of both operations, and the overall result would be `10`.

To solve this problem, we propose to use *effect coercions*. These constructs were introduced to the algebraic effect literature by Saleh et al. [2018] in order to make complex subtyping rules explicit and easier to track: we use a similar notion of a coercion, but in a slightly different way. In our example, the type system knows that `E` is abstract, hence should be treated as fresh with respect to all other effects, including `Reader`. But, since there is another effect in the row of the expression, we require an appropriate coercion to be placed in front of `M.my_ask`, which causes the evaluation to circumvent the inner handler. We discuss the coercions used in λ^{HEL} and how they affect such examples in Section 2.

1.3 Contributions

- We introduce the first type-and-effect system and operational semantics that accounts for *local definitions* of algebraic effects and *effect abstraction*.
- In order to ensure type-soundness of the calculus, we employ *effect coercions* in a novel way that extends sub-effecting to transitions with computational content.
- We construct an *abstract-machine implementation* that is correct with respect to the operational semantics.
- We provide a proof-of-concept programming language that allows the user to program with local and abstract effects in a familiar setting of an ML-like module system.

2 CORE CALCULUS

In this section, we introduce a core calculus of algebraic effects with abstract effects and row polymorphism, called λ^{HEL}. The calculus is based on the call-by-value λ-calculus, with a type system that allows for polymorphic and existential abstraction over types, type constructors, effects, and effect rows. It is an extension and generalization of the $\lambda^{\mathsf{H/L}}$-calculus, introduced in [Biernacki et al. 2018], where the authors consider a fragment of the type system addressed in the present work in which the only available means of type-level abstraction is row polymorphism [Hillerström and Lindley 2016; Leijen 2017].

2.1 Syntax

The syntax of λ^{HEL} is shown in Figure 1. We assume an infinite set Var of expression variables ranged over by $f, r, x, y, z, \ldots$ possibly with indices and primes. Similarly, we assume a set TVar of

$$
\begin{array}{rcll}
\mathrm{Var} \ni f, r, x, y, \ldots & & & \text{(variables)} \\
\mathrm{TVar} \ni \alpha, \beta, \ldots & & & \text{(effect and row variables)} \\
\mathrm{OpName} \ni o & & & \text{(operation names)} \\
\mathrm{Kind} \ni \kappa & ::= & \mathsf{T} \mid \mathsf{E} \mid \mathsf{R} \mid \kappa \to \kappa & \text{(kinds)} \\
\mathrm{Typelike} \ni \sigma, \tau, \varepsilon, \rho & ::= & \alpha \mid \tau\ \tau \mid \tau \to_{\rho} \tau \mid \forall\, \alpha :: \kappa.\ \tau \mid \exists\, \alpha :: \kappa.\ \tau \mid & \text{(types, etc.)} \\
& & \langle\rangle \mid \langle \varepsilon | \rho \rangle & \\
\theta & ::= & \overline{\alpha :: \kappa}.\ \left\{\overline{\delta}\right\} & \text{(effect declarations)} \\
\delta & ::= & o : \overline{\alpha :: \kappa}.\ \tau \Rightarrow \tau & \text{(operation declarations)} \\
\mathrm{TCont} \ni \Delta & ::= & \cdot \mid \Delta, \alpha :: \kappa \mid \Delta, \alpha = \theta & \text{(type contexts)} \\
\mathrm{Exp} \ni e & ::= & v \mid e\ e \mid e\ \tau \mid \langle c \rangle\ e \mid & \text{(expressions)} \\
& & \mathbf{pack}(\tau, e)\ \mathbf{as}\ \exists\, \alpha :: \kappa.\ \tau \mid & \\
& & \mathbf{unpack}\ e\ \mathbf{as}\ \alpha :: \kappa\mathbf{,}\ x : \tau\ \mathbf{in}\ e \mid & \\
& & \mathbf{effect}\ \alpha = \theta\ \mathbf{in}\ e \mid \mathbf{handle}_{\varepsilon}\ e\ \{\overline{h}; d\} & \\
\mathrm{Val} \ni u, v & ::= & x \mid \boldsymbol{\lambda}\, x : \tau\mathbf{.}\ e \mid \boldsymbol{\Lambda}\, \alpha :: \kappa\mathbf{.}\ e \mid & \text{(values)} \\
& & \mathbf{pack}(\varepsilon, v)\ \mathbf{as}\ \exists\, \alpha :: \kappa.\ \tau \mid o_{\varepsilon}\overline{\tau} & \\
c & ::= & c \cdot c \mid \varepsilon : c \mid \uparrow\!\varepsilon \mid \varepsilon \leftrightarrow \varepsilon & \text{(coercions)} \\
h & ::= & o\ \overline{\alpha :: \kappa}\ (x : \tau)/(r : \tau) \Rightarrow e & \text{(effect handlers)} \\
d & ::= & \mathbf{return}\ x : \tau \Rightarrow e & \text{(return clauses)} \\
\mathrm{ECont} \ni E & ::= & \square \mid E\ e \mid v\ E \mid E\ \tau \mid \langle c \rangle\ E \mid & \text{(evaluation contexts)} \\
& & \mathbf{pack}(\tau, E)\ \mathbf{as}\ \exists\, \alpha :: \kappa.\ \tau \mid & \\
& & \mathbf{unpack}\ E\ \mathbf{as}\ \alpha :: \kappa\mathbf{,}\ x : \tau\ \mathbf{in}\ e \mid & \\
& & \mathbf{handle}_{\varepsilon}\ E\ \{\overline{h}; d\} & \\
\mathrm{VCont} \ni \Gamma & ::= & \cdot \mid \Gamma, x : \tau & \text{(variable contexts)}
\end{array}
$$

Fig. 1. Syntax of the calculus λ^{HEL}

type variables, ranged over by $\alpha, \beta, \ldots$, that allow for polymorphic and existential abstraction over types, effects, rows of effects, etc. Operation names o are drawn similarly from the set OpName. All bound variables are type- or kind-annotated.

Conventions. We assume that all variables in type and term contexts are unique, implicitly alpha-renaming type- and term-level variables where necessary. Overlines denote (possibly empty) lists of objects, with the syntax like $\overline{\kappa} \to \kappa'$ denoting a right-associative chain of arrows, while $\sigma\ \overline{\tau}$ denotes a left-associative chain of (type) applications. We denote substitutions of values for term variables (in expressions, values, etc.) with $\{v\ /\ x\}$, and substitutions of types for type variables (in types, coercions, expressions, etc.) with $\{\tau\ /\ \alpha\}$. The substitutions are capture-avoiding and can be extended to entire lists of terms/variables, with the implicit understanding that the lists in question are required to be of equal lengths. Wherever the general types are assumed to be kinded, the ε metavariable ranges over effects (i.e., general types of kind E), and ρ – over rows (those of kind R). We use σ, τ to range over proper types (of kind T), as well as any general type where there is no kind distinction. Type constructors are usually variables, and at any rate their kind is then present in the rules or text.

Kinds, types, and well-formedness. The grammar of kinds includes types (T), effects (E), effect rows (R) as well as the arrow kind. The grammar of types allows us to construct annotated arrow types, universal and existential types with the bound variable ranging over types of arbitrary kind, type constructor application, and effect rows ($\langle\rangle$ and $\langle\varepsilon|\tau\rangle$).[1] The well-formedness of a type is considered in a type context Δ, which is a sequence of type variable declarations (possibly representing type or effect constructors) and of effect variable definitions. Well-formed types, effect definitions, as well as type (and variable) contexts are selected by the standard judgments $\Delta \vdash \tau :: \kappa$, $\Delta \vdash \theta$, and $\vdash \Delta$ ($\Delta \vdash \Gamma$), respectively, that we omit. We assume that the judgment $\vdash \Delta$ allows for recursive effect declarations.

Expressions, values and coercions. Expressions Exp and values Val include the call-by-value λ-calculus (variables are values), polymorphic abstraction and instantiation, standard operations for packing and unpacking values of an existential type or effect, and the effect-specific constructs. In particular, we allow for local effect definitions of the form **effect** $\alpha = \theta$ **in** e, which binds α to the effect declaration θ in e. Furthermore, the grammar of values includes type-instantiated operation names $o_\varepsilon\overline{\tau}$, associated with the effect ε, whereas the grammar of expressions includes effect-handling expressions $\textbf{handle}_\varepsilon\ e\ \{\overline{h}; d\}$, where d is a return clause of the form **return** $x : \tau \Rightarrow e$, and $\overline{h}$ is an effect handler for ε, that is, a finite list of operation-handling expressions. The order in the list is irrelevant, but we assume that all operations associated with an effect are mentioned exactly once in a given handler. An operation handler $o\ \overline{\alpha :: \kappa}\ (x : \tau)/(r : \tau) \Rightarrow e$ binds the variables x (representing the single argument of the operation) and r (standing for *r*esume and representing the continuation of the operation), whereas a return clause **return** $x : \tau \Rightarrow e$ binds x.

The final components of λ^{HEL} are coerced expressions $\langle c\rangle\ e$, where c is a coercion used to rearrange effects in effect rows as dictated by the type system of Section 2.2, and to introduce the corresponding behavior in the operational semantics. In particular, the coercion $\uparrow\varepsilon$ corresponds to the operator *lift* introduced in [Biernacki et al. 2018] to make the row polymorphism behave well in the presence of duplicated effect labels in a row, whereas the coercion $\varepsilon_1 \leftrightarrow \varepsilon_2$ (*swap*) allows us to exchange effects ε_1 and ε_2 even if they may not be distinct. This behavior allows us to treat abstract effects, which may or may not be distinct from each other, in a sound manner; we provide more explanation and illustrative examples in the following sections. Finally, the coercion $\varepsilon : c$ (*cons*) allows us to coerce deeper within an effect row (for instance to swap the second and third effects, or to express a generalized lift that Biernacki et al. encode at a steep computational cost), and composition of coercions allows us to push multiple atomic coercions under a cons, thus simplifying the structure of more complex coercions.

2.2 Type-and-Effect System

Before we explore the typing rules of the system, we introduce the notions of type equivalence, modulo which the remainder of the type system works, the notion of subtyping that we use, and the typing rules for coercions.

Type equivalence. It is standard in row-typed systems to consider a notion of row equivalence and work modulo that notion. In the case of algebraic effects, this usually amounts to allowing free exchange of any distinct effects (or, more precisely, distinct effect *constructors*) [Hillerström and Lindley 2016; Leijen 2017]. This is justified in the operational semantics by the fact that an operation must match the handler for *its* effect – thus all handlers of *other* effects one might encounter on the way are inconsequential. Clearly, this cannot generally extend to effect-kinded type variables, as these could potentially denote an incompatible effect declaration (in other words, such exchange would be incompatible with substitution). We can, however, find a middle ground, expressed by the

[1]We use the following syntactic sugar: $\langle\varepsilon_1, \varepsilon_2|\rho\rangle$ stands for $\langle\varepsilon_1|\langle\varepsilon_2|\rho\rangle\rangle$, whereas $\langle\varepsilon_1, \varepsilon_2\rangle$ stands for $\langle\varepsilon_1, \varepsilon_2|\langle\rangle\rangle$.

$$\frac{\Delta \vdash \sigma_2 <: \sigma_1 :: \mathsf{T} \qquad \Delta \vdash \rho_1 <: \rho_2 :: \mathsf{R} \qquad \Delta \vdash \tau_1 <: \tau_2 :: \mathsf{T}}{\Delta \vdash \sigma_1 \rightarrow_{\rho_1} \tau_1 <: \sigma_2 \rightarrow_{\rho_2} \tau_2 :: \mathsf{T}} \qquad \frac{\Delta, \alpha :: \kappa \vdash \tau_1 <: \tau_2 :: \mathsf{T}}{\Delta \vdash \forall\, \alpha :: \kappa.\ \tau_1 <: \forall\, \alpha :: \kappa.\ \tau_2 :: \mathsf{T}}$$

$$\frac{\Delta, \alpha :: \kappa \vdash \tau_1 <: \tau_2 :: \mathsf{T}}{\Delta \vdash \exists\, \alpha :: \kappa.\ \tau_1 <: \exists\, \alpha :: \kappa.\ \tau_2 :: \mathsf{T}} \qquad \frac{\Delta \vdash \rho :: \mathsf{R}}{\Delta \vdash \langle\rangle <: \rho :: \mathsf{R}} \qquad \frac{\Delta \vdash \rho_1 <: \rho_2 :: \mathsf{R}}{\Delta \vdash \langle \varepsilon | \rho_1 \rangle <: \langle \varepsilon | \rho_2 \rangle :: \mathsf{R}}$$

$$\frac{\Delta \vdash \sigma \simeq \tau :: \kappa}{\Delta \vdash \sigma <: \tau :: \kappa} \qquad \frac{\Delta \vdash \tau_1 <: \tau_2 :: \kappa \qquad \Delta \vdash \tau_2 <: \tau_3 :: \kappa}{\Delta \vdash \tau_1 <: \tau_3 :: \kappa}$$

Fig. 2. Subtyping. In $\Delta \vdash \sigma <: \tau :: \kappa$ we assume that $\Delta \vdash \sigma :: \kappa$ and ensure that $\Delta \vdash \tau :: \kappa$.

following rule:

$$\frac{\Delta_1 \vdash \beta :: \overline{\kappa} \rightarrow \mathsf{E}}{\Delta_1, \alpha = \theta, \Delta_2 \vdash \beta\ \overline{\sigma} \# \alpha\ \overline{\tau}}$$

This rule states that the effect declaration α, applied to appropriate types, is compatible with any effect constructor β – be it a definition or a type variable – as long as β is typable "before" α, i.e., in some prefix of the context that does not contain α. If β is also an effect declaration, this just makes for a slightly convoluted way to express the standard notion. However, when it is a type variable, this setup ensures it can never denote α at runtime (or, that the rule is indeed compatible with substitution).

This notion of effect compatibility leads straightforwardly to a notion of type equivalence, which we take as the smallest congruence that is compatible with the type well-formedness rules and includes swapping compatible effects in rows, as in the following rule:

$$\frac{\Delta \vdash \varepsilon_1 \# \varepsilon_2}{\Delta \vdash \langle \varepsilon_1, \varepsilon_2 | \rho \rangle \simeq \langle \varepsilon_2, \varepsilon_1 | \rho \rangle :: \mathsf{R}}$$

Note that this judgment is clearly decidable. Thus, in the interest of clarity, in the following we work modulo type equivalence, freely identifying τ and τ' rather than writing $\Delta \vdash \tau \simeq \tau' :: \kappa$, and using a negated form of the judgment in the premises of reasoning rules.

As an example, consider an effect constructor Reader $= \alpha :: \mathsf{T}.\ \{\mathsf{ask} : .\ \mathsf{unit} \Rightarrow \alpha\}$. Two instances of this effect, say, Reader Bool and Reader Int are incompatible, and so cannot be freely exchanged in a row. This corresponds to the intuition that the row encodes the order in which the effects will be handled: clearly, if the handler that supplies integers were to handle an operation that expects a boolean as a result, our calculus would not be sound! This reasoning extends to a row $\langle$Reader Bool, $\alpha\rangle$, where α is an effect-kinded type variable formed in the context that includes the declaration of Reader, as substitution could reduce this case to the previous one. However, if Reader were introduced in a context where α is already present (as a locally declared effect), scoping rules would preclude instantiation of α with Reader – and so we declare these effects compatible and can freely exchange them in a row.

Subtyping. Subtyping, presented in Figure 2, is defined as a reflexive and transitive relation that is compatible with the type formation rules, with the appropriate variance: quantifiers and row constructors are covariant and the arrow type is contravariant in its first argument (and covariant in the others) while effects are always invariant. Thus, except for the rule that allows us to "open" an empty effect row with any well-formed row ρ the subtyping rules are fairly standard.

$$\frac{\Delta \vdash \varepsilon :: \mathsf{E}}{\Delta \vdash \uparrow\!\varepsilon : \rho \triangleright \langle \varepsilon | \rho \rangle} \qquad \frac{}{\Delta \vdash \varepsilon_1 \leftrightarrow \varepsilon_2 : \langle \varepsilon_1, \varepsilon_2 | \rho \rangle \triangleright \langle \varepsilon_2, \varepsilon_1 | \rho \rangle}$$

$$\frac{\Delta \vdash c : \rho \triangleright \rho'}{\Delta \vdash \varepsilon : c : \langle \varepsilon | \rho \rangle \triangleright \langle \varepsilon | \rho' \rangle} \qquad \frac{\Delta \vdash c_1 : \rho_1 \triangleright \rho_2 \qquad \Delta \vdash \rho_1 \simeq \rho_2 :: \mathsf{R} \qquad \Delta \vdash c_2 : \rho_2 \triangleright \rho_3}{\Delta \vdash c_1 \cdot c_2 : \rho_1 \triangleright \rho_3}$$

Fig. 3. Coercion typing. In $\Delta \vdash c : \rho_1 \triangleright \rho_2$ we assume $\Delta \vdash \rho_1 :: \mathsf{R}$ and ensure that $\Delta \vdash \rho_2 :: \mathsf{R}$.

Coercion typing. We have noted before that coercions are intended to change the effect rows in a way that affects the operational semantics – and so beyond what we choose to express with subtyping. Thus, the judgment of a form $\Delta \vdash c : \rho_1 \triangleright \rho_2$ expresses that a coercion c takes the row ρ_1 to the row ρ_2; the rules may be found in Figure 3. As expected, the rule for the lift coercion matches the lift operation in Biernacki et al., and the cons and composition coercions behave in the obvious way. The interesting rule is the swap coercion, which exchanges the effects ε_1 and ε_2 at the beginning of the row. Note the similarity to the rule for row equivalence presented above: the only difference is in the lack of compatibility requirement and in the directedness of the rule (which is arbitrary). Note that this rule *can* be used to exchange compatible effects, even though the rows would then be equivalent: this is crucial to ensure compatibility under substitution.

Consider again the Reader effect and some effect-kinded type variable $\alpha :: \mathsf{E}$ that is incompatible with it. In order to coerce a row $\langle \mathsf{Reader\ Bool}, \alpha \rangle$ to a row $\langle \alpha, \mathsf{Reader\ Bool} \rangle$ we need a coercion $\mathsf{Reader\ Bool} \leftrightarrow \alpha$. Similarly, if we take an *open* row $\langle \mathsf{Reader\ Bool} | \beta \rangle$ for some $\beta :: \mathsf{R}$, and want to coerce it to $\langle \mathsf{Reader\ Bool}, \alpha | \beta \rangle$, we cannot simply use subtyping (as we cannot freely extend open rows). Instead, we need to apply a coercion $\mathsf{Reader\ Bool} : \uparrow\!\alpha$, which adds α to the row *under* the occurrence of the Reader effect. Another possibility is to use coercion composition, add α at the front of the row, and commute it with the Reader, as follows: $\uparrow\!\alpha \cdot \alpha \leftrightarrow \mathsf{Reader\ Bool}$. We revisit this example after considering the semantic content of coercions, to explain how the two correspond.

Expression typing. Finally, we come to the typing rules for expressions and handlers, which are presented in Figure 4. Most of the rules are standard, the interesting ones have to do with algebraic effects. Firstly, note that the rule for local effects adds the effect declaration to Δ, but ensures that the return type and effect are free of the local definition, much like the rules for local memory regions in type-and-effect systems for memory management [Tofte and Talpin 1997]. In effect this ensures that all the occurrences of the operations of the local effect are handled, including those in suspended computations. Secondly, the effect annotation at the operations and effect handlers has to start with a *definition* (α) applied to appropriate type arguments. This means that effect-kinded type variables, introduced for instance by unpacking an existential effect, cannot appear in this position – which ensures their abstract treatment. Finally, in all these rules, as in the rule for typing an effect handler, we somewhat abuse the overline notation to ensure that the numbers of arguments in various lists match, and that for each appropriate pair a given judgment holds.

2.3 Operational Semantics

We define the operational semantics of our calculus as a reduction semantics. The rules are presented in Figure 6, where we first give the notion of reduction (contraction) and then define how complete programs are evaluated (reduction relation). The first interesting thing to note is the shape of the judgment: $\Delta; e \longrightarrow \Delta'; e'$. Similarly to the typing rules, Δ stores the declared effects; the interesting part, however, is its global evolution. Note the reduction rule for the local effect, which allocates α

$$\frac{x : \tau \in \Gamma}{\Delta;\Gamma \vdash x : \tau \ /\ \langle\rangle} \qquad \frac{\Delta \vdash \sigma :: \mathsf{T} \qquad \Delta;\Gamma, x : \sigma \vdash e : \tau \ /\ \rho}{\Delta;\Gamma \vdash \boldsymbol{\lambda} x : \sigma\mathbf{.}\ e : \sigma \to_{\rho} \tau \ /\ \langle\rangle} \qquad \frac{\Delta, \alpha :: \kappa;\Gamma \vdash e : \tau \ /\ \langle\rangle}{\Delta;\Gamma \vdash \boldsymbol{\Lambda} \alpha :: \kappa\mathbf{.}\ e : \forall \alpha :: \kappa.\ \tau \ /\ \langle\rangle}$$

$$\frac{\Delta;\Gamma \vdash e_1 : \sigma \to_{\rho} \tau \ /\ \rho \qquad \Delta;\Gamma \vdash e_2 : \sigma \ /\ \rho}{\Delta;\Gamma \vdash e_1\ e_2 : \tau \ /\ \rho} \qquad \frac{\Delta;\Gamma \vdash e : \forall \alpha :: \kappa.\ \tau \ /\ \rho \qquad \Delta \vdash \sigma :: \kappa}{\Delta;\Gamma \vdash e\ \sigma : \tau\{\sigma / \alpha\} \ /\ \rho}$$

$$\frac{\Delta \vdash \sigma :: \kappa \qquad \Delta;\Gamma \vdash e : \tau\{\sigma / \alpha\} \ /\ \rho}{\Delta;\Gamma \vdash \mathbf{pack}(\sigma, e)\ \mathbf{as}\ \exists \alpha :: \kappa.\ \tau : \exists \alpha :: \kappa.\ \tau \ /\ \rho}$$

$$\frac{\Delta;\Gamma \vdash e_1 : \exists \alpha :: \kappa.\ \sigma \ /\ \rho \qquad \Delta, \alpha :: \kappa;\Gamma, x : \sigma \vdash e_2 : \tau \ /\ \rho \qquad \Delta \vdash \tau :: \mathsf{T}}{\Delta;\Gamma \vdash \mathbf{unpack}\ e_1\ \mathbf{as}\ \alpha :: \kappa\mathbf{,}\ x : \sigma\ \mathbf{in}\ e_2 : \tau \ /\ \rho}$$

$$\frac{\Delta, \alpha = \theta \vdash \theta \qquad \Delta, \alpha = \theta;\Gamma \vdash e : \tau \ /\ \rho \qquad \Delta \vdash \tau :: \mathsf{T} \qquad \Delta \vdash \rho :: \mathsf{R}}{\Delta;\Gamma \vdash \mathbf{effect}\ \alpha\ \mathbf{=}\ \theta\ \mathbf{in}\ e : \tau \ /\ \rho}$$

$$\frac{\alpha = \overline{\beta :: \kappa}.\ \left\{\overline{\delta}\right\} \in \Delta \qquad o : \overline{\gamma :: \kappa'}.\ \tau_1 \Rightarrow \tau_2 \in \overline{\delta} \qquad \overline{\Delta \vdash \sigma :: \kappa} \qquad \overline{\Delta \vdash \sigma' :: \kappa'}}{\Delta;\Gamma \vdash o_{\alpha\ \overline{\sigma}}\overline{\sigma'} : \tau_1\{\overline{\sigma} / \overline{\beta}\}\{\overline{\sigma'} / \overline{\gamma}\} \to_{\langle \alpha\ \overline{\sigma} \rangle} \tau_2\{\overline{\sigma} / \overline{\beta}\}\{\overline{\sigma'} / \overline{\gamma}\} \ /\ \langle\rangle}$$

$$\frac{\Delta;\Gamma \vdash e : \tau_a \ /\ \langle \alpha\ \overline{\sigma} | \rho \rangle \qquad \dfrac{\alpha = \overline{\beta :: \kappa}.\ \left\{\overline{\delta}\right\} \in \Delta \qquad \overline{\Delta \vdash \sigma :: \kappa}}{\Delta;\Gamma;\delta\{\overline{\sigma} / \overline{\beta}\} \vdash h : \tau_r \ /\ \rho} \qquad \Delta;\Gamma, x : \tau_a \vdash e_r : \tau_r \ /\ \rho}{\Delta;\Gamma \vdash \mathbf{handle}_{\alpha\ \overline{\sigma}}\ e\ \mathbf{\{}\overline{h};\ \mathbf{return}\ x : \tau_a\ \mathbf{\Rightarrow}\ e_r\mathbf{\}} : \tau_r \ /\ \rho}$$

$$\frac{\Delta, \overline{\alpha :: \kappa};\Gamma, x : \tau_1, r : \tau_2 \to_{\rho} \sigma \vdash e : \sigma \ /\ \rho}{\Delta;\Gamma; o : \overline{\alpha :: \kappa}.\ \tau_1 \Rightarrow \tau_2 \vdash o\ \overline{\alpha :: \kappa}\ (x : \tau_1)/(r : \tau_2 \to_{\rho} \sigma)\ \mathbf{\Rightarrow}\ e : \sigma \ /\ \rho}$$

$$\frac{\Delta;\Gamma \vdash e : \tau \ /\ \rho \qquad \Delta \vdash c : \rho \triangleright \rho'}{\Delta;\Gamma \vdash \langle c \rangle\ e : \tau \ /\ \rho'} \qquad \frac{\Delta;\Gamma \vdash e : \tau \ /\ \rho \qquad \Delta \vdash \tau <: \tau' :: \mathsf{T} \qquad \Delta \vdash \rho <: \rho' :: \mathsf{R}}{\Delta;\Gamma \vdash e : \tau' \ /\ \rho'}$$

Fig. 4. Expression and handler typing. We assume $\vdash \Delta$ and $\Delta \vdash \Gamma$, and ensure that $\Delta \vdash \tau :: \mathsf{T}$ and $\Delta \vdash \rho :: \mathsf{R}$.

globally in its contraction. This is required, since computations that refer to the local effect may get suspended, so the effect declaration itself has to be present in the "future world" where the suspended computation is called. At the same time, the typing discipline ensures that the effect is actually used only within its scope. This behavior is somewhat similar to the reference allocation in ML [Pierce 2002, Chapter 13] – although of course the effect declaration is immutable, so its behavior should be significantly easier to model.

The other important contraction rule is handling of an operation. Following [Biernacki et al. 2018] we express the fact that the operation is handled by its matching handler via a freeness judgment, presented in Figure 5. When the appropriate judgment is located, the operation is found within the handler, and the continuation gets captured and passed to the handler code as a resumption r. As the other rules are standard or trivial, we now explore freeness in more detail.

Freeness of effects in an evaluation context $\boxed{n\text{–free}(\varepsilon, E)}$

$$\frac{}{0\text{–free}(\alpha, \square)} \qquad \frac{n\text{–free}(\alpha, E)}{n\text{–free}(\alpha, E\ e)} \qquad \frac{n\text{–free}(\alpha, E)}{n\text{–free}(\alpha, v\ E)} \qquad \frac{n\text{–free}(\alpha, E)}{n\text{–free}(\alpha, E\ \tau)}$$

$$\frac{n\text{–free}(\alpha, E)}{n\text{–free}(\alpha, \mathbf{pack}(\tau, E)\ \mathbf{as}\ \exists\, \alpha :: \kappa.\ \tau)} \qquad \frac{n\text{–free}(\alpha, E)}{n\text{–free}(\alpha, \mathbf{unpack}\ E\ \mathbf{as}\ \beta :: \kappa, x : \tau\ \mathbf{in}\ e)}$$

$$\frac{n+1\text{–free}(\alpha, E)}{n\text{–free}(\alpha, \mathbf{handle}_{\alpha\ \overline{\sigma}}\ E\ \{\overline{h}; d\})} \qquad \frac{n\text{–free}(\alpha, E) \qquad \alpha \neq \alpha'}{n\text{–free}(\alpha, \mathbf{handle}_{\alpha'\ \overline{\sigma}}\ E\ \{\overline{h}; d\})}$$

$$\frac{n\text{–free}(\alpha, E) \qquad \alpha : n \overset{c}{\rightsquigarrow} m}{m\text{–free}(\alpha, \langle c \rangle\ E)}$$

Transformation of n-freeness through coercions $\boxed{\varepsilon : n \overset{c}{\rightsquigarrow} m}$

$$\frac{\alpha : n \overset{c_1}{\rightsquigarrow} m \qquad \alpha : m \overset{c_2}{\rightsquigarrow} k}{\alpha : n \overset{c_1 \cdot c_2}{\rightsquigarrow} k} \qquad \frac{\alpha : n \overset{c}{\rightsquigarrow} m}{\alpha : n+1 \overset{\alpha\ \overline{\sigma} : c}{\rightsquigarrow} m+1} \qquad \frac{}{\alpha : 0 \overset{\alpha\ \overline{\sigma} : c}{\rightsquigarrow} 0} \qquad \frac{\alpha : n \overset{c}{\rightsquigarrow} m \qquad \alpha \neq \alpha'}{\alpha : n \overset{\alpha'\ \overline{\sigma} : c}{\rightsquigarrow} m}$$

$$\frac{}{\alpha : n \overset{\uparrow \alpha\ \overline{\sigma}}{\rightsquigarrow} n+1} \qquad \frac{\alpha \neq \alpha'}{\alpha : n \overset{\uparrow \alpha'\ \overline{\sigma}}{\rightsquigarrow} n} \qquad \frac{}{\alpha : 0 \overset{\alpha\ \overline{\sigma} \leftrightarrow \alpha\ \overline{\sigma'}}{\rightsquigarrow} 1} \qquad \frac{}{\alpha : 1 \overset{\alpha\ \overline{\sigma} \leftrightarrow \alpha\ \overline{\sigma'}}{\rightsquigarrow} 0}$$

$$\frac{}{\alpha : n+2 \overset{\alpha\ \overline{\sigma} \leftrightarrow \alpha\ \overline{\sigma'}}{\rightsquigarrow} n+2} \qquad \frac{\alpha \neq \beta}{\alpha : n \overset{\beta\ \overline{\sigma} \leftrightarrow \beta\ \overline{\sigma'}}{\rightsquigarrow} n} \qquad \frac{\beta \neq \beta'}{\alpha : n \overset{\beta\ \overline{\sigma} \leftrightarrow \beta'\ \overline{\sigma'}}{\rightsquigarrow} n}$$

Fig. 5. Effect freeness

In the simplest case, it only checks that the effect is not handled earlier in the context via some other handler that would catch the same effect constructor. However, freeness interacts non-trivially with the coercions in the evaluation contexts, potentially causing some of the handlers in the contexts to become "inert." These are engineered to match the appropriate rules of the type system in a way that we discuss in the following. Intuitively, the lift coercion on an effect $\alpha\ \overline{\sigma}$ ensures that the nearest enclosing handler for α will *not* handle any operation under that coercion, while the swap coercion "exchanges" the matching handlers for the two first occurrences of the matching effect. Like in the typing judgment, the cons coercion simply shifts these coercions to handlers further outside the nearest enclosing ones.

In order to see the semantics in action, consider the examples from the previous section. First, consider two operations, $\mathsf{ask}_{\mathsf{Reader\ Bool}}\ ()$ and $\mathsf{ask}_{\mathsf{Reader\ Int}}\ ()$. Clearly, these should not be handled by the same handler. However, if we wrote $\mathsf{f}\ (\mathsf{ask}_{\mathsf{Reader\ Bool}}\ ())\ (\mathsf{ask}_{\mathsf{Reader\ Int}}\ ())$ (for some binary function f), this is what would happen, as both these operations would match the same enclosing handler! If we use a lift coercion on the second of these, we get $\mathsf{f}\ (\mathsf{ask}_{\mathsf{Reader\ Bool}}\ ())\ (\langle \uparrow \mathsf{Reader\ Bool} \rangle\ \mathsf{ask}_{\mathsf{Reader\ Int}}\ ())$, which, if we look carefully at the definition of freeness, ensures that the context for the second operation will always be more free than the one for the second. In the end, this means that (barring additional coercions) the second operation will skip past the handler that handles the first operation, *Contraction* $\boxed{\Delta; e \mapsto \Delta; e}$

$$\frac{}{\Delta; (\boldsymbol{\lambda}\, x : \tau.\, e)\; v \mapsto \Delta; e\{v \,/\, x\}} \qquad \frac{}{\Delta; (\boldsymbol{\Lambda}\, \alpha :: \kappa.\, e)\; \tau \mapsto \Delta; e\{\tau \,/\, \alpha\}}$$

$$\frac{}{\Delta; \textbf{unpack pack}(\sigma, v)\ \textbf{as}\ \exists\, \alpha :: \kappa.\ \tau\ \textbf{as}\ \alpha :: \kappa, x : \sigma\ \textbf{in}\ e \mapsto \Delta; e\{\sigma \,/\, \alpha\}\{v \,/\, x\}}$$

$$\frac{\begin{array}{cc} \alpha = \theta \in \Delta & 0\text{–free}(\alpha, E) \\ o\ \overline{\beta :: \kappa}\ (x : \tau_x)/(r : \tau_r) \Rightarrow e \in \overline{h} & v_c = \boldsymbol{\lambda}\, z : \tau_r.\ \textbf{handle}_{\alpha\ \overline{\sigma}}\ E[z]\ \{\overline{h}; d\} \end{array}}{\Delta; \textbf{handle}_{\alpha\ \overline{\sigma}}\ E[o_{\alpha\ \overline{\sigma'}} \overline{\tau}\ v]\ \{\overline{h}; d\} \mapsto \Delta; e\{\overline{\tau} \,/\, \overline{\beta}\}\{v \,/\, x\}\{v_c \,/\, r\}}$$

$$\frac{}{\Delta; \textbf{handle}_{\varepsilon}\ v\ \{\overline{h}; \textbf{return}\ x : \sigma \Rightarrow e\} \mapsto \Delta; e\{v \,/\, x\}}$$

$$\frac{}{\Delta; \langle c \rangle\ v \mapsto \Delta; v} \qquad \frac{}{\Delta; \textbf{effect}\ \alpha = \theta\ \textbf{in}\ e \mapsto \Delta, \alpha = \theta; e}$$

Reduction relation $\boxed{\Delta; e \longrightarrow \Delta; e}$

$$\frac{\Delta; e \mapsto \Delta'; e'}{\Delta; E[e] \longrightarrow \Delta'; E[e']}$$

Fig. 6. Operational semantics

and be handled by the one further outside in the context. Moreover, note that this is precisely the coercion that is required for such a composition to be well-typed, and that it is a legitimate concern also in the case when both operations are annotated with the *same* effect (say, Reader Bool), but that ought to be handled by different handlers. As pointed out in the introduction, this is a common occurrence in the presence of existential types, and as argued in [Biernacki et al. 2018], it occurs already in the presence of row polymorphism.

What if, for some reason, we need a set order of handling, like the one obtained above, reversed? Biernacki et al. argue that this can be encoded in their system. However, their construction is quite involved – and what's worse, it does not scale to a setting with effect-kinded type variables, where we might want to swap a concrete effect with an abstract effect. Thus, we include the *swap* coercion directly in the semantics. Consider the example from the previous paragraph, but with Reader Int handled first. Without any additional coercions, such a handler would give the interpretation to the operation associated with the boolean type, potentially leading to an error. To avoid this, we can place a swap coercion between the expression and the handler, as in the following:

$$\langle \mathsf{Reader\ Bool} \leftrightarrow \mathsf{Reader\ Int} \rangle\ \mathsf{f}\ (\mathsf{ask}_{\mathsf{Reader\ Bool}}\ \mathbf{()})\ (\langle \uparrow \mathsf{Reader\ Bool} \rangle\ \mathsf{ask}_{\mathsf{Reader\ Int}}\ \mathbf{()}).$$

As this outer coercion changes 0-free Reader effects into 1-free, and vice versa, in this case it's the context of the second operation (associated with Reader Int) that is 0-free – and thus would be interpreted by the first enclosing handler. The context of the first operation, on the other hand, would be 1-free, so the first interpretation would be skipped, bringing freeness back to 0.

In conjunction, the coercion rules provide us with a robust if somewhat complex system that by design behaves well with substitutions – an essential characteristic for a calculus with effect abstraction. We touch upon this in the following section, when we state the appropriate substitution lemma.

2.4 Type Soundness

We prove the soundness of the type system presented above via the standard combination of progress and preservation lemmas [Harper 2016; Wright and Felleisen 1994]. We begin with the progress property. In order to state the lemma, we first need an additional predicate that can be used to connect the notions of freeness and the row of effects in the typing judgment. This will allow us to express the appropriate property for expressions that are stuck due to an operation not having a matching handler in the context – which we have to take into account, since we can reduce under handlers. The relation is defined by the following rules:

$$\frac{}{\alpha^0 \subseteq \rho} \qquad \frac{\alpha^n \subseteq \rho}{\alpha^{n+1} \subseteq \langle \alpha\ \overline{\sigma} | \rho \rangle} \qquad \frac{\varepsilon \neq \alpha\ \overline{\sigma} \qquad \alpha^n \subseteq \rho}{\alpha^n \subseteq \langle \varepsilon | \rho \rangle}$$

We can now state the lemma that uses this notion to connect the typing of coercions to their semantic effect. The proof follows by simple induction on the structure of coercion typing.

Lemma 2.1. *If* $\Delta \vdash c : \rho \triangleright \rho'$ *and* $\alpha^{n+1} \subseteq \rho$, *then there exists* m *such that* $\alpha : n \overset{c}{\rightsquigarrow} m$ *and* $\alpha^{m+1} \subseteq \rho'$.

With this lemma, we can state and prove the progress property. Note that the unusual third case is never encountered for closed programs, which have empty effect rows. However, this case is crucial when reducing under an effect handler, since it is what enables the operation-handler reduction (when $n = 0$).

Lemma 2.2 (Progress). *If* $\Delta; \cdot \vdash e : \tau \ /\ \rho$, *then one of the following holds:*

- e *is a value, i.e., there exists* $v \in \mathrm{Val}$ *such that* $e = v$;
- e *reduces in* Δ, *i.e., there exist* Δ' *and* e' *such that* $\Delta; e \longrightarrow \Delta'; e'$;
- e *is* control-stuck, *i.e., there exist* E, o, α, $\overline{\sigma}$, $\overline{\tau}$, v *and* n *such that* $e = E[o_{\alpha\ \overline{\sigma}} \overline{\tau}\ v]$, $n\text{–free}(\alpha, E)$ *and* $\alpha^{n+1} \subseteq \rho$ *all hold.*

We now turn to the preservation property. We first define the typing of evaluation contexts $\boxed{\Delta; \Gamma \vdash E : \tau \ /\ \rho \rightsquigarrow \tau \ /\ \rho}$ in terms of "future-world-closed" typing of expressions:

$$\Delta; \Gamma \vdash E : \tau_1 \ /\ \rho_1 \rightsquigarrow \tau_2 \ /\ \rho_2 \triangleq \forall \Delta', \Gamma', e.\ \Delta, \Delta'; \Gamma, \Gamma' \vdash e : \tau_1 \ /\ \rho_1 \Rightarrow \Delta, \Delta'; \Gamma, \Gamma' \vdash E[e] : \tau_2 \ /\ \rho_2.$$

We can use this definition to prove the following decomposition lemma, which follows by induction on typing derivations, with appropriate application of standard weakening lemmas.

Lemma 2.3. *If* $\Delta; \cdot \vdash E[e] : \tau \ /\ \rho$, *then there exist* τ' *and* ρ' *such that both* $\Delta; \cdot \vdash e : \tau' \ /\ \rho'$ *and* $\Delta; \cdot \vdash E : \tau' \ /\ \rho' \rightsquigarrow \tau \ /\ \rho$ *hold.*

Like the definition of our operational semantics, we split the proof of the type preservation property into two steps. First, we show that contractions preserve typing, which is mostly standard for a calculus with a subtyping relation and requires standard lemmas about substitution of types (in terms and in expressions) and values (in expressions only). We show the most complex and crucial of these, the preservation of typing judgment under substitution of types. While this lemma is mostly standard (the only surprising part is substitution in Δ', which is due to the fact that the contexts also contain effect declarations, in which α may appear), its importance is crucial, given that types are involved in the reduction, through handlers and coercions.

Lemma 2.4 (Substitution/type/expression). *If* $\Delta, \overline{\alpha :: \kappa}, \Delta'; \Gamma \vdash e : \tau \ /\ \rho$ *and* $\overline{\Delta \vdash \sigma :: \kappa}$, *then*

$$\Delta, \Delta'; \Gamma\{\overline{\sigma} \,/\, \overline{\alpha}\} \vdash e\{\overline{\sigma} \,/\, \overline{\alpha}\} : \tau\{\overline{\sigma} \,/\, \overline{\alpha}\} \ /\ \rho\{\overline{\sigma} \,/\, \overline{\alpha}\}.$$

Lemma 2.5 (Preservation/contraction). *If* $\Delta; \cdot \vdash e : \tau \ /\ \rho$ *and* $\Delta; e \mapsto \Delta'; e'$ *then* $\Delta'; \cdot \vdash e' : \tau \ /\ \rho$.

Preservation under the more general reduction is then simply the case of using the decomposition lemma stated above.

Lemma 2.6 (Preservation). *If* $\Delta; \cdot \vdash e : \tau \ / \ \rho$ *and* $\Delta; e \longrightarrow \Delta'; e'$ *then* $\Delta'; \cdot \vdash e' : \tau \ / \ \rho$.

Finally, we are in a position to prove the type soundness property of the calculus, stating that "well-typed programs don't go wrong." The proof is standard, save for the presence of the final clause of the statement of the progress lemma – which is impossible for closed programs. (We write $e \not\rightarrow$ when there is no e' such that $e \rightarrow e'$.)

Theorem 2.7 (Type Soundness). *If* $\Delta; \cdot \vdash e : \tau \ / \ \langle\rangle$ *and* $\Delta; e \longrightarrow^{*} \Delta'; e' \not\rightarrow$*, then there exists* v *such that* $e' = v$.

3 UNTYPED CALCULUS AND TYPE ERASURE

In this section, we show an intermediate step towards the execution model via an abstract machine: the untyped calculus. While λ^{HEL} is heavily decorated with types, most of the annotations are not necessary at runtime, so they can be simply erased. The vital type information is the effect constructor, which makes it possible to pair an operation with a handler.

We show the calculus and the type-erasure procedure. Type erasure preserves semantics of well-typed programs, and then the abstract machine, defined in Section 4, works on terms of the untyped calculus, realizing the semantics given in this section.

Syntax and Semantics. The syntax and semantics of the untyped calculus is given in Figure 7. It is an untyped λ-calculus with the explicit unit value, operations and handlers (decorated with values rather than types), the pair constructor and **letp** (counterparts of the **pack**–**unpack** duo), and **new** (counterpart of **effect**). Note that the pair constructor has a value instead of a type in the first component, while **new** binds a single variable instead of providing a whole effect definition. Another new element is the set l of *effect labels*. The unit value together with effect labels form a new syntactic category, *simple values*. The fact that in some places the syntax is restricted to values or simple values might seem arbitrary at first, but it serves a very practical purpose: we want the abstract machine to exactly match the reduction semantics given in Figure 7, and the less restricted syntax would require the machine to include additional transitions, which are unnecessary for programs coming from well-typed λ^{HEL} expressions.

The reduction semantics of the untyped calculus is very similar to the semantics of λ^{HEL}. The reduction relation is accompanied by a context Σ, which lists allocated effect labels, and whose sole purpose is to ensure that the label l in the contraction rule for **new** is fresh (when writing Σ, l we assume that l does not occur in Σ, i.e., l is fresh with respect to Σ). Effect freeness is defined similarly to the effect freeness for λ^{HEL}. That is, the n-freeness for the untyped calculus is preserved by all evaluation contexts except for handlers and coercions. For handlers and coercions, the difference is that we compare effect labels instead of effect constructors. Thus, we do not spell out the full definition, and Figure 7 includes only a few selected rules.

Type Erasure. The type-erasing translation on types ($\lfloor \tau \rfloor_\eta$), coercions ($\lfloor c \rfloor_\eta$), and expressions ($\lfloor e \rfloor_\eta$) is presented in Figure 8. It is parameterized by η, which is a map from type variables of λ^{HEL} to the values of the untyped calculus. Intuitively, η reveals if a given variable represents an effect constructor (in which case its value is a variable that will be instantiated with an effect label) or some other type (in which case the value of η is the unit value).

The procedure for types erases (that is, maps to the unit value) everything except for effect constructors, which are given by some type variables (intuitively, those that refer to effect definitions θ allocated on Δ). Thus, the value for a variable α is provided by the environment η, while a type

Syntax

$$
\begin{array}{rcll}
s & ::= & l \mid () & \text{(simple values)}\\
v & ::= & x \mid s \mid \boldsymbol{\lambda} x.\, e \mid (l, v) \mid o_l[\overline{s}] & \text{(values)}\\
e & ::= & v \mid (v, e) \mid o_v[\overline{v}] \mid e\ e \mid \textbf{letp}\ (x, y) = e\ \textbf{in}\ e \mid \langle c \rangle e \mid & \text{(expressions)}\\
 & & \textbf{new}\ x\ \textbf{in}\ e \mid \textbf{handle}_v\ e\{\overline{h}; d\} & \\
h & ::= & o[\overline{x}]\ y\ /\ r \Rightarrow e & \text{(handlers)}\\
d & ::= & \textbf{return}\ x \Rightarrow e & \text{(return clauses)}\\
c & ::= & c \cdot c \mid v : c \mid \uparrow v \mid v \leftrightarrow v & \text{(coercions)}\\
E & ::= & \square \mid (l, E) \mid E\ e \mid v\ E \mid \textbf{letp}\ (x, y) = E\ \textbf{in}\ e \mid \langle c \rangle E \mid & \text{(evaluation contexts)}\\
 & & \textbf{handle}_l\ E\{\overline{h}; d\} & \\
\Sigma & ::= & \overline{l} & \text{(effect contexts)}
\end{array}
$$

Operational semantics $\boxed{\begin{array}{l}\Sigma; e \mapsto \Sigma; e\\ \Sigma; e \longrightarrow \Sigma; e\end{array}}$

$$
\frac{}{\Sigma; (\boldsymbol{\lambda} x.\, e)\ v \mapsto \Sigma; e\{v / x\}}
\qquad
\frac{}{\Sigma; \textbf{letp}\ (x, y) = (l, v)\ \textbf{in}\ e \mapsto \Sigma; e\{l / x\}\{v / y\}}
\qquad
\frac{}{\Sigma; \langle c \rangle v \mapsto \Sigma; v}
$$

$$
\Sigma; \textbf{new}\ x\ \textbf{in}\ e \mapsto \Sigma, l; e\{l / x\}
\qquad
\frac{}{\Sigma; \textbf{handle}_l\ v\{\overline{h}; \textbf{return}\ x \Rightarrow e\} \mapsto \Sigma; e\{v / x\}}
$$

$$
\frac{0\text{–free}(l, E) \qquad o[\overline{x}]\ y\ /\ r \Rightarrow e \in \overline{h} \qquad v_c = \boldsymbol{\lambda} z.\, \textbf{handle}_l\ E[z]\{\overline{h}; d\}}{\Sigma; \textbf{handle}_l\ E[o_l[\overline{s}]\ v]\{\overline{h}; d\} \mapsto \Sigma; e\{\overline{s} / \overline{x}\}\{v / y\}\{v_c / r\}}
$$

$$
\frac{\Sigma; e \mapsto \Sigma'; e'}{\Sigma; E[e] \longrightarrow \Sigma'; E[e']}
$$

Freeness of effects and transformation through coercions (selected rules) $\boxed{\begin{array}{l}n\text{–free}(l, E)\\ l : n \overset{c}{\rightsquigarrow} m\end{array}}$

$$
\frac{n + 1\text{–free}(l, E)}{n\text{–free}(l, \textbf{handle}_l\ E\{\overline{h}; d\})}
\qquad
\frac{n\text{–free}(l, E) \qquad l \neq l'}{n\text{–free}(l, \textbf{handle}_{l'}\ E\{\overline{h}; d\})}
\qquad
\frac{n\text{–free}(l, E) \qquad l : n \overset{c}{\rightsquigarrow} m}{m\text{–free}(l, \langle c \rangle\ E)}
$$

$$
\frac{}{l : 0 \overset{l \leftrightarrow l}{\rightsquigarrow} 1}
\qquad
\frac{}{l : 1 \overset{l \leftrightarrow l}{\rightsquigarrow} 0}
\qquad
\frac{l \neq l'}{l : n \overset{l' \leftrightarrow l'}{\rightsquigarrow} n}
\qquad
\frac{l_0 \neq l_1}{l : n \overset{l_0 \leftrightarrow l_1}{\rightsquigarrow} n}
$$

Fig. 7. Syntax and semantics of the type-free calculus

application is first stripped of its argument, which is no longer needed. Erasure for coercions simply goes down the structure, and applies itself to the types.

To define the procedure for expressions, we first define an auxiliary function. Let $C(\kappa)$ denote the result kind of κ, with the definition given as follows:

$$
C(\kappa_1 \to \kappa_2) \ \triangleq\ C(\kappa_2) \qquad\qquad C(\kappa) \ \triangleq\ \kappa \quad \text{for } \kappa \in \mathsf{T}, \mathsf{R}, \mathsf{E}
$$

Then, type erasure is defined structurally on expressions, translating the related constructs of the two calculi. The environment η is extended for recursive calls in the constructions that bind new type variables. Note that the erasure procedure for $\boldsymbol{\Lambda}$'s, **pack**'s, and **unpack**'s depends on the kind κ of the introduced type variable α. In the case of $\boldsymbol{\Lambda}$, if α is an effect constructor (that

Erasure in coercions and types.

$$\begin{aligned} \lfloor c_1 \cdot c_2 \rfloor_\eta &= \lfloor c_1 \rfloor_\eta \cdot \lfloor c_2 \rfloor_\eta \\ \lfloor \varepsilon : c \rfloor_\eta &= \lfloor \varepsilon \rfloor_\eta : \lfloor c \rfloor_\eta \\ \lfloor \uparrow\varepsilon \rfloor_\eta &= \uparrow\lfloor \varepsilon \rfloor_\eta \\ \lfloor \varepsilon_1 \leftrightarrow \varepsilon_2 \rfloor_\eta &= \lfloor \varepsilon_1 \rfloor_\eta \leftrightarrow \lfloor \varepsilon_2 \rfloor_\eta \end{aligned} \qquad \begin{aligned} \lfloor \alpha \rfloor_\eta &= \eta(\alpha) \\ \lfloor \tau_1\ \tau_2 \rfloor_\eta &= \lfloor \tau_1 \rfloor_\eta \\ \lfloor \tau \rfloor_\eta &= () \qquad \text{in all other cases} \end{aligned}$$

Erasure in expressions.

$$\begin{aligned}
\lfloor x \rfloor_\eta &= x \\
\lfloor \boldsymbol{\lambda}\, x : \tau.\, e \rfloor_\eta &= \boldsymbol{\lambda}\, x.\, \lfloor e \rfloor_\eta \\
\lfloor \boldsymbol{\Lambda}\, \alpha :: \kappa.\, e \rfloor_\eta &= \begin{cases} \boldsymbol{\lambda}\, x.\, \lfloor e \rfloor_{\eta[\alpha \mapsto x]} & \text{when } C(\kappa) = \mathsf{E} \\ \boldsymbol{\lambda}\, x.\, \lfloor e \rfloor_{\eta[\alpha \mapsto ()]} & \text{otherwise} \end{cases} \\
\lfloor o_\varepsilon \overline{\tau} \rfloor_\eta &= o_{\lfloor \varepsilon \rfloor_\eta} [\overline{\lfloor \tau \rfloor_\eta}] \\
\lfloor \mathbf{pack}(\sigma, e)\ \mathbf{as}\ \exists\, \alpha :: \kappa.\ \tau \rfloor_\eta &= \begin{cases} (\lfloor \sigma \rfloor_\eta, \lfloor e \rfloor_\eta) & \text{when } C(\kappa) = \mathsf{E} \\ \lfloor e \rfloor_\eta & \text{otherwise} \end{cases} \\
\lfloor e_1\ e_2 \rfloor_\eta &= \lfloor e_1 \rfloor_\eta\ \lfloor e_2 \rfloor_\eta \\
\lfloor e\ \tau \rfloor_\eta &= \lfloor e \rfloor_\eta\ \lfloor \tau \rfloor_\eta \\
\lfloor \mathbf{unpack}\ e_1\ \mathbf{as}\ \alpha :: \kappa, x : \tau\ \mathbf{in}\ e_2 \rfloor_\eta &= \begin{cases} \mathbf{letp}\ (y, x) = \lfloor e_1 \rfloor_\eta\ \mathbf{in}\ \lfloor e_2 \rfloor_{\eta[\alpha \mapsto y]} & \text{when } C(\kappa) = \mathsf{E} \\ (\boldsymbol{\lambda}\, x.\, \lfloor e_2 \rfloor_{\eta[\alpha \mapsto ()]})\ \lfloor e_1 \rfloor_\eta & \text{otherwise} \end{cases} \\
\lfloor \mathbf{handle}_\varepsilon\ e\ \{\overline{h}; \mathbf{return}\ x : \tau \Rightarrow e'\} \rfloor_\eta &= \mathbf{handle}_{\lfloor \varepsilon \rfloor_\eta}\ \lfloor e \rfloor_\eta \{\overline{\lfloor h \rfloor_\eta}; \mathbf{return}\ x \Rightarrow \lfloor e' \rfloor_\eta\} \\
\lfloor \mathbf{effect}\ \alpha = \theta\ \mathbf{in}\ e \rfloor_\eta &= \mathbf{new}\ x\ \mathbf{in}\ \lfloor e \rfloor_{\eta[\alpha \mapsto x]} \\
\lfloor \langle c \rangle\ e \rfloor_\eta &= \langle \lfloor c \rfloor_\eta \rangle \lfloor e \rfloor_\eta \\
\lfloor o\ \overline{\alpha :: \kappa}\ (x : \sigma)/(r : \tau) \Rightarrow e \rfloor_\eta &= o[\overline{y}]\ x\ /\ r \Rightarrow \lfloor e \rfloor_{\eta\overline{[\alpha \mapsto y]}}
\end{aligned}$$

Fig. 8. Erasure

is, $C(\kappa) = \mathsf{E}$), the expression is translated to a $\boldsymbol{\lambda}$-abstraction, in which the bound variable (x) is intended to be instantiated with an effect label. Otherwise, the expression is translated to a thunk – note that an application to a type is translated to an application to a value.[2] In the case of **pack**, if α is an effect constructor, we translate the expression to a pair. The first element of the pair stores the effect constructor given originally in the first component of the **pack** expression. Otherwise, we ignore the **pack** construct and translate only the inner expression. Similarly with **unpack**, if α is an effect constructor, we use **letp** to match elements of the pair. Otherwise, the expression becomes the usual $\boldsymbol{\lambda}$-abstraction applied to the packed expression.

[2]Another standard approach would be to impose the value restriction in the programmer-level language and simply erase $\boldsymbol{\Lambda}$'s in the case $C(\kappa) \neq \mathsf{E}$. Indeed, this is how type polymorphism is implemented in Helium.

$$
\begin{array}{rcll}
v & ::= & \lambda^\rho x.e \mid s \mid o_l[\overline{s}] \mid (l, v) \mid \theta & \text{(machine value)} \\
\rho & ::= & \{\} \mid \rho\{x \mapsto v\} & \text{(environment)} \\
\kappa & ::= & \bullet \mid \iota : \kappa & \text{(stack)} \\
\iota & ::= & e^\rho_{\mathsf{A}} \mid v_{\mathsf{A}} \mid l_{\mathsf{P}} \mid e^{x,y,\rho}_{\mathsf{L}} & \text{(stack frame)} \\
\pi & ::= & \bullet \mid \delta : \pi & \text{(meta-stack)} \\
\delta & ::= & (\mu, \kappa) & \text{(meta-stack frame)} \\
\mu & ::= & c^\rho \mid \{\overline{h}; d\}^\rho_l & \text{(meta-stack marker)} \\
\theta & ::= & \bullet \mid \delta : \theta & \text{(reified meta-stack)}
\end{array}
$$

Fig. 9. Syntax of the abstract machine

Correctness of Type Erasure. We write $\eta : \Delta \rightarrowtail \Sigma$ to denote maps such that $\mathrm{dom}(\eta) = \mathrm{dom}(\Delta)$ and $\mathrm{cod}(\eta) = \Sigma \cup \{()\}$, for which it is the case that

$$
\begin{cases}
\eta(\alpha) = () & \text{if } \Delta \vdash \alpha :: \kappa \text{ and } C(\kappa) \in \{\mathsf{T}, \mathsf{R}\} \\
\eta(\alpha) \in l & \text{if } \Delta \vdash \alpha :: \kappa \text{ and } C(\kappa) = \mathsf{E}
\end{cases}
$$

and the latter part of η is injective. Additionally, we note that the function $\lfloor - \rfloor_\eta$ naturally extends to evaluation contexts.

Lemma 3.1. *Erasure distributes over decomposition, i.e.,* $\lfloor E[e] \rfloor_\eta = \lfloor E \rfloor_\eta[\lfloor e \rfloor_\eta]$.

Lemma 3.2. *If* $\eta : \Delta \rightarrowtail \Sigma$, n–free(α, E), *and* $\Delta; \cdot \vdash E : \tau_1 \;/\; \langle \alpha\ \overline{\sigma} \rangle \rightsquigarrow \tau_2 \;/\; \rho$, *then* n–free$(\eta(\alpha), \lfloor E \rfloor_\eta)$.

Lemma 3.3. *If* $\Delta; \cdot \vdash e : \tau \;/\; \rho$, $\Delta; e \longrightarrow \Delta'; e'$ *and* $\eta : \Delta \rightarrowtail \Sigma$, *then there exist* Σ' *and* $\eta' : \Delta' \rightarrowtail \Sigma'$ *such that* $\eta \subseteq \eta'$ *and* $\Sigma; \lfloor e \rfloor_\eta \longrightarrow \Sigma'; \lfloor e' \rfloor_{\eta'}$.

4 ABSTRACT MACHINE

Runtime systems for functional languages have been typically and most successfully modeled with abstract machines, i.e., first-order tail-recursive transition systems [Biernacka et al. 2005; Clements and Felleisen 2004; Clinger 1998; Cousineau et al. 1985; Felleisen 1988; Felleisen and Friedman 1986; Krivine 2007; Landin 1964; Leroy 1990; Marlow and Peyton Jones 2006; Peyton Jones 1992]. In this section we follow this tradition and present an abstract machine for the untyped calculus of Section 3 which through the type erasure translation provides a model implementation for the λ^{HEL}-calculus. The machine is based on the architecture of the definitional abstract machine for the control operators shift and reset [Biernacka et al. 2005]. The definitional abstract machine for shift and reset extends the CEK abstract machine [Felleisen and Friedman 1986], the canonical abstract machine for the call-by-value λ-calculus, with an additional layer of stack, called the meta-stack. The structure of the meta-stack in the abstract machine considered here is richer in that it contains stack markers [Dybvig et al. 2007] corresponding to coercions and handlers, that are dynamically explored in search of the right handler, whenever an operation is being handled. A CEK-based abstract machine for algebraic effects, albeit for a different calculus and with different design choices, has been presented in [Hillerström and Lindley 2016].

4.1 Syntax and Transitions

Syntax and configurations. The syntax of the abstract machine is presented in Figure 9. Expressions are inherited from the type-free calculus. Machine values v include closures ($\lambda^\rho x.e$), simple values, type-instantiated operations ($o_l[\overline{s}]$), pairs representing a concrete implementation of

$$\begin{array}{lr}
\langle e \mid \rho \mid \kappa \mid \pi \rangle_{\mathsf{eval}} & \text{(eval configuration)} \\
\langle \kappa \mid v \mid \pi \rangle_{\mathsf{stack}} & \text{(stack configuration)} \\
\langle \pi \mid v \rangle_{\mathsf{mstack}} & \text{(meta-stack configuration)} \\
\langle o_l[\overline{s}] \mid n \mid \kappa \mid \pi \mid v \mid \theta \rangle_{\mathsf{op}} & \text{(operation configuration)} \\
\langle \theta \mid \kappa \mid \pi \mid v \rangle_{\mathsf{res}} & \text{(resumption configuration)}
\end{array}$$

Fig. 10. Configurations of the abstract machine

an existential effect ((l, v)), and reified meta-stacks representing a captured continuation used to resume computation in operation handlers (θ).

The machine uses an environment ρ that maps variables to machine values. The empty environment is written $\{\}$, updating an environment is written $\rho\{x \mapsto v\}$, and looking up a variable in an environment is written $\rho(x)$. Given an environment ρ we define a *partial* map $\widehat{\rho}$ from values to machine values as $\widehat{\rho}(x) \triangleq \rho(x)$ and $\widehat{\rho}(s) \triangleq s$ (and undefined for other kinds of values).

A stack κ is a list of stack frames, where $\bullet$ represents the empty stack, and $\iota : \kappa$ is the result of pushing ι on the stack κ. The stack frames e^{ρ}_{A} and v_{A} represent the operand (an expression coupled with its environment) and the operator (a machine value) in the call-by-value evaluation of expression application, respectively. The stack frame l_{p} represents the return information for evaluating the second component of a pair, whereas the stack frame $e^{x,y,\rho}_{\mathsf{L}}$ is used for evaluating local definitions.

A meta-stack π is a list of meta-stack frames, where $\bullet$ represents the empty meta-stack, and $\delta : \pi$ is the result of pushing δ on the meta-stack π. A meta-stack frame (μ, κ) consists of a stack marker μ, i.e., either a coercion closure c^{ρ} or a handler closure $\{\overline{h}; d\}^{\rho}_{l}$, and a stack κ. Since in the calculi we consider we do not assume a top-level handler, it is not possible to represent the stack as a list of frames terminated with a marker, and the meta-stack as a list of such stacks, as e.g., in [Biernacka et al. 2005]. Instead we represent the complete control stack as a pair κ_1 and $(\mu_1, \kappa_2) : \cdots : (\mu_n, \kappa_{n+1}) : \bullet$, where μ_i separates κ_i and κ_{i+1}. A reified meta-stack θ happens to have the same structure as a meta-stack, but it is interpreted differently, as explained later on.

The abstract machine operates in five modes, shown in Figure 10. The modes eval, stack and mstack form the core of the abstract machine and are mostly standard [Biernacka et al. 2005] – they cooperatively interpret expressions, stacks, and meta-stacks, respectively. The remaining modes play an auxiliary role. A configuration $\langle o_l[\overline{s}] \mid n \mid \kappa \mid \pi \mid v \mid \bullet \rangle_{\mathsf{op}}$ represents the process of searching the meta-stack π for the right handler for the operation o of an effect l, using a counter n that is suitably modified by the encountered meta-stack markers, and accumulating the traversed meta-stack (in reversed order) in θ. When in a configuration $\langle \theta \mid \kappa \mid \pi \mid v \rangle_{\mathsf{res}}$, the machine resumes the reified meta-stack θ, recursively concatenating it with the current control stack.

Transitions. The transitions of the abstract machine are presented in Figure 11 and Figure 12, and are labeled as administrative ($\Rightarrow_{\mathsf{a}}$), reducing ($\Rightarrow_{\beta_i}$), handler searching or context capturing ($\Rightarrow_{\mathsf{o}}$), and context resuming ($\Rightarrow_{\mathsf{r}}$).[3] We define $\Rightarrow$ as the union of all these relations. The evaluation of an expression e starts the machine in the initial configuration $\langle e \mid \{\} \mid \bullet \mid \bullet \rangle_{\mathsf{eval}}$, whereas the result v of evaluation is unloaded from the final configuration $\langle \bullet \mid v \rangle_{\mathsf{mstack}}$. Evaluating an expression e can

[3]Technically speaking, the transitions $\Rightarrow_{\beta_4}$ and $\Rightarrow_{\beta_5}$ do not correspond by themselves to a reduction in the calculus, but rather they trigger a terminating subcomputation that implements the reduction using the transitions $\Rightarrow_{\mathsf{o}}$ for β_4, and $\Rightarrow_{\mathsf{r}}$ for β_5.

$$
\begin{array}{rcl}
e & \Rightarrow_{\mathrm{a}} & \langle e \mid \{\} \mid \bullet \mid \bullet \rangle_{\mathsf{eval}} \\[1em]
\langle x \mid \rho \mid \kappa \mid \pi \rangle_{\mathsf{eval}} & \Rightarrow_{\mathrm{a}} & \langle \kappa \mid v \mid \pi \rangle_{\mathsf{stack}} \\
 & & \text{where } v = \rho(x) \\
\langle \boldsymbol{\lambda}\, x\boldsymbol{.}\, e \mid \rho \mid \kappa \mid \pi \rangle_{\mathsf{eval}} & \Rightarrow_{\mathrm{a}} & \langle \kappa \mid \lambda^{\rho} x.e \mid \pi \rangle_{\mathsf{stack}} \\
\langle s \mid \rho \mid \kappa \mid \pi \rangle_{\mathsf{eval}} & \Rightarrow_{\mathrm{a}} & \langle \kappa \mid s \mid \pi \rangle_{\mathsf{stack}} \\
\langle o_{v'}[\overline{v}] \mid \rho \mid \kappa \mid \pi \rangle_{\mathsf{eval}} & \Rightarrow_{\mathrm{a}} & \langle \kappa \mid o_l[\overline{s}] \mid \pi \rangle_{\mathsf{stack}} \\
 & & \text{when } \widehat{\rho}(v') = l \text{ and } \overline{\widehat{\rho}(v) = s} \\
\langle e_1\, e_2 \mid \rho \mid \kappa \mid \pi \rangle_{\mathsf{eval}} & \Rightarrow_{\mathrm{a}} & \langle e_1 \mid \rho \mid {e_2}^{\rho}_{\mathsf{A}} : \kappa \mid \pi \rangle_{\mathsf{eval}} \\
\langle (v, e) \mid \rho \mid \kappa \mid \pi \rangle_{\mathsf{eval}} & \Rightarrow_{\mathrm{a}} & \langle e \mid \rho \mid l_{\mathsf{P}} : \kappa \mid \pi \rangle_{\mathsf{eval}} \\
 & & \text{where } \widehat{\rho}(v) = l \\
\langle \mathbf{letp}\ (x, y) = e_1\ \mathbf{in}\ e_2 \mid \rho \mid \kappa \mid \pi \rangle_{\mathsf{eval}} & \Rightarrow_{\mathrm{a}} & \langle e_1 \mid \rho \mid {e_2}^{x,y,\rho}_{\mathsf{L}} : \kappa \mid \pi \rangle_{\mathsf{eval}} \\
\langle \mathbf{handle}_v\ e\{\overline{h}; d\} \mid \rho \mid \kappa \mid \pi \rangle_{\mathsf{eval}} & \Rightarrow_{\mathrm{a}} & \langle e \mid \rho \mid \bullet \mid (\{\overline{h}; d\}^{\rho}_{l}, \kappa) : \pi \rangle_{\mathsf{eval}} \\
 & & \text{when } \widehat{\rho}(v) = l \\
\langle \langle c \rangle e \mid \rho \mid \kappa \mid \pi \rangle_{\mathsf{eval}} & \Rightarrow_{\mathrm{a}} & \langle e \mid \rho \mid \bullet \mid (c^{\rho}, \kappa) : \pi \rangle_{\mathsf{eval}} \\
\langle \mathbf{new}\ x\ \mathbf{in}\ e \mid \rho \mid \kappa \mid \pi \rangle_{\mathsf{eval}} & \Rightarrow_{\beta_1} & \langle e \mid \rho\{x \mapsto l\} \mid \kappa \mid \pi \rangle_{\mathsf{eval}} \\
 & & \text{where } l \text{ fresh} \\[1em]
\langle \bullet \mid v \mid \pi \rangle_{\mathsf{stack}} & \Rightarrow_{\mathrm{a}} & \langle \pi \mid v \rangle_{\mathsf{mstack}} \\
\langle l_{\mathsf{P}} : \kappa \mid v \mid \pi \rangle_{\mathsf{stack}} & \Rightarrow_{\mathrm{a}} & \langle \kappa \mid (l, v) \mid \pi \rangle_{\mathsf{stack}} \\
\langle e^{x,y,\rho}_{\mathsf{L}} : \kappa \mid (l, v) \mid \pi \rangle_{\mathsf{stack}} & \Rightarrow_{\beta_2} & \langle e \mid \rho\{x \mapsto l\}\{y \mapsto v\} \mid \kappa \mid \pi \rangle_{\mathsf{eval}} \\
\langle e^{\rho}_{\mathsf{A}} : \kappa \mid v \mid \pi \rangle_{\mathsf{stack}} & \Rightarrow_{\mathrm{a}} & \langle e \mid \rho \mid v_{\mathsf{A}} : \kappa \mid \pi \rangle_{\mathsf{eval}} \\
\langle \lambda^{\rho} x.e_{\mathsf{A}} : \kappa \mid v \mid \pi \rangle_{\mathsf{stack}} & \Rightarrow_{\beta_3} & \langle e \mid \rho\{x \mapsto v\} \mid \kappa \mid \pi \rangle_{\mathsf{eval}} \\
\langle o_l[\overline{s}]_{\mathsf{A}} : \kappa \mid v \mid \pi \rangle_{\mathsf{stack}} & \Rightarrow_{\beta_4} & \langle o_l[\overline{s}] \mid 0 \mid \kappa \mid \pi \mid v \mid \bullet \rangle_{\mathsf{op}} \\
\langle \theta_{\mathsf{A}} : \kappa \mid v \mid \pi \rangle_{\mathsf{stack}} & \Rightarrow_{\beta_5} & \langle \theta \mid \kappa \mid \pi \mid v \rangle_{\mathsf{res}} \\[1em]
\langle (\{\overline{h}; \mathbf{return}\ x \Rightarrow e\}^{\rho}_{l}, \kappa) : \pi \mid v \rangle_{\mathsf{mstack}} & \Rightarrow_{\beta_6} & \langle e \mid \rho\{x \mapsto v\} \mid \kappa \mid \pi \rangle_{\mathsf{eval}} \\
\langle (c^{\rho}, \kappa) : \pi \mid v \rangle_{\mathsf{mstack}} & \Rightarrow_{\beta_7} & \langle \kappa \mid v \mid \pi \rangle_{\mathsf{stack}} \\
\langle \bullet \mid v \rangle_{\mathsf{mstack}} & \Rightarrow_{\mathrm{a}} & v
\end{array}
$$

Fig. 11. Core transitions of the abstract machine

either yield a value v, i.e., $e \Rightarrow^* v$, or diverge, written $e \Uparrow$, or it can get stuck, e.g., searching for a non existing handler.

The interesting transitions in the eval mode are the ones that concern algebraic effects. In particular, evaluating a local definition of an effect amounts to generating a fresh effect name and binding it with the locally defined variable, where a label is considered fresh when it does not occur in the configuration under consideration. Dealing with handlers and operations is more involved and actually determines the overall structure of the abstract machine. When a handler expression or a coerced expression is processed by the machine, a new meta-stack frame is created and pushed on the meta-stack, whereas the stack is reset, which corresponds exactly to the way an abstract machine for delimited continuations would treat a control delimiter [Biernacka et al. 2005]. Transitions from the mstack mode correspond to an "effect-free" return of a value by a "delimited" computation. There are two transitions from the stack mode that require some attention: an application of an

$$
\begin{array}{rcl}
\langle o_l[\overline{s}] \mid 0 \mid \kappa \mid (\{\overline{h}; d\}^{\rho}_{l}, \kappa') : \pi \mid v \mid \theta\rangle_{\mathsf{op}} & \Rightarrow_{\mathsf{o}} & \langle e \mid \rho' \mid \kappa' \mid \pi\rangle_{\mathsf{eval}} \\
& & \text{where } o[\overline{y}]\ x\ /\ r \Rightarrow e \in h \\
& & \text{and } \rho' = \rho\{\overline{y} \mapsto \overline{s}\}\{x \mapsto v\}\{r \mapsto (\{\overline{h}; d\}^{\rho}_{l}, \kappa) : \theta\} \\
\langle o_l[\overline{s}] \mid n \mid \kappa \mid (\{\overline{h}; d\}^{\rho}_{l}, \kappa') : \pi \mid v \mid \theta\rangle_{\mathsf{op}} & \Rightarrow_{\mathsf{o}} & \langle o_l[\overline{s}] \mid n-1 \mid \kappa' \mid \pi \mid v \mid (\{\overline{h}; d\}^{\rho}_{l}, \kappa) : \theta\rangle_{\mathsf{op}} \\
& & \text{if } n \neq 0 \\
\langle o_l[\overline{s}] \mid n \mid \kappa \mid (\{\overline{h}; d\}^{\rho}_{l'}, \kappa') : \pi \mid v \mid \theta\rangle_{\mathsf{op}} & \Rightarrow_{\mathsf{o}} & \langle o_l[\overline{s}] \mid n \mid \kappa' \mid \pi \mid v \mid (\{\overline{h}; d\}^{\rho}_{l'}, \kappa) : \theta\rangle_{\mathsf{op}} \\
& & \text{if } l \neq l' \\
\langle o_l[\overline{s}] \mid n \mid \kappa \mid (c^{\rho}, \kappa') : \pi \mid v \mid \theta\rangle_{\mathsf{op}} & \Rightarrow_{\mathsf{o}} & \langle o_l[\overline{s}] \mid m \mid \kappa' \mid \pi \mid v \mid (c^{\rho}, \kappa) : \theta\rangle_{\mathsf{op}} \\
& & \text{if } l : n \overset{c^{\rho}}{\rightsquigarrow} m \\
\\
\langle \bullet \mid \kappa \mid \pi \mid v\rangle_{\mathsf{res}} & \Rightarrow_{\mathsf{r}} & \langle \kappa \mid v \mid \pi\rangle_{\mathsf{stack}} \\
\langle (\mu, \kappa') : \theta \mid \kappa \mid \pi \mid v\rangle_{\mathsf{res}} & \Rightarrow_{\mathsf{r}} & \langle \theta \mid \kappa' \mid (\mu, \kappa) : \pi \mid v\rangle_{\mathsf{res}}
\end{array}
$$

Fig. 12. Operation and resumption transitions of the abstract machine

operation that switches the mode to op and an application of a reified meta-stack that switches the mode to res.

When the machine is in the op-mode, it searches the first handler for a given operation o (of an effect l) in the meta-stack for which the counter n is equal 0. Whenever a handler for l is encountered but the counter is not equal 0, it is decremented, and whenever a coercion is encountered, the counter is modified accordingly. The auxiliary relation $l : n \overset{c^{\rho}}{\rightsquigarrow} m$ means $l : n \overset{c'}{\rightsquigarrow} m$ for a coercion c' that corresponds to the coercion closure c^{ρ}.[4] During the search of appropriate handler, the traversed meta-context is being accumulated (in reversed order) and finally it is stored in the environment as the resume argument of the operation handler. When the machine is in the res-mode, the captured meta-stack $(\mu_n, \kappa_n) : \cdots : (\mu_1, \kappa_1) : \bullet$ is recursively pushed frame by frame on the current control stack given by κ and π, yielding a new control stack formed by κ_1 and $(\mu_1, \kappa_2) : \cdots : (\mu_n, \kappa) : \pi$.

4.2 Correctness

In this section we sketch the correctness proof of the abstract machine with respect to the reduction semantics of the untyped calculus. Our approach is fairly standard and it follows quite closely the developments presented in [Hillerström and Lindley 2016], with some variations that we find appropriate in the case of our abstract machine. The idea is to view the configurations comprising the core of the abstract machine, i.e., eval, stack and mstack, as expressions, and to relate transitions on these configurations with reductions on the corresponding expressions. To this end we define a family of *decompilation* functions $(\!|\cdot|\!)$ that map the eval, stack and mstack configurations to expressions (decompilation is *undefined* for the remaining configurations which play only a role of auxiliary and always terminating sub-machines):

$$
\begin{array}{rcl}
(\!|\langle e \mid \rho \mid \kappa \mid \pi\rangle_{\mathsf{eval}}|\!) & = & (\!|\pi|\!)_{\mathsf{m}}[(\!|\kappa|\!)_{\mathsf{s}}[e\ (\!|\rho|\!)_{\mathsf{e}}]] \\
(\!|\langle \kappa \mid v \mid \pi\rangle_{\mathsf{stack}}|\!) & = & (\!|\pi|\!)_{\mathsf{m}}[(\!|\kappa|\!)_{\mathsf{s}}[(\!|v|\!)_{\mathsf{v}}]] \\
(\!|\langle \pi \mid v\rangle_{\mathsf{mstack}}|\!) & = & (\!|\pi|\!)_{\mathsf{m}}[(\!|v|\!)_{\mathsf{v}}]
\end{array}
$$

[4]For simplicity, we consider the process of evaluation of coercion as a meta-function of the machine. In fact, this step require linear time with rescpect to the size of the coercion. In Appendix we present a more realistic version of the machine, where this process is implemented by additional machine transitions.

where (formal definitions omitted) $(\!|\kappa|\!)_{\mathsf{s}}$ and $(\!|\pi|\!)_{\mathsf{m}}$ yield evaluation contexts represented by κ and π, $(\!|\nu|\!)_{\mathsf{v}}$ yields a corresponding value in the calculus (recursively turning θ into a λ-abstraction representing the captured context), and $(\!|\rho|\!)_{\mathsf{e}}$ yields a substitution of values for variables as given in ρ.

We define $\Rightarrow_{\mathsf{or}}$ as the union of $\Rightarrow_{\mathsf{o}}$ and $\Rightarrow_{\mathsf{r}}$ relations, and $\Rightarrow_{\beta}$ as the union of $\Rightarrow_{\beta_i}$ for $1 \leq i \leq 7$. Then, we show some selected lemmas that identify the role of the $\Rightarrow_{\mathsf{o}}$ and $\Rightarrow_{\mathsf{r}}$ transitions as, respectively, context capturing and context resuming. (When stating a property of a reduction semantics configuration $\Sigma; e$, we tacitly assume that all labels occurring in e are listed in Σ – an invariant that is obviously maintained by the reduction semantics.)

Lemma 4.1. *If $\Sigma; e \longrightarrow \Sigma'; e'$ and $(\!|\gamma|\!) = e$, where $\gamma = \langle o_l[\bar{s}]_{\mathsf{A}} : \kappa \mid \nu \mid \pi \rangle_{\mathsf{stack}}$, then there exists γ' such that $\gamma \Rightarrow_{\beta_4} \Rightarrow_{\mathsf{o}}^{*} \gamma'$ and $(\!|\gamma'|\!) = e'$.*

Lemma 4.2. *If $\Sigma; e \longrightarrow \Sigma'; e'$ and $(\!|\gamma|\!) = e$, where $\gamma = \langle \theta_{\mathsf{A}} : \kappa \mid \nu \mid \pi \rangle_{\mathsf{stack}}$, then there exists γ' such that $\gamma \Rightarrow_{\beta_5} \Rightarrow_{\mathsf{r}}^{*} \gamma'$ and $(\!|\gamma'|\!) = e'$.*

Using these and similar lemmas covering other reduction rules, we can prove a main lemma that gives a forward simulation result, i.e., that the abstract machine simulates the reduction semantics.

Lemma 4.3. *If $\Sigma; e \longrightarrow \Sigma'; e'$, then for all γ such that $(\!|\gamma|\!) = e$, there exists γ' such that $\gamma \Rightarrow_{\mathsf{a}}^{*} \Rightarrow_{\beta} \Rightarrow_{\mathsf{or}}^{*} \gamma'$ and $(\!|\gamma'|\!) = e'$.*

This lemma immediately yields the following theorem that both successful as well as divergent evaluations in reduction semantics are reflected by the abstract machine.

Theorem 4.4. *If $\Sigma; e \longrightarrow^{*} \Sigma'; v$, then there exists ν such that $e \Rightarrow^{*} \nu$ and $(\!|\nu|\!)_{\mathsf{v}} = v$. If $\Sigma; e \uparrow$ diverges, then $e \Uparrow$.*

Since there are no infinite $\Rightarrow_{\mathsf{a}}$ or $\Rightarrow_{\mathsf{or}}$ transition sequences, a converse theorem holds as well:

Theorem 4.5. *If $e \Rightarrow^{*} \nu$ and $(\!|\nu|\!)_{\mathsf{v}} = v$, then for all Σ, there exists Σ' such that $\Sigma; e \longrightarrow^{*} \Sigma'; v'$, where v and v' are equal modulo (generated) effect labels. If $e \Uparrow$, then $\Sigma; e$ diverges for all Σ.*

5 IMPLEMENTATION

To appreciate effect abstraction (or, actually, any kind of abstraction, such as modules or abstract data types), one usually needs to work through a larger project, where modularity, separation of concerns, and code reuse are essential aspects of the internal design of the system. Moreover, since algebraic effects and handlers are a fairly novel addition to the functional programming landscape, the pragmatics of employing them in such larger projects is still a vast area to explore. Thus, to allow more experimentation with effect abstraction and the coercion-based semantics that we propose in this paper, we have implemented an experimental programming language, tentatively named Helium. The language supports advanced algebraic effects and handlers, sophisticated parametric polymorphism (including polymorphic records and constructors of algebraic data types), and type and effect abstraction through an ML-style module system with signatures and functors.

One (not very surprising) observation that we made when playing around with abstract effects is that it is rather inconvenient for the programmer to insert the necessary coercions manually – indeed, Biernacki et al. [2018] note that their *lift* coercion is more of a semantics-level artifact than a surface-level construct. For this reason, Helium incorporates a notion of subtyping, which is much more natural to work with for the programmer than explicit coercions. Some applications of subtyping rules in type derivations can be reified as coercions during type inference, while some can be erased altogether – a similar in spirit approach was taken by Saleh et al. [2018], although the coercions that they propose serve a different purpose, and do not have any computational content.

Practice shows that in this way we are able to handle reasonably large examples with no overhead for the programmer whatsoever.

6 EXTENDED EXAMPLE

In this section, we present an extended example of a program that uses abstract algebraic effects. Our aim is to provide a feel for how the abstraction can be used to hide unnecessary detail and ensure that the user cannot break the contract that the implementer of the effectful algorithm relies on. The algorithm we present is a version of Huet's unification algorithm that uses a union-find based disjoint set data structure in order to avoid unifying the same terms multiple times [Huet 1976]. The algorithm is adapted from Knight's survey [Knight 1989], although in the interest of briefness we do not implement the acyclicity check, thus allowing for infinite unifiers.[5]

We consider a unification problem for terms over some signature, represented by a type constructor `Sig : type -> type` with variables represented by a type `Var`, given by the following definition:

```
data rec Term Sig Var = Var of Var | Term of Sig (Term Sig Var)
```

We assume that variables can be compared for equality, and that we have two functions, `fmap` and `zipWith`, of the following types:

```
val fmap    : (a ->[|r] b) -> Sig a ->[|r] Sig b
val zipWith : (a -> b ->[|r] c) -> Sig a -> Sig b ->[Error | r] Sig c
```

The former of these is a simple generalization of `map` to our type constructor `Sig`. The latter takes a function and two structures, and applies the combining function under the function symbol *provided that* the given function symbols agree. If the symbols do not agree an error is raised, as seen in the effect ascription of the function. Note that both the mapped function and the combiner given to `zipWith` can themselves use algebraic effects, and that these are retained by the resulting computation.

In order to implement Huet's unification, we need a union-find based disjoint set data structure. We have already presented its interface in the Introduction, let us recall it here:

```
type   Set : type -> type
effect UF  : type -> effect

val new    : a ->[UF a] Set a
val find   : Set a ->[UF a] a
val union  : (a -> a ->[| r] a) -> Set a -> Set a ->[UF a | r] Unit
val withUF : (Unit ->[UF a | r] b) ->[| r] b
```

The idea behind this specification is that each disjoint set `Set a` has a representative of type `a`, which is given to it at creation (`new`) and can be retrieved using `find`. Moreover, `union` takes a function that determines how the representatives of two disjoint sets will be merged, and performs the operation. Note that since unannotated arrows in λ^{HEL} are pure and the return type of `union` is trivial, it *has to* be effectful. Likewise, in usual implementations both `new` and `find` perform some computational effects. We capture all these effects in an abstract effect `UF a`, modeled in the calculus as an existential quantifier over effects, thus hiding the actual implementation choices from the user – in our case, the unification algorithm. The novel aspect is the presence of the handler, `withUF`, which *removes* the `UF` effect from a computation. Thus, we can use the union-find data structure locally, and expose a *pure* interface to the clients – a notable improvement over traditional ML, where we could never guarantee that a computation is pure.

[5]This check does not pose additional problems, but it does add some noise.

```
let unify (type Sig) t1 t2 =
  data rec UTerm = UTerm of Set (Option (Sig UTerm))
  let rec walkTree t =
    UTerm
    match t with
    | Var x  => assoc x (fn () => new None)
    | Term f => new (Some (fmap walkTree f))
    end
  let addAndPick a b = addToSet (a, b); a
  let uniteSyms s1 s2 =
    match s1, s2 with
    | None, _ => s2
    | _, None => s1
    | Some f1, Some f2 =>
      Some (zipWith addAndPick f1 f2)
    end
  let process p =
    let (UTerm s1, UTerm s2) = p in
    union uniteSyms s1 s2
  in
  handle addToSet (walkTree t1, walkTree t2)
  with   processSet process $> withAssocList $> withUF $>
    handle
    | return () => True
    | error  () => False
    end
```

Fig. 13. Huet-style unification procedure in Helium. The $> operator composes handlers sequentially.

In addition to the union-find module above, we use some more standard, non-abstract effects. In addition to the well-known `Error` (or failure) effect, in which the operation is given a polymorphic return type to avoid having to explicitly eliminate the empty type, we define two effects that embody common programming patterns: using a work set and a association map shared by the computation. The `WorkSet` effect is isomorphic with the common writer effect, while `Assoc` is obviously a particular form of state – but making these explicit cleans up the resulting unification code immensely. Note that the `assoc` operation takes as arguments both the key, and the suspended computation that would provide the value that would get associated with the key should it be absent in the map.

```
effect Error       = { error    : type T. Unit => T}
effect WorkSet T   = { addToSet : T => Unit }
effect Assoc K R V = { assoc    : K, (Unit ->[|R] V) => V }
```

We define handlers for the work set and the association map, which we treat here as library functions, and omit their code. The association map handler is specialized for the type of variables in our unification problem.

```
val processSet    : (t ->[WorkSet t | r] Unit) -> (Unit ->[WorkSet t | r] Unit) ->[| r] Unit
val withAssocList : (Unit ->[Assoc Var [| r] v | r] a) ->[| r] a
```

Now we can proceed to the unification procedure, presented in Figure 13. First, we define the local representation of terms as disjoint sets, the representatives of which are either `None`, if it is associated with a variable, or an element of the signature, with disjoint sets as subterms. This representation is at the crux of the algorithm, as it ensures that we do not reconsider terms that have already been unified (and so belong to the same set). Then, we define the function that converts the input representation to the internal one. For variables, we use the `assoc` operation to ensure that all occurrences of the variable are associated with the same set, creating new ones as needed; for function symbols, we simply create a new set and proceed down the tree using `fmap`. The function has two latent effects: `Assoc` and `UF`. Next, we define the function `uniteSyms`, which is used to pick the representative when uniting two disjoint sets. If one of the sets denotes a variable, we simply pick the other representative; however, when both are function symbols, we need to ensure that the subterms are unifiable. This is accomplished through the use of `zipWith` operation with a convenience function that takes the two subterms and adds the pair to the work set of pairs that need to be unified (and picks arbitrary one as a representative). Recall that `zipWith` raises an error if the two symbols do not match, which is precisely what we want: in that case, the original terms were not unifiable! The `uniteSyms` function is then used by the `process`, which simply calls `union` on the pair of sets. Note that the function passed as the argument is quite effectful: its latent effects include `Error` and `WorkSet`, while `process` itself adds the `UF` abstract effect to the above three. With all these auxiliary procedures defined, completing the unification is quite straightforward: we need to transform the input trees to the internal representation, treat the resulting pair as the initial element of the work set for which `process` encodes the task to be performed, and handle the resulting effects: `Assoc` and `WorkSet` are actually independent and we can choose either order, but both have to be handled before we use `withUF`, which provides the (abstract, from this perspective) interpretation to the effect `UF`. We compose these three sequentially with a (definable) operator `$>`, which saves us the boilerplate of nested handle/with constructs.

This leaves us with `Error` – and, due to the use of the `WorkSet`, with no meaningful return value. However, recall that `Error` was signaled precisely when unification failed: thus, it suffices to handle it by treating the `error` operation as failure, while a return (with a trivial value) – as a success, giving us the final result. This is encoded in the final handler of the function.

Note that we have omitted any coercions that would be necessary in the full syntax of our calculus, as it is clear from the context which of them should appear where. We treat this form omission as syntactic sugar that in practice allows us to write most programs without mentioning lifting or swapping effects at all.

7 DISCUSSION

To our knowledge, the issue of abstraction in languages with algebraic effects has not been discussed in the literature before, with the exception of a technical report by Leijen [2018] where they are introduced, but not developed theoretically. The two mentioned languages with row-based effects, Links [Hillerström and Lindley 2016] and Koka [Leijen 2017], are both equipped with (undocumented) module systems, but they give only a weak form of abstraction, offering no more than namespace management, akin, for example, Haskell [Peyton Jones 2003]. Among other languages with algebraic effects, but whose type systems do not rely on rows of effects are Eff [Bauer and Pretnar 2015] and Frank [Lindley et al. 2017]. Based on the related literature and the language documentation, the two languages do not seem to offer a module system at the moment. It is a matter of future work to investigate if the ideas shown in this work can be transferred to languages without row-based effects.

On the other hand, Eff provides a form of abstraction via *effect instances*. A new instance is created with the `new` keyword. The instance is a first-class value that can be associated with an operation and a handler clause. This way, one can obtain a form of local effects. The downside of effect instances is that in general it is not possible to statically decide which instance is associated with a given operation or a handler, which means that the type system is unable to keep track of which effects are handled. To our understanding, this is in accordance with Eff's principles, since its type system underapproximates the set of effects used by an expression (see [Bauer and Pretnar 2014]), while row-based systems overapproximate the effects.

As seen in Section 6, the places where we need to insert coercions are very often clear from context. This suggests an interesting direction for future research that could focus on the pragmatics of the design of a high-level interface to the relatively low-level calculus, where coercions are scrapped entirely.

REFERENCES

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

Malgorzata Biernacka, Dariusz Biernacki, and Olivier Danvy. 2005. An Operational Foundation for Delimited Continuations in the CPS Hierarchy. *Logical Methods in Computer Science* 1, 2:5 (2005), 1–39. https://doi.org/10.2168/LMCS-1(2:5)2005

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

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

John Clements and Matthias Felleisen. 2004. A tail-recursive machine with stack inspection. *ACM Trans. Program. Lang. Syst.* 26, 6 (2004), 1029–1052. https://doi.org/10.1145/1034774.1034778

William D. Clinger. 1998. Proper Tail Recursion and Space Efficiency. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI), Montreal, Canada, June 17-19, 1998*, Jack W. Davidson, Keith D. Cooper, and A. Michael Berman (Eds.). ACM, 174–185. https://doi.org/10.1145/277650.277719

Guy Cousineau, Pierre-Louis Curien, and Michel Mauny. 1985. The Categorical Abstract Machine. In *Functional Programming Languages and Computer Architecture, FPCA 1985, Nancy, France, September 16-19, 1985, Proceedings (Lecture Notes in Computer Science)*, Jean-Pierre Jouannaud (Ed.), Vol. 201. Springer, 50–64. https://doi.org/10.1007/3-540-15975-4_29

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

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

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

Robert Harper. 2016. *Practical Foundations for Programming Languages* (2nd ed.). Cambridge University Press, New York, NY, USA.

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

Gerard Huet. 1976. *Resolution d'equation dans les langages d'ordre 1, 2, …, ω.* Ph.D. Dissertation. Université de Paris VII, France.

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

Kevin Knight. 1989. Unification: A Multidisciplinary Survey. *ACM Comput. Surv.* 21, 1 (1989), 93–124. https://doi.org/10.1145/62029.62030

Jean-Louis Krivine. 2007. A call-by-name lambda-calculus machine. *Higher-Order and Symbolic Computation* 20, 3 (2007), 199–207. https://doi.org/10.1007/s10990-007-9018-9

Peter J. Landin. 1964. The Mechanical Evaluation of Expressions. *Comput. J.* 6, 4 (1964), 308–320.

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. 2017. Type directed compilation of row-typed algebraic effects, See [Castagna and Gordon 2017], 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.

Xavier Leroy. 1990. *The Zinc experiment: an economical implementation of the ML language.* Rapport Technique 117. INRIA Rocquencourt, Le Chesnay, France.

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

Simon Marlow and Simon L. Peyton Jones. 2006. Making a fast curry: push/enter vs. eval/apply for higher-order languages. *J. Funct. Program.* 16, 4-5 (2006), 415–449. https://doi.org/10.1017/S0956796806005995

Simon Peyton Jones (Ed.). 2003. *Haskell 98 Language and Libraries – The Revised Report.* Cambridge University Press, Cambridge, England.

Simon L. Peyton Jones. 1992. Implementing Lazy Functional Languages on Stock Hardware: The Spineless Tagless G-Machine. *J. Funct. Program.* 2, 2 (1992), 127–202. https://doi.org/10.1017/S0956796800000319

Benjamin C. Pierce. 2002. *Types and programming languages.* The MIT Press.

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

Gordon D. Plotkin and John Power. 2001. Semantics for Algebraic Operations. *Electr. Notes Theor. Comput. Sci.* 45 (2001), 332–345. https://doi.org/10.1016/S1571-0661(04)80970-8

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

Gordon D. Plotkin and Matija Pretnar. 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

Amr Hany Saleh, Georgios Karachalias, Matija Pretnar, and Tom Schrijvers. 2018. Explicit Effect Subtyping. In *ESOP 2018 (to appear)*.

Mads Tofte and Jean-Pierre Talpin. 1997. Region-based Memory Management. *Inf. Comput.* 132, 2 (1997), 109–176. https://doi.org/10.1006/inco.1996.2613

Andrew K. Wright and Matthias Felleisen. 1994. A Syntactic Approach to Type Soundness. *Inf. Comput.* 115, 1 (1994), 38–94. https://doi.org/10.1006/inco.1994.1093

A ABSTRACT MACHINE WITH COERCION TRANSITIONS

In this section we provide a version of abstract machine, that have extra transitions for interpreting coercions. Most of the machine is the same as in Section 4, so we describe only parts where they differ.

Syntax and configurations. The syntax of the abstract machine is extended by three new syntactic categories.

$$
\begin{array}{rcll}
\chi & ::= & \bullet \mid \sigma : \chi & \text{(coercion stack)} \\
\sigma & ::= & \circ \mid c & \text{(coercion stack frame)} \\
\psi & ::= & (o_l[\overline{s}], \kappa, \pi, \nu, \theta) & \text{(coercion meta-frame)}
\end{array}
$$

A coercion stack χ serves as a continuation in the process of evaluation of compound coercions, i.e., $c_1 \cdot c_2$ and $v : c$, with $\bullet$ representing the empty coercion stack, and $\sigma : \chi$ representing the result of pushing σ on the coercion stack χ. A coercion-stack frame $\circ$ is used to mark the stack when a $v : c$ is evaluated, whereas a frame c is used to sequentialize evaluation of the two sub-coercions in $c_1 \cdot c_2$. Additionally, the machine needs a coercion meta-frame ψ which represents the state from which the machine resumes searching a handler after a coercion has been processed.

The abstract machine needs two additional modes for interpreting coercions

$$
\begin{array}{ll}
\langle c \mid \rho \mid \chi \mid l \mid n \mid \psi \rangle_{\text{coerce}} & \text{(coercion configuration)} \\
\langle \chi \mid \rho \mid l \mid n \mid \psi \rangle_{\text{cstack}} & \text{(coercion stack configuration)}
\end{array}
$$

that play similar roles as eval and stack modes during evaluation of expression.

Transitions. The new machine when encounter a coercion in the op mode, enters the coerce mode in order to modify effect counter. We replace the last rule of transition $\Rightarrow_{\text{o}}$ with the following one.

$$
\langle o_l[\overline{s}] \mid n \mid \kappa \mid (c^{\rho}, \kappa') : \pi \mid \nu \mid \theta \rangle_{\text{op}} \quad \Rightarrow_{\text{o}} \quad \langle c \mid \rho \mid \bullet \mid l \mid n \mid (o_l[\overline{s}], \kappa', \pi, \nu, (c^{\rho}, \kappa) : \theta) \rangle_{\text{coerce}}
$$

Evaluating a coercion is done by an eval-continue sub-machine presented in Figure 14, with ψ playing the role of a dump as in, e.g., the SECD machine [Landin 1964]. These transitions implement the relation $l : n \overset{c}{\rightsquigarrow} m$ in Figure 7.

Correctness. In order to establish correctness of the machine we need to prove that the sub-machine for coercions implements the relation $l : n \overset{c}{\rightsquigarrow} m$:

Lemma A.1. *If $l : n \overset{c\,(\!|\rho|\!)_{\text{e}}}{\rightsquigarrow} m$, then $\langle c \mid \rho \mid \bullet \mid l \mid n \mid \psi \rangle_{\text{coerce}} \Rightarrow_{\text{c}}^{*} \langle \bullet \mid \rho \mid l \mid m \mid \psi \rangle_{\text{cstack}}$.*

We also have a lemma that identify the role of the $\Rightarrow_{\text{oc}}$ as context capturing.

Lemma A.2. *If $\Sigma; e \longrightarrow \Sigma'; e'$ and $(\!|\gamma|\!) = e$, where $\gamma = \langle o_l[\overline{s}]_{\text{A}} : \kappa \mid \nu \mid \pi \rangle_{\text{stack}}$, then there exists γ' such that $\gamma \Rightarrow_{\beta_4} \Rightarrow_{\text{oc}}^{*} \gamma'$ and $(\!|\gamma'|\!) = e'$.*

Rest of the proof is the same as for the machine from Section 4.

$$
\begin{array}{rcl}
\langle {\uparrow} v \mid \rho \mid \chi \mid l \mid n \mid \psi \rangle_{\text{coerce}} & \Rightarrow_{c} & \langle \chi \mid \rho \mid l \mid n+1 \mid \psi \rangle_{\text{cstack}} \\
 & & \text{when } \widehat{\rho}(v) = l \\
\langle {\uparrow} v \mid \rho \mid \chi \mid l \mid n \mid \psi \rangle_{\text{coerce}} & \Rightarrow_{c} & \langle \chi \mid \rho \mid l \mid n \mid \psi \rangle_{\text{cstack}} \\
 & & \text{when } \widehat{\rho}(v) = l' \text{ and } l' \neq l \\
\langle v_1 \leftrightarrow v_2 \mid \rho \mid \chi \mid l \mid 0 \mid \psi \rangle_{\text{coerce}} & \Rightarrow_{c} & \langle \chi \mid \rho \mid l \mid 1 \mid \psi \rangle_{\text{cstack}} \\
 & & \text{when } \widehat{\rho}(v_1) = \widehat{\rho}(v_2) = l \\
\langle v_1 \leftrightarrow v_2 \mid \rho \mid \chi \mid l \mid 1 \mid \psi \rangle_{\text{coerce}} & \Rightarrow_{c} & \langle \chi \mid \rho \mid l \mid 0 \mid \psi \rangle_{\text{cstack}} \\
 & & \text{when } \widehat{\rho}(v_1) = \widehat{\rho}(v_2) = l \\
\langle v_1 \leftrightarrow v_2 \mid \rho \mid \chi \mid l \mid n \mid \psi \rangle_{\text{coerce}} & \Rightarrow_{c} & \langle \chi \mid \rho \mid l \mid n \mid \psi \rangle_{\text{cstack}} \\
 & & \text{when } \widehat{\rho}(v_1) = l_1 \text{ and } \widehat{\rho}(v_2) = l_2 \\
 & & \text{and either } l_1 \neq l, l_2 \neq l \text{ or } n > 1 \\
\langle v : c \mid \rho \mid \chi \mid l \mid 0 \mid \psi \rangle_{\text{coerce}} & \Rightarrow_{c} & \langle \chi \mid \rho \mid l \mid 0 \mid \psi \rangle_{\text{cstack}} \\
 & & \text{when } \widehat{\rho}(v) = l \\
\langle v : c \mid \rho \mid \chi \mid l \mid n \mid \psi \rangle_{\text{coerce}} & \Rightarrow_{c} & \langle c \mid \rho \mid \circ : \chi \mid l \mid n-1 \mid \psi \rangle_{\text{coerce}} \\
 & & \text{when } \widehat{\rho}(v) = l \text{ and } n \neq 0 \\
\langle v : c \mid \rho \mid \chi \mid l \mid n \mid \psi \rangle_{\text{coerce}} & \Rightarrow_{c} & \langle c \mid \rho \mid \chi \mid l \mid n \mid \psi \rangle_{\text{coerce}} \\
 & & \text{when } \widehat{\rho}(v) = l' \text{ and } l' \neq l \\
\langle c_1 \cdot c_2 \mid \rho \mid \chi \mid l \mid n \mid \psi \rangle_{\text{coerce}} & \Rightarrow_{c} & \langle c_1 \mid \rho \mid c_2 : \chi \mid l \mid n \mid \psi \rangle_{\text{coerce}} \\
\\
\langle \circ : \chi \mid \rho \mid l \mid n \mid \psi \rangle_{\text{cstack}} & \Rightarrow_{c} & \langle \chi \mid \rho \mid l \mid n+1 \mid \psi \rangle_{\text{cstack}} \\
\langle c : \chi \mid \rho \mid l \mid n \mid \psi \rangle_{\text{cstack}} & \Rightarrow_{c} & \langle c \mid \rho \mid \chi \mid l \mid n \mid \psi \rangle_{\text{coerce}} \\
\langle \bullet \mid \rho \mid l \mid n \mid (o_l[\overline{s}], \kappa, \pi, \nu, \theta) \rangle_{\text{cstack}} & \Rightarrow_{c} & \langle o_l[\overline{s}] \mid n \mid \kappa \mid \pi \mid \nu \mid \theta \rangle_{\text{op}}
\end{array}
$$

Fig. 14. Coercion, and coercion-stack transitions of the abstract machine