A Type System for Effect Handlers and Dynamic Labels

Paulo Emílio de Vilhena and François Pottier

Inria Paris
`{paulo-emilio.de-vilhena,francois.pottier}@inria.fr`

Abstract. We consider a simple yet expressive λ-calculus equipped with references, effect handlers, and dynamic allocation of effect labels, and whose operational semantics does not involve coercions or rely on type information. We equip this language with a type system that supports type and effect polymorphism, allows reordering row entries and extending a row with new entries, and supports (but is not restricted to) lexically scoped handlers. This requires addressing the issue of potential aliasing between effect names. Our original solution is to interpret a row not only as a permission to perform certain effects but also as a disjointness requirement bearing on effect names. The type system guarantees strong type soundness: a well-typed program cannot crash or perform an unhandled effect. We prove this fact by encoding the type system into a novel Separation Logic for effect handlers, which we build on top of Iris. Our results are formalized in Coq.

1 Introduction

Effect handlers [30,17] can be viewed as a generalization of exception handlers. Like raising an exception, *performing an effect* interrupts the normal flow of execution and transfers control to a handler. Unlike an exception handler, an effect handler gains access to a *delimited continuation*, which represents the fragment of the evaluation context comprised between the point where the effect was performed and the point where the effect handler was installed. Invoking this continuation resumes the computation whose execution was suspended by performing an effect.

To allow programmers to exploit several independent effects simultaneously, it is desirable for effects to have *names*. Each effect handler handles a specific name, or a specific set of names. When an effect is performed, the name of this effect determines which handler is selected. This idea immediately gives rise to several key questions about names. What are they: strings, variables, addresses? Where are they defined? What is their scope?

In the simplest approach [2,14,22], effect names are global. All possible names are predefined and are in scope everywhere. This approach is simple but unsatisfactory in terms of expressiveness and modularity: an accidental collision, where two unrelated pieces of code happen to use the same effect name, can have surprising unintended consequences. We illustrate this problem later on (§2).

To remedy this problem, several authors have proposed to change the nature of names. Their work falls broadly in two categories: the “lexical approach” and the “generative approach”.

The “lexical approach” introduces local effect names with lexical scope. One can then think of an effect name essentially as a variable. *Tunneled exceptions* [42] and *lexically scoped handlers* [41,6,7,27] fall in this approach. In some of these proposals, the local effect name is never exposed to the user, but a “capability” to perform the effect is made available via a local variable. A potential pain point of this approach is that one must somehow ensure that a name or capability cannot escape its scope: this must be guaranteed by some combination of syntactic restrictions, runtime tests, and static typing rules.

The “generative approach” consists in allowing new effects to be generated afresh at runtime. This requires introducing a distinction between effect *labels*, which are allocated at runtime, and effect *names*, which are variables (with lexical scope) that the programmer uses to refer to effect labels. This is similar to the distinction between memory locations and variables that is traditionally used in the operational semantics of mutable references [29]. This approach has long been in use for exceptions in Standard ML [25] and OCaml [24], and is used also for effects in OCaml 5. It is powerful: in particular, it can simulate lexically scoped handlers.[1] However, it introduces several pitfalls of its own. First, it creates the possibility of nameless effects, that is, the possibility that there is no static effect name for a certain effect label. Second, it introduces the possibility of *aliasing* between effect names, that is, the possibility that two distinct effect names denote the same effect label. Aliasing creates a challenge for type system designers: if one cannot statically tell whether two effect names denote distinct labels, then it seems unclear how one can propose a sound and precise type discipline.

At least three ways of evading or addressing this challenge appear in the literature.

First, several mainstream languages adopt the generative approach but avoid the aliasing challenge by offering a weak type soundness guarantee: a well-typed program cannot crash, but can halt due to an unhandled exception or effect. This is the case in Standard ML, where exceptions are untracked, and in OCaml, where exceptions and effects are untracked. It is also the case in Eff [3].

Second, a number of authors evade or resolve the aliasing challenge by altering the syntax and the operational semantics of the language. Instead of letting the correspondence between an effect and a handler be determined purely by the notion of equality of effect labels or effect names, they introduce *coercions*

[1] This can be a source of confusion. A language that has “lexically scoped handlers” can, technically, be presented in either of these two styles. Biernacki et al. [6] present one semantics in each style, the “open semantics” and the “generative semantics”, and prove an equivalence between them. Zhang and Myers [41] adopt what we believe is a combination of lexically scoped handlers and implicit arguments, which they refer to as “tunneling”, in their surface language. This language is then translated down to a core language whose operational semantics is in the generative style.

that enable explicit disambiguation and collision avoidance. Examples include Koka [21] as well as several papers by Biernacki et al. [4,5].

Third, some authors evade the challenge by restricting the programming language in one or more ways, such as restricting attention to lexically scoped handlers [6,7] and forbidding first-class functions [7].

This sets the scene for this paper. We stick with the generative approach, which offers a simple and expressive semantics. We do not introduce coercions or otherwise alter the operational semantics. We do not restrict our attention to lexically scoped handlers. We address the aliasing challenge.

We propose TES, a type-and-effect system that statically rules out unhandled effects. As in most previous work, the potential effects of an expression are described by a *row*, a concept introduced to type-check records and variants [32,38] and later applied to the analysis of exceptions [28] and effects [14,22]. Type and effect polymorphism are supported. Furthermore, a simple and powerful subsumption relation allows reordering the entries in a row and extending a row with new entries, without any side conditions.

How is this possible? How is the aliasing challenge addressed? Our key idea is this: *whenever a question about aliasing arises, require absence of aliasing.* In other words, we interpret a row not just as a description of the names and types of the effects that may be performed, but also as *a requirement that these names be pairwise distinct*. For instance, if a typing judgment states that an expression e has effect $(s : \iota \Rightarrow \kappa) \cdot (s' : \iota' \Rightarrow \kappa')$, then this means not only that e may perform the effects s and s', but also that e *requires* the effect labels denoted by s and s' to be distinct. In the presence of effect polymorphism, if e has effect $(s : \iota \Rightarrow \kappa) \cdot \theta$, where θ is a row variable, then we take this to mean that e *requires* the effect label denoted by s to lie outside the set of effect labels denoted by θ. We adapt our typing and subtyping rules, where needed, so as to be sound with respect to this new interpretation of rows.

The reader may find our approach somewhat reminiscent of the manner in which the separating conjunction of Separation Logic [31] requires disjointness between the footprints of two formulae. Although this requirement may at first seem strong, experience has shown that Separation Logic is in fact concise and expressive. The examples that we present in Section 4.4 seem to suggest that our disjointness requirement is acceptable; we have not yet found examples where it is problematic. That said, we do not yet have practical experience with an implementation of this type system.

TES offers a strong type soundness guarantee: a well-typed program cannot crash and cannot halt due to an unhandled effect. To prove this fact, we follow a semantic approach that has become popular in the last few years [1,20,19]. We introduce TESLOGIC, a novel variant of Separation Logic, constructed on top of Iris [16], which allows reasoning about programs in the presence of effects and handlers, multi-shot continuations, and dynamic allocation of effect labels. We prove that this logic is sound, and we provide an interpretation of TES's typing rules in terms of TESLOGIC's reasoning rules. All of our results are formalized in Coq, and our Coq formalization is available [36].

In summary, the main contributions of this paper are the design of TES, a type system for TESLANG, a λ-calculus equipped with general references, effect handlers, and dynamic allocation of effect labels, and a proof of type soundness, which is carried out via a semantic interpretation into a new program logic, TESLOGIC.

In Section 2, we provide more background and examples about the semantics of effect handling: we discuss name collisions, effect coercions, lexically scoped handlers, and dynamic allocation of effect labels, and we justify why we wish to study a calculus where effect handling and dynamic allocation of effect labels are separate constructs. In Section 3, we present the syntax and operational semantics of TESLANG. In Section 4, we introduce TES and show a number of examples of constructions that TES is able to type-check. In Section 5, we present a brief overview of the proof of type soundness. Finally, we discuss the related work and conclude.

2 A Panorama of Semantics for Effect Handlers

The various mechanisms that we have mentioned so far, namely lexically scoped handlers, dynamic allocation of effect labels, and effect coercions, aim to resolve the basic problem of accidental collisions between effect names. Let us illustrate this problem with an example.

Anticipating on Section 3, we use a λ-calculus equipped with constructs to perform and handle effects. The expression `perform` $s\ v$ *performs an effect* with effect name s and payload v. The expression `handle` e `with` $s : h \mid r$ installs an effect handler which monitors the execution of the subexpression e and which handles the effects that carry the name s.[2] If e returns a value v, then the *return branch* r is invoked and receives the value v as an argument. If e performs an effect with name s and with payload v, then the execution of e is suspended and control is transferred to the *effect branch* h, which receives the payload v and a continuation k representing the suspended computation.

Let us now introduce the function `bad_counter`. In a system of simple types, which does not keep track of effects, `bad_counter` expects a function ff of type $(\alpha \to \beta) \to \gamma$ and returns a function of type $(\alpha \to \beta) \to \gamma \times \texttt{int}$. The *intended* behavior of `bad_counter` ff is to produce a new function ff' such that ff' behaves like ff but at the same time counts how many times ff uses its argument. That is, for an arbitrary function f, the application $ff'\ f$ is expected to return a pair (v, n), where v is the result of the computation $ff\ f$ and n is the number of invocations of f that have taken place during this computation. The function `bad_counter` is defined as follows:

$$\texttt{bad\_counter}\, ff \;=\; \lambda f.\, \begin{pmatrix} \texttt{handle}\, ff\ (\lambda x.\, \texttt{perform}\ tick\ ();\ f\ x)\ \texttt{with} \\ tick : \lambda\_\ k.\, \lambda n.\, k\ ()\ (n+1) \mid \lambda y.\, \lambda n.\, (y, n) \end{pmatrix}\ 0$$

This code has a free effect name, *tick*. The function f is wrapped in a proxy which performs an effect named *tick*. This effect is handled by `bad_counter`; the

[2] For simplicity, this construct selects just one name, as opposed to a set of names.

handler implements a memory cell (in state-passing style) to count the number of ticks, that is, the number of calls made by ff to f.

Unfortunately, because this function uses a fixed effect name, *tick*, it can exhibit an unintended behavior, caused by an accidental collision of effect names. The following use of `bad_counter` exhibits this issue:

$$\texttt{bad\_counter}\ (\texttt{bad\_counter}\ (\lambda f.\, f\ ()))\ (\lambda\_.\, ())$$

Because the function $\lambda f.\, f\ ()$ calls its argument once, one might expect the above expression to return $(((), 1), 1)$. Its actual result, however, is $(((), 2), 0)$. In the interest of space, we omit an explanation of its operational behavior. The key reason why it behaves incorrectly is that *the two instances of* `bad_counter` *use the same effect name*. Each application of `bad_counter` installs a handler for the effect name *tick*. One handler is nested inside the other. As a result, the innermost handler intercepts two *tick* effects and the outermost handler never observes any effect, whereas what was naively intended was that each handler observes and handles one effect. As a result of the name collision, one of the effects is *accidentally handled* by the innermost handler.

To avoid or help avoid accidental collisions between names, the literature describes several mechanisms: (1) effect coercions, (2) lexically scoped handlers, which can be viewed as a restricted case of (3) dynamic allocation of effect labels. Let us now say a little more about these mechanisms.

Effect coercions. An effect coercion modifies the manner in which an effect is matched with one of the enclosing handlers. Perhaps the simplest example is that of the `lift` coercion [4,5], but there are other forms of coercions in the literature, such as `swap`. Normally, performing an effect named s transfers control to the innermost enclosing handler that selects the name s. However, in a language with effect coercions, if there is a `lift` coercion between the point where the effect is performed and the innermost enclosing handler, then this handler is *skipped* and control is transferred instead to the next enclosing handler for the name s.[3] Under such a semantics, a coercion can be employed to write a fixed version of `bad_counter`:

$$\begin{array}{l}\texttt{lift\_counter}\ \mathit{ff}\ = \\ \qquad \lambda f.\left(\begin{array}{l}\texttt{handle}\ \mathit{ff}\ (\lambda x.\,\texttt{perform}\ \mathit{tick}\ ();\ \texttt{lift}\ \mathit{tick}\ (f\ x))\ \texttt{with} \\ \mathit{tick} : \lambda\_\ k.\, \lambda n.\, k\ ()\ (n+1) \mid \lambda y.\, \lambda n.\, (y, n)\end{array}\right)\ 0\end{array}$$

As desired, `lift_counter (lift_counter` $(\lambda f.\ f\ ()))\ (\lambda\_.\ ())$ returns the value $(((), 1), 1)$. One *tick* effect is intercepted by the innermost handler; the other effect is intercepted by the outermost handler thanks to the `lift` coercion. In Biernacki et al.'s λ^{HEL} [5], `lift_counter` is well-typed. The `lift` coercion is mandatory; without it, the code would be ill-typed.

[3] A `lift` coercion behaves like an end-of-scope marker for the name s. This concept has been studied, independently of effects, by various authors [13,10].

Lexically scoped handlers and dynamic allocation of effect labels. Perhaps the most straightforward way to describe the operational behavior of lexically scoped handlers is by means of their encoding in terms of ordinary effect handlers and dynamic generation of effect labels. So, let us first extend our calculus with dynamic allocation of effect labels. We introduce the construct $\texttt{effect}\ s\ \texttt{in}\ e$, which binds the effect name s to a freshly generated *effect label*, then executes e. The effect name s is a local variable: its scope is the subexpression e. An effect label is a runtime entity; later in the paper, we let ℓ range over effect labels. In this setting, a "lexically scoped handler" is encoded (simulated) as follows:

$$\begin{array}{l}\texttt{lex-handle}\ e\ \texttt{with}\ h \mid r = \\ \quad \texttt{effect}\ s\ \texttt{in}\ \texttt{handle}\ e\ (\lambda x.\, \texttt{perform}\ s\ x)\ \texttt{with}\ s : h \mid r\end{array} \tag{1}$$

This code first generates a fresh effect label, denoted by the name s. Then, it installs a handler for the name s. This handler monitors the execution of the expression e to the anonymous function $\lambda x.\, \texttt{perform}\ s\ x$, which can be viewed as a "capability" to perform the effect s.

A noteworthy aspect of the syntactic sugar $\texttt{lex-handle}\ e\ \texttt{with}\ h \mid r$ is that it does not explicitly involve any effect name. This construct is known as a "lexically scoped handler".

A lexically scoped handler can be used to write a fixed version of `bad_counter`:

$$\texttt{counter}\ \mathit{ff} \;=\; \lambda f.\left(\begin{array}{l}\texttt{lex-handle}\ \lambda \mathit{tick}.\, \mathit{ff}\ (\lambda x.\, \mathit{tick}\ ();\ f\ x)\ \texttt{with} \\ \lambda\_\ k.\, \lambda n.\, k\ ()\ (n+1) \mid \lambda y.\, \lambda n.\, (y, n)\end{array}\right) 0 \tag{2}$$

When `lex-handle` is executed, a fresh effect label (which is never explicitly mentioned in this code) is generated. The variable *tick* stands for the "capability" to perform this fresh nameless effect. One can check that the expression $\texttt{counter}\ (\texttt{counter}\ (\lambda f.\, f\ ()))\ (\lambda\_.\, ())$ reduces to the value $(((), 1), 1)$, as desired, because the two instances of `counter` generate two distinct dynamic labels and install one handler for each of these labels. Thus, no collision takes place.

Arguments in favor of dynamic allocation of effect labels. In summary, dynamic allocation of effect labels is a way of avoiding collisions between effect names. It can express lexically scoped handlers, but does not impose the use of lexically scoped handlers: it also allows working with global names when desired. Its dynamic semantics is simple. It is in use in several established programming languages, such as Standard ML and OCaml.

We believe that lexically scoped handlers are an elegant idiom, which is well suited to many but not all situations. So, we would not be satisfied with a restricted programming language where lexically scoped handlers are the sole form of effect handling. Indeed, lexically scoped handlers impose a somewhat unnatural "capability-passing" style, where the capability to perform an effect must be passed as an argument to a function (or captured in its closure). This style becomes especially cumbersome when multiple effects are involved. Implicit arguments can help, as suggested by Zhang and Myers [41] and by Odersky et al. [27]. However, elaboration of implicit arguments is usually a type-directed

$$
\begin{array}{rl}
n ::= & s \mid \ell \\
v ::= & () \mid \ell \mid \texttt{rec}\, f\, x.\, e \mid \S K \\
e ::= & v \mid x \mid e\ e \mid \texttt{ref}\ e \mid\ !e \mid e := e \\
& \mid \texttt{effect}\ s\ \texttt{in}\ e \mid \texttt{perform}\ n\ e \mid \texttt{handle}\ e\ \texttt{with}\ n : v \mid v \mid \texttt{eff}\ \ell\ v\ K \\
K ::= & \bullet \mid e\ K \mid K\ v \mid \texttt{ref}\ K \mid\ !K \mid e := K \mid K := v \\
& \mid \texttt{perform}\ \ell\ K \mid \texttt{handle}\ K\ \texttt{with}\ \ell : v \mid v
\end{array}
$$

Fig. 1. Syntax of effect values, values, expressions, and evaluation contexts

$$
\begin{array}{c}
\texttt{effect}\ s\ \texttt{in}\ e \ /\ \sigma \to e[\ell/s]\ /\ \sigma[\ell \mapsto ()] \\
\texttt{perform}\ \ell\ v\ /\ \sigma \to \texttt{eff}\ \ell\ v\ \bullet\ /\ \sigma \\
\texttt{handle}\ v\ \texttt{with}\ \ell : h \mid r\ /\ \sigma \to r\ v\ /\ \sigma \\
\texttt{handle}\ (\texttt{eff}\ \ell\ v\ K)\ \texttt{with}\ \ell : h \mid r\ /\ \sigma \to h\ v\ \S(\texttt{handle}\ K\ \texttt{with}\ \ell : h \mid r)\ /\ \sigma \\
\S K\ v\ /\ \sigma \to K[v]\ /\ \sigma \\
\\
(\texttt{eff}\ \ell\ v_1\ K)\ v_2\ /\ \sigma \to \texttt{eff}\ \ell\ v_1\ (K\ v_2)\ /\ \sigma \\
e_1\ (\texttt{eff}\ \ell\ v_2\ K)\ /\ \sigma \to \texttt{eff}\ \ell\ v_2\ (e_1\ K)\ /\ \sigma \\
\texttt{handle}\ (\texttt{eff}\ \ell\ v\ K)\ \texttt{with}\ \ell' : h \mid r\ /\ \sigma \to \texttt{eff}\ \ell\ v\ (\texttt{handle}\ K\ \texttt{with}\ \ell' : h \mid r)\ /\ \sigma
\end{array}
$$

Fig. 2. The head reduction relation (selected rules)

translation. If at all possible, we wish to preserve the "type erasure" property: that is, we prefer a language whose operational semantics is not influenced by type information, because such a semantics is easier to explain to an end user. Similarly, we wish to avoid effect coercions because we believe that they introduce unwarranted complexity, making the language and its dynamic semantics more difficult to explain to programmers.

3 Syntax and Semantics

We introduce TesLang, a calculus with mutable state, effect handlers, multiple named effects, dynamic allocation of effect labels, and multi-shot continuations. The operational semantics of this calculus allows a continuation to be invoked several times. With respect to this semantics, the type system presented in this paper (§4) is strongly sound: it rules out all runtime errors (§5). With respect to a dynamic semantics where invoking a continuation twice causes a runtime failure, such as the semantics of OCaml 5, our type system would be weakly sound, because it does not rule out this kind of runtime failure. Ensuring that every continuation is invoked at most once would require an affine type system and is beyond the scope of this paper. We note that an affine program logic, such as Hazel [35], can guarantee that no continuation is invoked twice, therefore can guarantee strong soundness even in the presence of one-shot continuations.

Our small-step operational semantics is very straightforward. It is equipped with dynamic allocation of effect labels and with a standard treatment of effects and effect handlers [2]. When an effect with label ℓ is performed, a dynamic lookup takes place: the nearest enclosing handler that is able to handle the label ℓ is selected. This is expressed, in small-step style, via several reduction rules. In contrast with some papers in the literature, where *coercions* influence the process of selecting a handler [21,4,5], here, this process is based purely on equality of effect labels.

3.1 Syntax

We let f and x range over an infinite set of variables. We let s range over an infinite set of variables, and we refer to these variables as *effect names*. These two namespaces are independent of one another: an effect name cannot be passed as a parameter to a function. We let ℓ range over an infinite set of addresses. These addresses model both *memory locations* and *effect labels*. Both kinds of entities are dynamically allocated, so, for simplicity, we use a single namespace of addresses and a single store. Whereas variables f, x and effect names s can appear in source programs, memory locations and effect labels ℓ exist only at runtime. The reduction rules of the small-step semantics cause them to appear.

The syntax of effect values, values, expressions, and evaluation contexts is shown in Figure 1.

An *effect value* n is either an effect name s or an effect label ℓ. This syntactic category is closed under substitutions of effect labels for effect names. It is used in the constructs $\texttt{perform}\, n\ e$ and $\texttt{handle}\ e\ \texttt{with}\ n : v \mid v$. A programmer always writes $\texttt{perform}\, s\ e$ and $\texttt{handle}\ e\ \texttt{with}\ s : v \mid v$, where s is an effect name, but the more general form is required in the operational semantics.

A *value* v is the unit value $()$, a memory location ℓ, a possibly recursive function $\texttt{rec}\, f\, x.\, e$, or a continuation $\S K$.

The syntax of *expressions* e includes values, variables, function application, operations for allocating, reading, and writing references, as well as constructs for allocating a fresh effect label, performing an effect, and handling an effect. Sequencing is encoded as function application: $\texttt{let}\ x = e_1\ \texttt{in}\ e_2$ is sugar for $(\lambda x.\, e_2)\ e_1$. The construct $\texttt{effect}\ s\ \texttt{in}\ e$ dynamically allocates a new effect label and binds the effect name s to this label in the expression e. The construct $\texttt{perform}\ s\ v$ performs an effect whose name is s and whose *payload* is the value v. The construct $\texttt{handle}\ e\ \texttt{with}\ s : h \mid r$ monitors the execution of the expression e. If an effect named s is performed, then the effect branch h takes control. If a value is returned, then the return branch r takes control. An effect that carries a name other than s is propagated up through this construct. Finally, the construct $\texttt{eff}\ \ell\ v\ K$, an *active effect*, does not appear in source program, but plays a role in the operational semantics, as we shall explain in the next subsection.

Our Coq formalization [36] covers a richer calculus, whose features include base types, pairs, sums, and lists.

The syntax of *evaluation contexts* K defines a right-to-left evaluation order. This choice is arbitrary: it is inspired by Iris's HeapLang language [33], but our results would hold also with left-to-right evaluation.

3.2 Semantics

The operational semantics of TesLang involves two relations, namely the *head reduction* relation $e \;/\; \sigma \rightarrow e' \;/\; \sigma'$ and the *reduction* relation $e \;/\; \sigma \longrightarrow e' \;/\; \sigma'$. They act on configurations, where a configuration $e \;/\; \sigma$ is a pair of an expression e and a *store* σ. The head reduction relation, a fragment of whose definition appears in Figure 2, is the most interesting relation. The reduction relation, whose definition is omitted, allows one step of head reduction to take place under an evaluation context.

A *store* is a finite map of addresses to values. We use addresses ℓ to denote both memory locations and effect labels. If ℓ denotes a memory location (that is, the address of a reference), then $\sigma(\ell)$ is the value stored at this address. If ℓ denotes an effect label, then the value $\sigma(\ell)$ is irrelevant: by convention, we use the unit value $()$.

The rules *not* shown in Figure 2, such as β_v-reduction and the rules for allocating, reading, and writing references, are standard.

The first rule in Figure 2 states that `effect` s `in` e allocates a fresh address ℓ, extends the store with a mapping of ℓ to the unit value, and substitutes the effect label ℓ for the effect name s in the expression e. (The rule has the side condition $\ell \notin \operatorname{dom} \sigma$.) According to the second reduction rule, `perform` $\ell\ v$ reduces to an active effect `eff` $\ell\ v\ \bullet$. An active effect has the ability to capture the surrounding evaluation context, until it reaches a handler that is able to handle it. In this rule, it is initialized with an empty evaluation context $\bullet$. The last three rules in Figure 2 show how an active effect captures its evaluation context, one frame at a time. (The last rule has the side condition $\ell \neq \ell'$.) The third and fourth rules in Figure 2 show how the return branch or the effect branch of a `handle` construct are taken. In the latter rule, the handler h is applied to the payload value value v and to a continuation, which reifies the captured evaluation context K. The continuation contains a copy of the effect handler: this is a *deep-handler semantics* [15]. The fifth reduction rule in Figure 2 describes the application of a continuation $\S K$ to a value v.

4 Type System

4.1 Syntax of types, rows, and signatures

We let α, β, and γ range over an infinite set of *type variables*. We let θ range over an infinite set of *row variables*. We distinguish three syntactic categories, namely *types*, *rows*, and *signatures* (Figure 3). The syntax of types is stable under substitutions of types τ for type variables α. The syntax of rows is stable under substitutions of rows ρ for row variables θ, for an ad hoc notion of substitution, which reduces row concatenation expressions "$\rho \cdot \rho'$" on the fly.[4]

[4] The distinction between rows and signatures enforces the view that a row ρ is a list where each component (known as a "signature") is either a signature for an effect name s or a row variable θ. Thus, we impose a simple form on rows. As an alternate

$$\begin{aligned}\tau,\,\kappa,\,\iota &::= \mathtt{unit} \mid \bot \mid \top \mid \alpha \mid \tau\ \mathtt{ref} \mid \tau \xrightarrow{\rho} \tau \mid \forall\alpha.\,\tau \mid \forall\theta.\,\tau\\ \rho &::= \langle\rangle \mid \sigma\cdot\rho\\ \sigma &::= (s : \tau \Rightarrow \tau) \mid \theta\end{aligned}$$

Fig. 3. Syntax of types, rows, and signatures

Sub

$$\frac{\Xi \mid \Delta \mid \Gamma \vdash e : \rho : \tau \qquad \rho' \vdash_b \rho \leq_R \rho' \qquad \rho' \vdash \tau \leq_T \tau'}{\Xi \mid \Delta \mid \Gamma \vdash e : \rho' : \tau'}$$

Var

$$\frac{\Gamma(x) = \tau}{\Xi \mid \Delta \mid \Gamma \vdash x : \rho : \tau}$$

RecFun

$$\frac{\Xi \mid \Delta \mid \Gamma, f : \tau \xrightarrow{\rho} \kappa, x : \tau \vdash e : \rho : \kappa}{\Xi \mid \Delta \mid \Gamma \vdash \mathtt{rec}\, f\, x.\, e : \langle\rangle : \tau \xrightarrow{\rho} \kappa}$$

App

$$\frac{\Xi \mid \Delta \mid \Gamma \vdash e : \rho : \tau \xrightarrow{\rho} \kappa \qquad \Xi \mid \Delta \mid \Gamma \vdash e' : \rho : \tau}{\Xi \mid \Delta \mid \Gamma \vdash e\, e' : \rho : \kappa}$$

TypeIntro

$$\frac{\alpha \notin \Xi, \Gamma, \rho \qquad \Xi, \alpha \mid \Delta \mid \Gamma \vdash v : \rho : \tau}{\Xi \mid \Delta \mid \Gamma \vdash v : \rho : \forall\alpha.\,\tau}$$

TypeElim

$$\frac{\Xi \mid \Delta \mid \Gamma \vdash e : \rho : \forall\alpha.\,\tau}{\Xi \mid \Delta \mid \Gamma \vdash e : \rho : \tau[\tau'/\alpha]}$$

RowIntro

$$\frac{\theta \notin \Xi, \Gamma, \rho \qquad \Xi, \theta \mid \Delta \mid \Gamma \vdash v : \rho : \tau}{\Xi \mid \Delta \mid \Gamma \vdash v : \rho : \forall\theta.\,\tau}$$

RowElim

$$\frac{\Xi \mid \Delta \mid \Gamma \vdash e : \rho : \forall\theta.\,\tau}{\Xi \mid \Delta \mid \Gamma \vdash e : \rho : \tau[\rho'/\theta]}$$

Effect

$$\frac{s \notin \Gamma, \rho, \tau \qquad \Xi \mid \Delta, s \mid \Gamma \vdash e : (s : \mathtt{abs})\cdot\rho : \tau}{\Xi \mid \Delta \mid \Gamma \vdash \mathtt{effect}\ s\ \mathtt{in}\ e : \rho : \tau}$$

Perform

$$\frac{s \in \Delta \qquad (s : \iota \Rightarrow \kappa) \in \rho \qquad \Xi \mid \Delta \mid \Gamma \vdash e : \rho : \iota}{\Xi \mid \Delta \mid \Gamma \vdash \mathtt{perform}\ s\ e : \rho : \kappa}$$

Handle

$$\frac{\begin{array}{c} s \in \Delta \qquad \Xi \mid \Delta \mid \Gamma \vdash e : \rho : \tau \\ \rho = (s : \iota \Rightarrow \kappa)\cdot\rho_0 \qquad \rho' = (s : \iota' \Rightarrow \kappa')\cdot\rho_0 \\ \Xi \mid \Delta \mid \Gamma \vdash h : \rho' : \iota \to (\kappa \xrightarrow{\rho'} \tau') \xrightarrow{\rho'} \tau' \qquad \Xi \mid \Delta \mid \Gamma \vdash r : \rho' : \tau \xrightarrow{\rho'} \tau' \end{array}}{\Xi \mid \Delta \mid \Gamma \vdash \mathtt{handle}\ e\ \mathtt{with}\ s : h \mid r : \rho' : \tau'}$$

Fig. 4. The type system (selected rules)

Our types are standard: they include the unit type `unit`, the bottom and top types $\bot$ and $\top$, type variables α, reference types, effect-annotated arrow types, value-polymorphic types, and effect-polymorphic types. Effect-annotated arrow types and effect-polymorphic types are discussed below.

A row is a list of *signatures* σ. A signature, in turn, is either a *singleton signature* $s : \iota' \Rightarrow \kappa'$ or a *row variable* θ. A singleton signature $s : \iota' \Rightarrow \kappa'$ means that performing the effect s is permitted and is analogous to calling a function of argument type ι' and return type κ'. According to this reading, a singleton signature of the form $s : \bot \Rightarrow \top$ actually forbids the effect s, because a function whose argument type is $\bot$ can never be called. We write $s : \texttt{abs}$ as a short-hand for this signature, and we refer to it as an *absence signature* for the effect s.

In addition to an argument type τ and a return type κ, an arrow type $\tau \xrightarrow{\rho} \kappa$ carries an "effect", that is, a row ρ. Intuitively, a value of type $\tau \xrightarrow{\rho} \kappa$ is a function, which, when applied to an argument of type τ, either returns a result of type κ or performs an effect that is permitted by the row ρ. On top of this standard reading of effect annotations, Tes introduces a novel aspect. The effect annotation ρ is interpreted not only as a set of permitted effects, but also as a precondition: we impose the semantic requirement that *a function of type $\tau \xrightarrow{\rho} \kappa$ can be invoked only if the multiset of effect labels denoted by the row ρ has no duplicate elements.* This is not a syntactic requirement, which would be either "true" or "false" and would be decided just by inspecting the syntax of the row ρ. Indeed, in general, a row contains occurrences of effect names s, which denote a-priori-unknown effect labels, and of row variables θ, which denote a-priori-unknown multisets of effect labels. What we wish to require is that, *at runtime, after effect names and row variables have been substituted away* by some substitution η, a function of type $\tau \xrightarrow{\rho} \kappa$ can be invoked only if no effect label appears twice in the closed row $\eta(\rho)$. Thus, the requirement that "ρ contains no duplicate labels" should be thought of as a *disjointness hypothesis* bearing on the row ρ. Such a hypothesis may or may not be satisfied, depending on how the effect names and row variables that occur in ρ are instantiated.

In Tes, disjointness hypotheses are sometimes explicit and most of the time implicit. In the subsumption judgments (Figure 5), a disjointness context D is explicit: it can be interpreted as a conjunction of disjointness hypotheses. In function types $\tau \xrightarrow{\rho} \kappa$ and in typing judgments $\Xi \mid \Delta \mid \Gamma \vdash e : \rho : \tau$, an *implicit disjointness hypothesis* bearing on the row ρ is built in, so there is no need for an explicit disjointness context.

An effect-polymorphic type $\forall\theta.\ \tau$ involves a universal quantification over a row variable θ. For instance, the function `iter`, which iterates over a list, can be defined as follows:

$$\texttt{iter} = \texttt{rec}\ \mathit{iter}\ \mathit{xs}\ f.\ \texttt{match}\ \mathit{xs}\ \texttt{with}\ (\lambda x\, \mathit{xs}.\ f\ x;\ \mathit{iter}\ \mathit{xs}\ f \mid \lambda\_.\ ()) \qquad (3)$$

path, one could use a single syntactic category $\rho ::= \langle\rangle \mid \rho \cdot \rho \mid (s : \tau \Rightarrow \tau) \mid \theta$, where a more general form of row concatenation is allowed. This would allow using a standard notion of substitution, and would lead to different statements for some of the row subsumption rules.

This function admits the following value- and effect-polymorphic type:

$$\texttt{iter} : \forall\alpha.\ \forall\theta.\ \alpha\ \texttt{list} \to (\alpha \xrightarrow{\theta} \texttt{unit}) \xrightarrow{\theta} \texttt{unit}$$

This type states that the call `iter` xs f is safe, regardless of what the elements of the list xs might be, and regardless of what effects the user function f might perform. This type also guarantees that `iter` does not perform any effect of its own: instantiating θ with $\langle\rangle$ shows that this must be the case. Finally, one might think that this type guarantees that `iter` cannot intercept the effects performed by f. This may or may not be true, depending on which interpretation of effect-polymorphic types is chosen. A stronger interpretation can guarantee this property, but rules out certain useful programming language constructs, such as "dynamic-wind". Conversely, a weaker interpretation of effect-polymorphic types allows type-checking "dynamic-wind", but breaks this guarantee. At this time, the interpretation that we have verified in Coq is the weaker one (§5). We further discuss this point in Section 6.

4.2 The typing judgment

A typing judgment in TES takes the form $\Xi \mid \Delta \mid \Gamma \vdash e : \rho : \tau$. It involves three environments: a *row- and type-variable context* Ξ, which binds row and type variables θ and α; an *effect-name context* Δ, which binds effect names s; and a *type environment* Γ, which maps variables x to types τ. This typing judgment states that the expression e has effect ρ and type τ. Like an arrow type, this judgment involves an implicit disjointness hypothesis bearing on the row ρ. That is, this judgment guarantees that it is safe to execute e *provided* the row variables and type variables in Ξ are instantiated in such a way that the multiset of effect labels denoted by ρ has no duplicate elements.

A selection of the typing rules appears in Figure 4. The typing rules for variables, functions, and applications are the same as in most type-and-effect systems. The typing rules for references are also standard, and are omitted. The rules TYPEINTRO, TYPEELIM, ROWINTRO, ROWELIM, which introduce and eliminate value- and effect-polymorphic types, are also standard. In the presence of mutable state, an unrestricted introduction rule for polymorphic types is unsound [34]. In this paper, we avoid this problem simply by building the value restriction [39,12] into TYPEINTRO and ROWINTRO. Our Coq formalization [36] proposes a more elaborate approach, where function types and typing judgments are annotated with *purity attributes*. This approach yields a slightly more expressive system, where, in particular, `perform` s x is considered a pure expression, therefore can receive a polymorphic type.

Rule EFFECT, read from bottom to top, changes the current effect from ρ to $(s : \texttt{abs}) \cdot \rho$. Intuitively, this means several things. First, while type-checking e, it is safe to *assume* that the effect label denoted by s is disjoint from the multiset of effect labels denoted by ρ. This assumption is implicitly expressed by the mere appearance of the row $(s : \texttt{abs}) \cdot \rho$ in the premise. This assumption is justified indeed, since the effect name s is bound to a *fresh* effect label when `effect` s `in` e

is executed. Second, because of the absence signature $s : \texttt{abs}$, one must *check* that the expression e does not perform any effect with the name s. This seems a natural and unavoidable restriction: if such an effect was allowed, there would be no static effect name by which it can be described. Third, because of the side condition $s \notin \rho$, one must *check* that the row that appears in the premise contains *at most one singleton signature* for the effect name s. As a counterexample, if the expression e has effect $(s : \texttt{abs}) \cdot (s : \texttt{abs})$, then the typing rule Effect cannot be applied. The subsumption rule Sub cannot help, because the subsumption judgment $(s : \texttt{abs}) \cdot (s : \texttt{abs}) \leq (s : \texttt{abs})$ does not hold. Thus, the rule Effect enforces a disjointness constraint.

Rule Perform states that, when one performs an effect whose signature is $s : \iota \Rightarrow \kappa$, one must pass a payload value of type ι, and, in return, one can expect a value of type κ. This supports the intuitive idea that performing an effect is analogous to calling an effect-free function of type $\iota \rightarrow \kappa$.

Rule Handle type-checks `handle` e `with` $s : h \mid r$, where the expression e is monitored by a handler for the effect s. This rule expresses the idea that this construct establishes a boundary between the inside, where effects named s may be performed in accord with the signature $s : \iota \Rightarrow \kappa$, and the outside, where effects named s may be performed in accord with a different signature $s : \iota' \Rightarrow \kappa'$. Because $s : \texttt{abs}$ is sugar for $s : \bot \Rightarrow \top$, this rule also covers the common case where the effect s is absent on the outside. Both the effect branch h and the return branch r are part of the "outside world", so their effects are described by the outside row ρ'. This remark explains all occurrences of ρ' in the last two premises, except the one in the type of the continuation. The continuation, which is the second parameter of the effect branch h, has type $\kappa \xrightarrow{\rho'} \tau'$. Because we have adopted a "deep-handler" semantics (§3), a copy of the handler is reinstalled inside the continuation. This explains why the effect ρ' and the result type τ' of the continuation are the same as those of the whole `handle` construct.

Rule Sub weakens a typing judgment by replacing an effect ρ and a type τ with a weaker effect ρ' and a weaker type τ'. This rule relies on several subsumption judgments, which we discuss next.

4.3 The subsumption judgments

The subsumption judgments on types, signatures, and rows appear in Figure 5. An original aspect is that these judgments depend on a *disjointness context* D, which appears on the left of the turnstile. A disjointness context is a (possibly empty, unordered) list of rows, and is interpreted as a conjunction of disjointness hypotheses: one hypothesis bears on each row. For instance, the disjointness context $(s_1 : \iota_1 \Rightarrow \kappa_1) \cdot (s_2 : \iota_2 \Rightarrow \kappa_2)$, $(s_3 : \iota_3 \Rightarrow \kappa_3) \cdot \theta$, which is a list of two rows, is equivalent to a conjunction of two disjointness hypotheses. The first hypothesis is equivalent to $s_1 \neq s_2$: it represents the assumption that the effect names s_1 and s_2 denote two distinct effect labels. The second hypothesis expresses the assumption that the effect label denoted by s_3 is not a member of the multiset of effect labels denoted by θ *and* that this multiset has no duplicate elements.

Type subsumption

$$\textsc{TypeRefl}\quad D \vdash \tau \leq_T \tau \qquad \textsc{Bot}\quad D \vdash \bot \leq_T \tau \qquad \textsc{Top}\quad D \vdash \tau \leq_T \top \qquad \textsc{TypeTrans}\quad \frac{D \vdash \tau \leq_T \tau' \qquad D \vdash \tau' \leq_T \tau''}{D \vdash \tau \leq_T \tau''}$$

$$\textsc{Arrow}\quad \frac{D, \rho' \vdash \tau' \leq_T \tau \qquad D, \rho' \vdash_b \rho \leq_R \rho' \qquad D, \rho' \vdash \kappa \leq_T \kappa'}{D \vdash \tau \xrightarrow{\rho} \kappa \leq_T \tau' \xrightarrow{\rho'} \kappa'}$$

Signature subsumption

$$\textsc{SigRefl}\quad D \vdash \sigma \leq_S \sigma \qquad \textsc{SigCons}\quad \frac{D \vdash \iota \leq_T \iota' \qquad D \vdash \kappa' \leq_T \kappa}{D \vdash (s : \iota \Rightarrow \kappa) \leq_S (s : \iota' \Rightarrow \kappa')}$$

Row subsumption

$$\textsc{Empty}\quad D \vdash_b \langle\rangle \leq_R \langle\rangle \qquad \textsc{Extend}\quad D \vdash_b \rho \leq_R \sigma \cdot \rho \qquad \textsc{Swap}\quad D \vdash_b \sigma \cdot \sigma' \cdot \rho \leq_R \sigma' \cdot \sigma \cdot \rho$$

$$\textsc{RowCons}\quad \frac{\begin{array}{c} D \vdash \sigma \leq_S \sigma' \\ D \vdash_{\mathit{false}} \rho \leq_R \rho' \end{array}}{D \vdash_b \sigma \cdot \rho \leq_R \sigma' \cdot \rho'} \qquad \textsc{Erase}\quad \frac{D \Vdash s \mathbin{\#} \rho}{D \vdash_{\mathit{true}} (s : \mathsf{abs}) \cdot \rho \leq_R \rho} \qquad \textsc{RowTrans}\quad \frac{\begin{array}{c} D \vdash_b \rho \leq_R \rho' \\ D \vdash_b \rho' \leq_R \rho'' \end{array}}{D \vdash_b \rho \leq_R \rho''}$$

Effect/row disjointness

$$D \Vdash s \mathbin{\#} \langle\rangle \qquad \frac{D \Vdash s \mathbin{\#} \sigma \qquad D \Vdash s \mathbin{\#} \rho}{D \Vdash s \mathbin{\#} (\sigma \cdot \rho)}$$

$$\frac{\rho \in D \qquad \{(s : \cdot \Rightarrow \cdot),\ (s' : \cdot \Rightarrow \cdot)\} \subseteq_m \rho}{D \Vdash s \mathbin{\#} (s' : \iota' \Rightarrow \kappa')} \qquad \frac{\rho \in D \qquad \{(s : \cdot \Rightarrow \cdot),\ \theta\} \subseteq_m \rho}{D \Vdash s \mathbin{\#} \theta}$$

Fig. 5. The subsumption judgments

In the subsumption rules, the disjointness context is extended in the rule Arrow and exploited in the rule Erase. Elsewhere, it is just transported.

Subsumption on types. The subsumption judgment on types $D \vdash \tau \leq_T \tau'$ means that, under the hypothesis D, τ is a subtype of τ'. The rules in Figure 5 state that this relation is reflexive, transitive, and admits $\bot$ and $\top$ as bottom and top elements. On function types, as usual, subsumption is contravariant in the domain and covariant in the effect and in the codomain. One original aspect of Arrow is that this rule enriches the disjointness context: in the premises, the disjointness context changes from D to D, ρ'. The intuitive reason why this is sound is that if someone uses a function at type $\tau' \xrightarrow{\rho'} \kappa'$ then (at the point where the function is used) the disjointness hypothesis ρ' must be satisfied, because this hypothesis is part of our interpretation of function types. Thus, when proving that a function of type $\tau \xrightarrow{\rho} \kappa$ can be used as a function of type $\tau' \xrightarrow{\rho'} \kappa'$, it is safe to rely on the disjointness hypothesis ρ'.

Subsumption on signatures. The subsumption judgment on signatures takes the form $D \vdash \sigma \leq_S \sigma'$. Signature subsumption is reflexive and transitive. (Reflexivity is given by SigRefl; transitivity is derivable.) According to SigCons, unlike the standard function type constructor $\cdot \rightarrow \cdot$, the signature constructor $s : \cdot \Rightarrow \cdot$ is covariant in its domain and contravariant in its codomain. Indeed, when the signature $s : \iota \Rightarrow \kappa$ appears in the effect of an expression e, this means that e has *permission* to perform an effect named s at type $\iota \Rightarrow \kappa$. In other words, e can *assume* that performing an effect named s is analogous to calling a function of type $\iota \rightarrow \kappa$. This explains the reversed variance.

Subsumption on rows. The row subsumption judgment is $D \vdash_b \rho \leq_R \rho'$. The Boolean parameter b will be explained shortly. Row subsumption is reflexive and transitive. (Reflexivity is derivable; transitivity is given by RowTrans.) By combining Empty, Extend, RowCons, Swap, and RowTrans, one finds that if two rows, viewed as multisets of effect signatures, are related by multiset inclusion, then they are related by subsumption. Thus, subsumption allows permuting row entries in arbitrary ways and extending a row with new entries.

The last row subsumption rule, Erase, allows dropping an effect signature of the form $s : \texttt{abs}$. This rule may seem plausible because, both in the presence of the effect signature $s : \texttt{abs}$ and its absence, the effect s is forbidden. However, an unqualified axiom $\vdash (s : \texttt{abs}) \cdot \rho \leq_R \rho$ would be unsound. This is due to our interpretation of the row carried by a typing judgment (or by a function type) as a disjointness hypothesis. By changing a typing judgment that carries the row $(s : \texttt{abs}) \cdot \rho$ into one that carries the row ρ, *one removes the hypothesis* that the effect label denoted by s is not a member of the multiset of effect labels denoted by ρ. In order to safely remove a hypothesis, one must prove that it is satisfied. This explains why Erase must carry the premise $D \Vdash s \# \rho$, whose intuitive meaning is that "the hypotheses in D guarantee that the effect label denoted by s is not among the effect labels denoted by ρ".

The parameter b serves to forbid a use of Erase under RowCons. Erase requires this flag to be *true*, but RowCons sets it to *false* in its premise. Without this restriction, one could first combine Erase and DisjEmpty to prove $\vdash (s : \texttt{abs}) \cdot \langle\rangle \leq_R \langle\rangle$, then use RowCons and induction to obtain $\vdash (s : \texttt{abs}) \cdot \rho \leq_R \rho$ *without any side condition*, thus circumventing the side condition in Erase.

The four rules that define the effect/row disjointness judgment $D \Vdash s \# \rho$ are straightforward. The first two rules decompose the row ρ, which is a list of effect signatures σ. The last two rules look up the disjointness context D so as to find a disjointness hypothesis ρ that implies the goal. Whether ρ implies the goal is decided based on a simple syntactic criterion: the relation $\cdot \subseteq_m \cdot$ denotes multiset inclusion; the row on the right-hand side is viewed as a multiset of effect signatures.[5]

The desire to support Erase is the reason why the subsumption judgments carry a disjointness context. In a hypothetical simplified system where these judgments do not carry such a context, the premise of Erase would have to use an empty disjointness context *True*. This premise would become $\mathit{True} \Vdash s \# \rho$, which is false, so Erase would become inapplicable. Yet Erase is desirable, because it is useful in practice. We use it to type-check our encoding of a lexically scoped handler: this is illustrated in Section 4.4.

Why is $\vdash (s : \texttt{abs}) \cdot \rho \leq_R \rho$ unsound? In the presence of this axiom, the judgment $\vdash (s : \texttt{abs}) \cdot (s : \texttt{abs}) \leq_R (s : \texttt{abs})$ would be derivable. This judgment can be exploited to type-check the following unsafe program:

```
1 effect s in
2 handle
3   handle (perform s ()) with s : λx _. not x | λ_. true
4 with s : λ_ _. () | λ_. ()
```

This program is unsafe because the effect s is performed with a payload of type `unit`, namely the unit value $()$ on line 3, and this effect is handled by the innermost handler, also on line 3, which expects the payload x to be a Boolean value. When this program is executed, it becomes stuck by attempting to execute the function application `not` $()$.

Yet, under the assumption $\vdash (s : \texttt{abs}) \cdot (s : \texttt{abs}) \leq_R (s : \texttt{abs})$, this program is well-typed, with an empty row and with the type `unit`. Beginning at the root and working towards the leaves, the type derivation begins with an application of Effect, which changes the empty row into the row $(s : \texttt{abs})$. Then, by using Sub and by exploiting the above assumption, the row $(s : \texttt{abs})$ can be changed to $(s : \texttt{abs}) \cdot (s : \texttt{abs})$. At this point, the harm is done. Indeed, under the row $(s : \texttt{abs}) \cdot (s : \texttt{abs})$, the subprogram at lines 2–4 is well-typed. The fact that this row includes two signatures for the effect name s allows us to install two handlers for this name. The handler on line 2 allows its handlee—the expression

[5] Our Coq code [36] presently employs a different representation of disjointness contexts and a different definition of the effect/row disjointness judgment. We believe, but have not yet checked, that the Coq and paper formulations are equivalent.

on line 3—to perform effects according to the signature $s : \texttt{unit} \Rightarrow \texttt{unit}$. The handler on line 3 allows its handlee to perform effects as per $s : \texttt{bool} \Rightarrow \texttt{unit}$. The expression `perform` s () is type-checked with respect to the composite row $(s : \texttt{unit} \Rightarrow \texttt{unit}) \cdot (s : \texttt{bool} \Rightarrow \texttt{unit})$, which means that this expression must respect *either* of these two signatures. It does indeed respect the first one, so it is well-typed.

4.4 Examples

Filter Recall the higher-order iteration function `iter` (Eq. 3), whose type is

$$\texttt{iter} : \forall\alpha.\ \forall\theta.\ \alpha\ \texttt{list} \rightarrow (\alpha \xrightarrow{\theta} \texttt{unit}) \xrightarrow{\theta} \texttt{unit}.$$

Let us use `iter` in the definition of `filter`:

$$\texttt{filter}\ xs\ f = \texttt{let}\ g = (\lambda x.\ \texttt{if}\ f\ x\ \texttt{then}\ \texttt{perform}\ \mathit{yield}\ x)\ \texttt{in}\ \texttt{iter}\ xs\ g$$

The expression `filter` xs f “yields” each element x of the list xs in turn, by performing a *yield* effect if $f\ x$ returns `true`. In Tes, `filter` is well-typed, and its type is:

$$\texttt{filter} : \forall\alpha.\ \forall\theta.\ \alpha\ \texttt{list} \rightarrow (\alpha \xrightarrow{\theta} \texttt{bool}) \xrightarrow{(\mathit{yield}\,:\,\alpha \Rightarrow \texttt{unit})\cdot\theta} \texttt{unit}$$

Checking that `filter` is well-typed is not difficult. Under the assumption that f has type $\alpha \xrightarrow{\theta} \texttt{bool}$, the subexpression $f\ x$ has effect θ. Under the assumption that x has type α, the subexpression `perform` *yield* x has effect $(\mathit{yield} : \alpha \Rightarrow \texttt{unit})$. Because our subsumption rules allow extending a row with a new entry and exchanging row entries, the composite subexpression `if` $f\ x$ `then` `perform` *yield* x admits the composite effect $(\mathit{yield} : \alpha \Rightarrow \texttt{unit}) \cdot \theta$.

What does `filter`'s type mean? Ostensibly, the row $(\mathit{yield} : \alpha \Rightarrow \texttt{unit}) \cdot \theta$ tells us that every effect performed by `filter` xs f must be either a *yield* effect or an effect caused by f. Less obviously, these alternatives must be mutually exclusive: indeed, the row $(\mathit{yield} : \alpha \Rightarrow \texttt{unit}) \cdot \theta$ carries the implicit requirement that the effect label denoted by *yield* is not among the effect labels denoted by θ. In other words, `filter`*'s type forbids f from performing yield effects.*

The reader may wonder what prevents us from instantiating θ with a row that includes the effect name *yield*, such as $(\mathit{yield} : \alpha \Rightarrow \texttt{unit})$. The answer is, nothing prevents such an instantiation. The result, however, would be a view of `filter` as a function whose effect is $(\mathit{yield} : \alpha \Rightarrow \texttt{unit}) \cdot (\mathit{yield} : \alpha \Rightarrow \texttt{unit})$. Such an effect carries an unsatisfiable disjointness hypothesis, namely $\mathit{yield} \neq \mathit{yield}$. As a result, once the type of `filter` has been instantiated in this way, `filter` cannot be called anymore.[6]

[6] Technically, an application of this instantiated `filter` function can still be well-typed, but only if it appears in the body of a function which itself carries an unsatisfiable disjointness hypothesis and therefore can never be called.

Lexically scoped handlers We now derive a typing rule for lexically scoped handlers. Recall the encoding of a lexically scoped handler (Eq. 1):[7]

$$\begin{array}{l}\texttt{lex-handle}_s\ e\ \texttt{with}\ h \mid r = \\ \quad \texttt{effect}\ s\ \texttt{in handle}\ e\ (\lambda x.\ \texttt{perform}\ s\ x)\ \texttt{with}\ s : h \mid r\end{array}$$

For this construct, Tes admits the following derived typing rule:

LexHandle

$$\frac{\begin{array}{c}\Xi \mid \Delta \mid \Gamma \vdash e : \rho : \forall\theta.\ (\iota \xrightarrow{\theta} \kappa) \xrightarrow{\theta\cdot\rho} \tau \qquad s \notin \Gamma, \rho, \iota, \kappa, \tau, \tau' \\ \Xi \mid \Delta \mid \Gamma \vdash h : \rho : \iota \to (\kappa \xrightarrow{\rho} \tau') \xrightarrow{\rho} \tau' \qquad \Xi \mid \Delta \mid \Gamma \vdash r : \rho : \tau \xrightarrow{\rho} \tau'\end{array}}{\Xi \mid \Delta \mid \Gamma \vdash \texttt{lex-handle}_s\ e\ \texttt{with}\ h \mid r : \rho : \tau'}$$

This rule is similar to the typing rule for lexically scoped handlers that appears in Figure 3 of Biernacki et al.'s paper [6]. What is new and noteworthy is that we obtain this rule as a special case of a more permissive type discipline, Tes, which supports general effect handlers, as opposed to just lexically scoped handlers.

In LexHandle, whereas the effect on the outside is ρ, the effect on the inside is $\theta \cdot \rho$. That is, inside the handlee, one more effect is permitted. The handlee (the expression e) must be polymorphic in the row variable θ: that is, it must treat this extra effect as an abstract effect.

The derivation of LexHandle involves an application of Effect and an application of Handle. While proving that the premises of Handle hold, a key step is to prove that the type of the effect branch h can be weakened as follows, where ρ' is a shorthand for $(s : \texttt{abs}) \cdot \rho$:

$$\frac{\Xi \mid \Delta \mid \Gamma \vdash h : \rho : \iota \to (\kappa \xrightarrow{\rho} \tau') \xrightarrow{\rho} \tau'}{\Xi \mid \Delta \mid \Gamma \vdash h : \rho : \iota \to (\kappa \xrightarrow{\rho'} \tau') \xrightarrow{\rho'} \tau'} \qquad \rho' = (s : \texttt{abs}) \cdot \rho$$

It is not at all obvious that this is possible! Two occurrences of ρ must be changed into ρ'. One occurrence is positive and one is negative, and the rows ρ and ρ' are not equal. Still, this implication can be established, via rule Sub. One must check the following chain of subsumption relations:

$$\iota \to (\kappa \xrightarrow{\rho} \tau') \xrightarrow{\rho} \tau' \leq_T \iota \to (\kappa \xrightarrow{\rho} \tau') \xrightarrow{\rho'} \tau' \leq_T \iota \to (\kappa \xrightarrow{\rho'} \tau') \xrightarrow{\rho'} \tau'$$

The first step requires $\vdash_b \rho \leq_R \rho'$, which, by Extend, is true. The second step requires $\rho' \vdash_{true} \rho' \leq_R \rho$, which, by Erase, is true as well. The disjointness hypothesis ρ' plays a key role: indeed, $\mathit{True} \vdash_{true} \rho' \leq_R \rho$ is false. In other words, Erase is applicable because the disjointness hypothesis ρ' is available, and this hypothesis exists because Arrow causes it to appear as it descends into the domains of two function types that are annotated with ρ'.

[7] This encoding requires choosing an arbitrary name s that does not occur in e, h or r. Furthermore, in the derivation of the typing rule LexHandle, s may need to be renamed. On paper, we would normally not mention these details. However, because our Coq code does not currently allow α-conversion of effect names, we make s a parameter of the macro `lex-handle` and we include a freshness hypothesis bearing on s in LexHandle.

Counter Using the type rule LexHandle, it is straightforward to check that `counter` (§2, Eq. 2) can be assigned the following type:

$$\texttt{counter} : \forall \alpha\, \beta\, \gamma.\quad (\forall \theta.\, (\alpha \xrightarrow{\theta} \beta) \xrightarrow{\theta} \gamma) \quad \to \quad \forall \theta.\, (\alpha \xrightarrow{\theta} \beta) \xrightarrow{\theta} (\gamma * \texttt{int})$$

This means that `counter` accepts an arbitrary effect-polymorphic second-order function *ff* and produces a function *ff*′ whose type is similar to *ff*'s type. The only difference between the types of *ff* and *ff*′ is in their result types, to wit, γ versus $\gamma * \texttt{int}$.

It is not hard to see that the expression $\texttt{counter}\ (\texttt{counter}\ (\lambda f.\, f\ ()))\ (\lambda\_.\, ())$, where two instances of `counter` are nested, is also well-typed, and that its type is $(\texttt{unit} * \texttt{int}) * \texttt{int}$.

Mix The following second-order function, `mix`, involves a potentially challenging mixture of features:

$$\begin{aligned}&\texttt{mix}\ f\ =\\ &\quad \texttt{handle}\ (\texttt{perform}\ s\ ()\texttt{;}\ f\ ())\\ &\quad \texttt{with}\ s\ \texttt{:}\ \lambda\_\ k.\, k\ ()\ |\ \lambda\_.\, ()\end{aligned}$$

The effect name s occurs free in this code, so this is *not* an instance of a lexically scoped handler. (We assume that the name s is introduced by the surrounding context.) The subexpression $\texttt{perform}\ s\ ()\texttt{;}\ f\ ()$ visibly performs the effect s and calls the unknown function f, which itself may perform various effects, perhaps including the effect s. This subexpression is monitored by a handler for the effect s at type $\texttt{unit} \Rightarrow \texttt{unit}$.

In Tes, `mix` is well-typed. In fact, it admits several types. We show three: the first two are equivalent, and the last one subsumes the first two.

The first idea that comes to mind may be: "since f has an unknown effect, let's represent this effect with a row variable θ". Thus, one introduces a row variable θ, and one assumes that f has type $\texttt{unit} \xrightarrow{\theta} \texttt{unit}$. Under this assumption, one finds that $\texttt{perform}\ s\ ()\texttt{;}\ f\ ()$ has effect $(s : \texttt{unit} \Rightarrow \texttt{unit}) \cdot \theta$. (The subsumption rule Extend is used, twice, to merge the effect of $\texttt{perform}\ s\ ()$ and the effect of $f\ ()$.) Finally, using Handle, one finds that the body of the function `mix` has effect $(s : \texttt{abs}) \cdot \theta$. In summary, `mix` admits the following type:

$$\texttt{mix} : \forall \theta.\, (\texttt{unit} \xrightarrow{\theta} \texttt{unit}) \xrightarrow{(s\,:\,\texttt{abs})\cdot\theta} \texttt{unit} \tag{4}$$

The effect $(s : \texttt{abs}) \cdot \theta$ carried by the second arrow means that `mix` never throws the effect s and transmits whatever effects f may throw, *provided these effects do not include* s. Indeed, the row $(s : \texttt{abs}) \cdot \theta$ is interpreted not only as a description of `mix`'s potential effects, but also as a disjointness constraint. Thus, the row $(s : \texttt{abs}) \cdot \theta$ in this type (4) cannot be replaced with just θ. Such a replacement would amount to discarding the disjointness constraint, which would be unsound.

The reader may wonder what happens if θ is instantiated, in the above type, with a row that mentions s, such as $s : \texttt{int} \Rightarrow \texttt{int}$. Technically, this is permitted, but yields a version of `mix` whose effect is $(s : \texttt{abs}) \cdot (s : \texttt{int} \Rightarrow \texttt{int})$. Such a function can never be called.

Thus, this type (4) effectively forbids f from performing effect s. One may wonder whether this fact can be made explicitly visible in the type of `mix`. In fact, it can. By the subsumption rules Arrow, Extend, and Erase, the type (4) is equivalent to the following type:

$$\texttt{mix} : \forall\theta.\ (\texttt{unit} \xrightarrow{(s\,:\,\texttt{abs})\cdot\theta} \texttt{unit}) \xrightarrow{(s\,:\,\texttt{abs})\cdot\theta} \texttt{unit} \qquad (5)$$

Indeed, under the disjointness constraint carried by the outer arrow, the rows θ and $(s : \texttt{abs}) \cdot \theta$ are equivalent.

It is worth noting that this type allows the function f to use the effect s internally, if desired, and at an arbitrary type, provided this effect is handled internally by f and does not escape.

Finally, one may wonder whether it is necessary to forbid f from visibly performing effect s. In fact, it is not: one can allow f to perform this effect and let it escape, provided it is performed at type `unit` $\Rightarrow$ `unit`, which is the type expected by the handler inside `mix`. It is not difficult to check that `mix` admits the following type:

$$\texttt{mix} : \forall\theta.\ (\texttt{unit} \xrightarrow{(s\,:\,\texttt{unit}\Rightarrow\texttt{unit})\cdot\theta} \texttt{unit}) \xrightarrow{(s\,:\,\texttt{abs})\cdot\theta} \texttt{unit} \qquad (6)$$

This type (6) is in fact more general than (that is, a subtype of) the previous type (5). This follows directly from the fact that $s : \texttt{abs}$ is a short-hand for $s : \bot \Rightarrow \top$ and from the subsumption rules SigCons, RowCons, and Arrow.

5 Metatheory

In this section, we present the general architecture of the proof of our type soundness statement (Theorem 3), which states that, if a closed program e is well-typed, then e is *safe*: that is, e may diverge or terminate with a value, but cannot perform an unhandled effect. Full details are found in our Coq code [36].

Our first step is to interpret our typing judgments as *semantic typing judgments*. A semantic typing judgment $\Xi \mid \Delta \mid \Gamma \vDash e : \rho : \tau$ is a logical assertion stating that substituting *certain values* for the free variables of e yields a closed program that meets a *certain specification*. To fill in the details, one must define precisely which values may be substituted and what specification is met.

To do so, we introduce TesLogic, an extension of Iris [16], an expressive Separation Logic. Iris's base logic has no built-in support for effects and handlers, but allows constructing a program logic with such support. de Vilhena and Pottier define such a logic, Hazel [35]. Because Hazel is tailored for unnamed effects and one-shot continuations, we cannot re-use it. Nevertheless, in the design of TesLogic, we do rely on one of Hazel's key features, *protocols*.

A protocol Ψ describes a service on which the handlee can rely and which the handler must implement. Mathematically, it is a binary relation between a value v, the payload of the effect, and a predicate Φ, the precondition of the continuation for this effect. A typical example of a protocol is the *pre/post protocol*

Weakest precondition

$$wp\ e\ \langle E\rangle\{\Phi\} \;\triangleq\; \mathit{ValidDistinct}\ E.1 \mathrel{-\!\!*} ewp\ e\ \langle E\rangle\{\Phi\}$$

Basic weakest precondition

$$\begin{aligned}
ewp\ v\ \langle E\rangle\{\Phi\} &\triangleq \Phi(v)\\
ewp\ (\texttt{eff}\ \ell\ v\ K)\ \langle E\rangle\{\Phi\} &\triangleq \exists \Psi.(\ell,\Psi) \in E \ * \ (\uparrow_{\square}\Psi)\ v\ (\lambda w.\, \triangleright\, ewp\ K[w]\ \langle E\rangle\{\Phi\})\\
ewp\ e\ \langle E\rangle\{\Phi\} &\triangleq \forall \sigma.\, S(\sigma)\ {}^{\top}{\Rrightarrow}^{\emptyset}\\
&\qquad \begin{cases} \exists e', \sigma'.\ e \mathbin{/} \sigma \longrightarrow e' \mathbin{/} \sigma' \ *\\ \forall e', \sigma'.\ e \mathbin{/} \sigma \longrightarrow e' \mathbin{/} \sigma'\ {}^{\emptyset}{\Rrightarrow}^{\emptyset}\, \triangleright\, {}^{\emptyset}{\Rrightarrow}^{\top}\\ \qquad\quad S(\sigma') \ * \ ewp\ e'\ \langle E\rangle\{\Phi\} \end{cases}
\end{aligned}$$

Persistent upward closure

$$(\uparrow_{\square}\Psi)\ v\ \Phi \;\triangleq\; \exists \Phi'.\ \Psi\ v\ \Phi' \ * \ \square\forall w.\, \Phi'(w) \mathrel{-\!\!*} \Phi(w)$$

Validity-and-distinctness property

$$\mathit{ValidDistinct}\ L \;\triangleq\; \mathit{NoDup}\ L \wedge \bigwedge_{\ell\in L} \ell \mapsto_{\square} ()$$

Fig. 6. Definition of the weakest precondition

$\{\Phi_1\}.\{\Phi_2\}$, defined as $\lambda\, v\, \Phi.\ \Phi_1(v) \ * \ \square\forall\, w.\ \Phi_2(w) \mathrel{-\!\!*} \Phi(w)$. We use this protocol (in the interpretation of signatures, Figure 7) to attach a precondition Φ_1 and a postcondition Φ_2 to an effect: performing an effect with payload v is permitted if $\Phi_1(v)$ holds, and one can assume that it returns a value w such that $\Phi_2(w)$ holds. The symbol $\square$ is Iris's *persistence modality*. Here, it reflects the fact that continuations are multi-shot: a single `perform` expression can "return" several times with several different values of w, so we must be prepared to exploit Φ_2 several times.

To reason about labeled effects, we introduce the notion of a *protocol list* E, a list of pairs of a label and a protocol. Therefore, whereas Hazel's weakest precondition modality is parameterized with a single protocol, ours is parameterized with a protocol list. In our setting, the assertion $wp\ e\ \langle E\rangle\{\Phi\}$ means that (1) it is safe to execute e; (2) if e produces a value v then $\Phi(v)$ holds; and (3) if e performs an effect labeled ℓ then it does so according to a protocol Ψ such that $(\ell, \Psi) \in E$ holds. Its definition appears in Figure 6. It is broadly similar to Hazel's *wp* modality, save for three aspects: the use of a protocol list E; the use of a *persistent upward closure*; and the appearance of a *validity-and-distinctness property* as an assumption of the weakest precondition assertion. The persistent upward closure again has to do with the fact that continuations are multi-shot. The validity-and-distinctness property expresses two properties of the labels in the list E; first, these labels are pairwise distinct; second, these labels have been allocated. The latter fact is expressed by a *persistent points-to assertion* [37].

Interpretation of types (selected cases)

$$\begin{aligned}\mathcal{V}[\![\tau \xrightarrow{\rho} \kappa]\!]^{\delta}_{\eta}(v) &\triangleq \Box\,\forall\, w.\ \mathcal{V}[\![\tau]\!]^{\delta}_{\eta}(w) \mathrel{-\!\!*} \mathit{wp}\ (v\ w)\ \langle \mathcal{R}[\![\rho]\!]^{\delta}_{\eta}\rangle\{\mathcal{V}[\![\kappa]\!]^{\delta}_{\eta}\} \\ \mathcal{V}[\![\forall\theta.\ \tau]\!]^{\delta}_{\eta}(v) &\triangleq \forall\, E.\ \mathcal{V}[\![\tau]\!]^{\delta}_{\eta,\theta\mapsto E}(v)\end{aligned}$$

Interpretation of rows and signatures

$$\mathcal{R}[\![\rho]\!]^{\delta}_{\eta} \triangleq \bigcup_{\sigma\in\rho} \mathcal{S}[\![\sigma]\!]^{\delta}_{\eta} \qquad \begin{aligned}\mathcal{S}[\![(s : \iota \Rightarrow \kappa)]\!]^{\delta}_{\eta} &\triangleq (\delta(s),\ \{\mathcal{V}[\![\iota]\!]^{\delta}_{\eta}\}.\{\mathcal{V}[\![\kappa]\!]^{\delta}_{\eta}\}) \\ \mathcal{S}[\![\theta]\!]^{\delta}_{\eta} &\triangleq \eta(\theta)\end{aligned}$$

Interpretation of typing judgments

$$\begin{aligned}\Xi \mid \Delta \mid \Gamma \vDash e : \rho : \tau &\triangleq \forall\,\eta,\,\delta,\,vs.\ \mathcal{G}[\![\Gamma]\!]^{\delta}_{\eta}(vs) \mathrel{-\!\!*} \mathit{wp}\ (e[vs][\delta])\ \langle \mathcal{R}[\![\rho]\!]^{\delta}_{\eta}\rangle\{\mathcal{V}[\![\tau]\!]^{\delta}_{\eta}\} \\ \mathcal{G}[\![\Gamma]\!]^{\delta}_{\eta}(vs) &\triangleq \forall\,\{x \mapsto \tau\} \subseteq \Gamma.\ \mathcal{V}[\![\tau]\!]^{\delta}_{\eta}(vs(x))\end{aligned}$$

Fig. 7. Interpretation of types, rows, signatures, and typing judgments

This notion of *wp* enjoys a set of reasoning rules that we omit. The following theorem states that it is sound to reason about programs by means of these rules:

Theorem 1 (Soundness of TesLogic). *If wp e* $\langle [\,]\rangle\{\Phi\}$ *holds, then e is safe.*

With TesLogic at hand, let us come back to the definition of the semantic judgment $\Xi \mid \Delta \mid \Gamma \vDash e : \rho : \tau$.

As usual, a type τ is interpreted as a semantic type, that is, a persistent predicate $\mathcal{V}[\![\tau]\!]^{\delta}_{\eta}$ on values. More unusually, a row ρ is interpreted as a protocol list $\mathcal{R}[\![\rho]\!]^{\delta}_{\eta}$, defined as $\bigcup_{\sigma\in\rho} \mathcal{S}[\![\sigma]\!]^{\delta}_{\eta}$, the list concatenation of the interpretations of the elements of ρ. The environment δ maps effect names to effect labels; η maps type variables to semantic types and row variables to protocol lists.

This said, our interpretation of types (Figure 7) is mostly standard [19]. The interpretation of a function type, $\mathcal{V}[\![\tau \xrightarrow{\rho} \kappa]\!]^{\delta}_{\eta}$, is the set of values v such that the application of v to a value w in $\mathcal{V}[\![\tau]\!]^{\delta}_{\eta}$ satisfies a *wp* assertion with protocol list $\mathcal{R}[\![\rho]\!]^{\delta}_{\eta}$ and postcondition $\mathcal{V}[\![\kappa]\!]^{\delta}_{\eta}$. What is crucial is that the validity-and-distinctness property that we have built into the definition of *wp* formalizes the requirement that effect names be pairwise distinct. The interpretation of an effect-polymorphic type involves a quantification $\forall E$ over protocol lists.

Theorem 2 (Fundamental Theorem). *The syntactic judgment entails the semantic judgment:* $\Xi \mid \Delta \mid \Gamma \vdash e : \rho : \tau \implies \Xi \mid \Delta \mid \Gamma \vDash e : \rho : \tau$.

We establish this theorem by induction on the syntactic typing judgment. For every syntactic typing rule, we prove that the interpretation of the conclusion follows from the interpretations of the premises.

The previous two theorems lead directly to the desired type soundness result:

Theorem 3 (Soundness of TES). *If* $\emptyset \mid \emptyset \mid \emptyset \vdash e : \langle\rangle : \texttt{unit}$, *then e is safe.*

6 Related Work

Hillerström and Lindley [14] study the core calculus of Links [9], a functional programming language for web applications, which they extend with support for effect handlers. Taking advantage of Links's row-based approach to type-checking records, they annotate function types with rows of effects. Their rows use Rémy's kind discipline [32] to ensure that an effect name can never appear twice in a row.

Leijen [22] formalizes a subset of the Koka language [23]. He presents a calculus with support for handlers and globally defined effects, a type system with value and effect polymorphism, and a compilation strategy for explicitly-typed programs. This strategy relies on a *selective CPS transformation* [26], which he extends with support for effect polymorphism. A row in Leijen's system is *univariate*: it contains at most one row variable. TES, in contrast, allows a row to contain several row variables. This ability is exploited, for example, in the typing rule LEXHANDLE. Indeed, the premise contains the effect-polymorphic type $\forall\theta.\ (\alpha \xrightarrow{\theta} \beta) \xrightarrow{\theta\cdot\rho} \tau$, where θ abstracts away the fresh effect label that is allocated by `lex-handle`.

A notable omission from Leijen's formalization is Koka's `inject` [21], which is akin to a `lift` coercion. Biernacki et al. [4] are the first authors to provide a formal treatment of such a construct. They define its operational semantics and they propose a type system with effect polymorphism and univariate rows. They present the first binary logical relations for effect handlers, and they use these relations to prove that their system is sound. In a later paper [5], the same authors introduce λ^{HEL}, a calculus that supports both dynamic allocation of effect labels and effect coercions. In addition to the `lift` coercion, they consider (1) the *swap* coercion, which exchanges two effects in a row; (2) the *cons* coercion, which rearranges effects deep in a row; and (3) composition of coercions. These new coercions do not add expressiveness: they can be expressed in terms of `lift`. Still, they help programmers control the dynamic search for a handler. Biernacki et al. propose a type system with support for universal and existential types. Although `counter`, discussed in Sections 2 and 4, is expressible in λ^{HEL}, Biernacki et al.'s type system does not accept this program. (This has been confirmed by the authors in a personal communication.) The technical reason why `counter` is ill-typed is that the subsumption rules are not sufficiently flexible: an abstract row θ cannot be weakened to a larger row. It is not trivial how to overcome this issue, because the interpretation of a signature in Biernacki et al.'s system depends on the signature's position in the row. TES, in contrast, allows extension, thanks to the rule EXTEND.

Zhang and Myers [41] present "a new semantics based on tunneling", which they claim avoids "accidental handling" by construction. As far as we understand, however, they do not propose a semantics in the usual sense, that is, a reduction semantics. Instead, their "semantics" seems to be a translation of the surface language into a core calculus, $\lambda_{\Downarrow\Uparrow}$. This translation is not formally defined: it is sketched by way of examples. Furthermore, as noted by Biernacki et al. [6], there is a discrepancy between the paper presentation of $\lambda_{\Downarrow\Uparrow}$ and its Coq formalization. The paper does not mention dynamic generation of effect labels, but the calculus that is formalized in Coq supports this feature via a construct that generates a fresh effect label and installs a handler for this label; in other words, a lexically scoped handler.

For this calculus with lexically scoped handlers, Zhang and Myers propose a type system with support for effect polymorphism. They prove its soundness using binary logical relations. Then, they exploit these logical relations to establish interesting typed contextual equivalence laws. One law [41, Example 1] shows that an effect-polymorphic function cannot intercept the effects represented by an abstract row variable. This law seems to express the intuitive idea of "absence of accidental handling", but we remark that this notion is never formally defined.

Zhang and Myers [41] and other authors [8] suggest that "absence of accidental handling", sometimes also referred to as "effect safety", has something to do with parametricity. Unfortunately, "parametricity" itself is a somewhat loosely-defined concept. As far as we understand, the word "parametricity" refers to the fact that a syntactic universal type is interpreted via a meta-level universal quantification over a certain universe of semantic types. However, the strength of this meta-level quantification depends on which universe of semantic types is chosen. A smaller universe yields a system with weaker universal types, which may enjoy fewer equivalence laws, but may also admit more well-typed programs.

To illustrate this point, let us ask whether our calculus, TesLang, can be extended with a "dynamic-wind" construct [11]. This construct, `dynamic-wind` $p\ e\ q$, monitors the execution of e and invokes the thunk p whenever control enters e (at the beginning of e's execution and every time e is resumed) and invokes the thunk q whenever control leaves e (at the end of e's execution and every time e performs an effect). To type-check this construct, one might extend Tes with the following typing rule:

DynamicWind

$$\frac{\begin{array}{c}\varXi \mid \varDelta \mid \varGamma \vdash e : \rho : \tau \\ \varXi \mid \varDelta \mid \varGamma \vdash p : \rho : \texttt{unit} \rightarrow \texttt{unit} \qquad \varXi \mid \varDelta \mid \varGamma \vdash q : \rho : \texttt{unit} \rightarrow \texttt{unit}\end{array}}{\varXi \mid \varDelta \mid \varGamma \vdash \texttt{dynamic-wind}\ p\ e\ q : \rho : \tau}$$

We have proved that this rule is sound with respect to the interpretation of types presented in Section 5. So, our semantic model supports `dynamic-wind`. Furthermore, our semantic model arguably enjoys "parametricity", since a universal type is interpreted via a meta-level universal quantification. Yet, introducing `dynamic-wind` breaks Zhang and Myers's desired equivalence law [41, Example 1], because it allows observing arbitrary effects, without knowledge of their name and type. Therefore, "parametricity" does not guarantee "absence of accidental handling".

The lesson that we draw from this remark is that a programming language designer is faced with a tension between making the language more powerful by introducing constructs such as `dynamic-wind`, allowing new programs to be written, and making the language less powerful by forbidding such constructs, thereby validating new equivalence laws. Our (unary) semantic model (§5) errs on the side of admitting more constructs and fewer equivalence laws. In future work, it would be interesting to propose a (binary) semantic model that admits fewer constructs and validates more laws, so as to prove that TES without `dynamic-wind` validates Zhang and Myers's law [41, Example 1].

Despite their previous studies of coercions [4,5], Biernacki et al. [6] argue against coercions, which they deem impractical for real-world programming, and propose a type system for a language that supports lexically scoped handlers only. They present two semantics for this language: (1) an *open semantics*, where effect names are not substituted with labels, and where evaluation is defined among open terms in a capture-avoiding way; and (2) a *generative semantics*, where effect names are substituted at runtime with effect labels, as in TESLANG. By means of binary logical relations, they prove that the type system is sound and that the two semantics are equivalent.

Kammar and Pretnar [18] show that a calculus with effects and handlers but without references and without dynamic allocation of effect labels admits a type system with unrestricted polymorphism. Thus, generalization applies even to an expression that performs and handles effects. Kammar and Pretnar establish the soundness of their system via a syntactic approach [40]. The version of TES that we have formalized in Coq [36] distinguishes *pure* and *impure* expressions and allows generalizing the type of a pure expression. The pure expressions include expressions that perform or handle effects. Allocating a fresh effect label is still considered impure. Although such an allocation seems intuitively harmless, our current semantic model interprets allocation as an Iris "update", and Iris does not allow exchanging a universal quantifier with an update modality, so we are unable to justify that allocation is pure. We conjecture that this problem would perhaps not appear in a syntactic approach.

7 Conclusion

In this paper, we have argued in favor of a simple semantics for effect handlers, where the dynamic search for a handler is based purely on equality of effect labels, and where fresh labels can be generated at runtime. This language can express, but is not restricted to, lexically scoped handlers. We have proposed a type system equipped with type and effect polymorphism and with a powerful subsumption relation. A distinguishing feature is the idea that a row expresses a disjointness requirement on effect labels. We have established type soundness via a semantic approach.

In future work, it would be desirable to strengthen our semantic model and turn it into a binary model, so as to establish contextual equivalence laws such as Zhang and Myers's [41]. We also wish to investigate support for modules and inference of principal types, with the ultimate aim of proposing a strong type system for OCaml 5.

References

1. Ahmed, A.J., Fluet, M., Morrisett, G.: A step-indexed model of substructural state. In: International Conference on Functional Programming (ICFP). pp. 78–91 (Sep 2005)
2. Bauer, A., Pretnar, M.: An effect system for algebraic effects and handlers. Logical Methods in Computer Science **10**(4) (2014)
3. Bauer, A., Pretnar, M.: Programming with algebraic effects and handlers. Journal of Logical and Algebraic Methods in Programming **84**(1), 108–123 (2015)
4. Biernacki, D., Piróg, M., Polesiuk, P., Sieczkowski, F.: Handle with care: relational interpretation of algebraic effects and handlers. Proceedings of the ACM on Programming Languages **2**(POPL), 8:1–8:30 (2018)
5. Biernacki, D., Piróg, M., Polesiuk, P., Sieczkowski, F.: Abstracting algebraic effects. Proceedings of the ACM on Programming Languages **3**(POPL), 6:1–6:28 (2019)
6. Biernacki, D., Piróg, M., Polesiuk, P., Sieczkowski, F.: Binders by day, labels by night: effect instances via lexically scoped handlers. Proceedings of the ACM on Programming Languages **4**(POPL), 48:1–48:29 (2020)
7. Brachthäuser, J.I., Schuster, P., Ostermann, K.: Effects as capabilities: effect handlers and lightweight effect polymorphism. Proceedings of the ACM on Programming Languages **4**(OOPSLA), 126:1–126:30 (2020)
8. Brachthäuser, J.I., Schuster, P., Ostermann, K.: Effekt: Capability-passing style for type- and effect-safe, extensible effect handlers in Scala. Journal of Functional Programming **30**, e8 (2020)
9. Cooper, E., Lindley, S., Wadler, P., Yallop, J.: Links: Web programming without tiers. In: Formal Methods for Components and Objects. Lecture Notes in Computer Science, vol. 4709, pp. 266–296. Springer (Nov 2006)
10. Dolan, S., White, L.: Syntax with shifted names (Aug 2019), presented at the Workshop on Type-driven Development (TyDe)
11. Flatt, M., Yu, G., Findler, R.B., Felleisen, M.: Adding delimited and composable control to a production programming environment. In: International Conference on Functional Programming (ICFP). pp. 165–176 (Oct 2007)
12. Garrigue, J.: Relaxing the value restriction. In: Functional and Logic Programming. Lecture Notes in Computer Science, vol. 2998, pp. 196–213. Springer (Apr 2004)
13. Hendriks, D., van Oostrom, V.: adbmal. In: International Conference on Automated Deduction (CADE). Lecture Notes in Computer Science, vol. 2741, pp. 136–150. Springer (Aug 2003)
14. Hillerström, D., Lindley, S.: Liberating effects with rows and handlers. In: International Workshop on Type-Driven Development (TyDe@ICFP). pp. 15–27 (Sep 2016)
15. Hillerström, D., Lindley, S.: Shallow effect handlers. In: Asian Symposium on Programming Languages and Systems (APLAS). Lecture Notes in Computer Science, vol. 11275, pp. 415–435. Springer (Dec 2018)
16. Jung, R., Krebbers, R., Jourdan, J.H., Bizjak, A., Birkedal, L., Dreyer, D.: Iris from the ground up: A modular foundation for higher-order concurrent separation logic. Journal of Functional Programming **28**, e20 (2018)
17. Kammar, O., Lindley, S., Oury, N.: Handlers in action. In: International Conference on Functional Programming (ICFP). pp. 145–158 (Sep 2013)
18. Kammar, O., Pretnar, M.: No value restriction is needed for algebraic effects and handlers. Journal of Functional Programming **27**, e7 (2017)

19. Krebbers, R., Timany, A., Birkedal, L.: Interactive proofs in higher-order concurrent separation logic. In: Principles of Programming Languages (POPL) (Jan 2017)
20. Krogh-Jespersen, M., Svendsen, K., Birkedal, L.: A relational model of type-and-effects in higher-order concurrent separation logic. In: Principles of Programming Languages (POPL). pp. 218–231 (Jan 2017)
21. Leijen, D.: Koka: Programming with row polymorphic effect types. In: Workshop on Mathematically Structured Functional Programming (MSFP). vol. 153, pp. 100–126 (Apr 2014)
22. Leijen, D.: Type directed compilation of row-typed algebraic effects. In: Principles of Programming Languages (POPL). pp. 486–499 (Jan 2017)
23. Leijen, D.: Koka. `https://www.microsoft.com/en-us/research/project/koka/` (2020)
24. Leroy, X., Doligez, D., Frisch, A., Garrigue, J., Rémy, D., Vouillon, J.: The OCaml system: documentation and user's manual (Sep 2019)
25. Milner, R., Tofte, M., Harper, R., MacQueen, D.: The Definition of Standard ML – Revised. MIT Press (May 1997)
26. Nielsen, L.R.: A selective CPS transformation. Electronic Notes in Theoretical Computer Science **45**, 311–331 (Nov 2001)
27. Odersky, M., Boruch-Gruszecki, A., Brachthäuser, J.I., Lee, E., Lhoták, O.: Safer exceptions for Scala. In: Symposium on Scala. pp. 1–11 (Oct 2021)
28. Pessaux, F., Leroy, X.: Type-based analysis of uncaught exceptions. ACM Transactions on Programming Languages and Systems **22**(2), 340–377 (2000)
29. Pierce, B.C.: Types and Programming Languages. MIT Press (2002)
30. Plotkin, G.D., Pretnar, M.: Handlers of algebraic effects. In: European Symposium on Programming (ESOP). Lecture Notes in Computer Science, vol. 5502, pp. 80–94. Springer (Mar 2009)
31. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: Logic in Computer Science (LICS). pp. 55–74 (2002)
32. Rémy, D.: Type checking records and variants in a natural extension of ML. In: Principles of Programming Languages (POPL). pp. 77–88 (1989)
33. The Iris Team: HeapLang. `https://gitlab.mpi-sws.org/iris/iris/-/blob/master/iris_heap_lang/lang.v` (2022)
34. Tofte, M.: Type inference for polymorphic references. Information and Computation **89**(1), 1–34 (1990)
35. de Vilhena, P.E., Pottier, F.: A separation logic for effect handlers. Proceedings of the ACM on Programming Languages **5**(POPL) (Jan 2021)
36. de Vilhena, P.E., Pottier, F.: A type system for effect handlers and dynamic labels: Coq formalization. `https://gitlab.inria.fr/pdevilhe/tes` (2022)
37. Vindum, S.F., Birkedal, L.: Contextual refinement of the Michael-Scott queue. In: Certified Programs and Proofs (CPP). pp. 76–90 (Jan 2021)
38. Wand, M.: Type inference for record concatenation and multiple inheritance. Information and Computation **93**(1), 1–15 (Jul 1991)
39. Wright, A.K.: Simple imperative polymorphism. Lisp and Symbolic Computation **8**(4), 343–356 (Dec 1995)
40. Wright, A.K., Felleisen, M.: A syntactic approach to type soundness. Information and Computation **115**(1), 38–94 (Nov 1994)
41. Zhang, Y., Myers, A.C.: Abstraction-safe effect handlers via tunneling. Proceedings of the ACM on Programming Languages **3**(POPL), 5:1–5:29 (2019)

42. Zhang, Y., Salvaneschi, G., Beightol, Q., Liskov, B., Myers, A.C.: Accepting blame for safe tunneled exceptions. In: Programming Language Design and Implementation (PLDI). pp. 281–295 (Jun 2016)