A Type System for Effect Handlers and Dynamic Labels

PAULO EMÍLIO DE VILHENA, Inria, France
FRANÇOIS POTTIER, Inria, France

We propose a static type discipline for effect handlers in the presence of multiple effect names, dynamic allocation of effect names, and primitive mutable state. Our calculus, *λ-labels*, decouples the operations of generating a fresh effect name, installing a handler for a certain name, and performing an effect with a certain name. This flexibility allows several programming styles, including but not restricted to "lexically scoped handlers". Our type system, Tes, supports value and effect polymorphism. The introduction of polymorphism is restricted to pure expressions, a category that includes values as well as expressions that perform and handle effects. A function type is annotated with a row that is interpreted not only as a permission to perform effects with certain names and types but also as a requirement that these names be pairwise distinct. Tes guarantees *effect safety*: well-typed programs do not perform unhandled effects. We prove this fact by presenting a semantic interpretation of Tes into Hazel+, a novel program logic for *λ-labels*, built on top of Iris. Our results are formalized in Coq.

ACM Reference Format:
Paulo Emílio de Vilhena and François Pottier. 2022. A Type System for Effect Handlers and Dynamic Labels. 1, 1 (July 2022), 29 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

As written by Filinski [16], *"an effect is any deviation from the intuitive characterization of a program fragment as representing a simple function from inputs to outputs."* These deviations usually arise from the interaction of a program with its environment – for example, the memory, or other threads. Another source of deviation is the ability to capture an evaluation context and to reify it as a first-class value, a *continuation*. A language equipped with this ability allows the programmer to simulate the behavior of effectful operations, such as reading and updating the state of a memory cell, or raising exceptions. This possibility gives rise to *user-defined effects*. Filinski demonstrates these ideas in a Scheme-like language with *undelimited continuations*, which capture the entire evaluation context. Nowadays, there exist a myriad programming constructs to capture and manipulate continuations [17]. In this paper, we are interested in one particular construct: *effect handlers*.

Effect handlers have been introduced by Plotkin and Pretnar [36]. From an operational point of view, they can be seen as a generalization of exception handlers. Akin to raising an exception, a program can *perform an effect* to interrupt the normal flow of execution and transfer control to an effect 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.

Delimited continuations offer simpler and stronger reasoning principles than their undelimited counterparts [25]. In recent work, Dolan et al. [15] show that effect handlers have interesting

Authors' addresses: Paulo Emílio de Vilhena, Inria, Paris, France, paulo-emilio.de-vilhena@inria.fr; François Pottier, Inria, Paris, France, francois.pottier@inria.fr.

XXXX-XXXX/2022/7-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

applications to systems programming. In particular, they show how effect handlers can be used to write an *asynchronous computation* library, whose clients can fork lightweight threads and wait for a thread to produce a result. Another interesting application of effect handlers is *control inversion*, which allows converting a higher-order iteration method for a collection of elements into a lazy sequence of elements [14], or in other words, converting a "push" producer into a "pull" producer. More generally, it has been argued that, "by separating effect signatures from their implementation, [effect handlers] provide a high degree of modularity" [23]. For these reasons, effect handlers are finding their way into research programming languages such as Eff [3, 4], Effekt [10, 11], Frank [31], Koka [28, 30], Links [21], and into mainstream programming languages such as OCaml 5 [39].

The adoption of effect handlers in typed programming languages gives rise to the question: how to extend a static type system with support for effect handlers? This is an important question: a type system establishes strong guarantees, such as memory safety or the absence of unhandled effects, thus protecting users at compile time against a class of bugs. Moreover, types are a source of documentation, revealing how the program interacts with its environment, but hiding implementation details. However, before addressing the problem of designing a type system for a language with support for effect handlers, we must first decide exactly what language and what dynamic semantics we want. An ideal language should be expressive yet simple; its dynamic semantics should be easy to understand. In this paper, we argue in favor of two features: *multiple named effects* and *dynamic allocation of effect labels*.

Multiple named effects. A program often exploits more than one form of effect. Thus, we would like effects to be named, that is, to carry an identifier that is used to match client and handler. More precisely, we would like the expression `perform` s v to perform an effect identified by the name s and carrying payload v. Accordingly, we would like an effect handler to indicate which effects it wishes to handle, based on the name s. Thus, in the effect-handling construct `handle` e `with` $(s : h \mid r)$, we would like the *effect branch* h to be invoked when the expression e performs an effect named s, and we would like the *return branch* r to be invoked if e terminates normally.

Dynamic allocation of effect labels. Dynamic allocation of effect labels is the ability to introduce fresh effect labels. We would like to the language to have a construct of the form `effect` s `in` e, which binds the *effect name* s to a freshly generated *effect label*. This distinction between effect names and effect labels is similar to the distinction between variables and memory locations that is traditionally used in the operational semantics of mutable references [35]. An argument in favor of this feature is that dynamic allocation of effect labels can serve as a tool to defend against *accidental handling* [44]. It is also worth noting that dynamic allocation of exception labels has existed for a long time in Standard ML and in OCaml. OCaml 5 has dynamic allocation of effect labels; the calculus and dynamic semantics that we present in this paper are intended to closely model the semantics of OCaml 5.

Accidental handling. Accidental handling is an *informal* concept that can be described roughly as follows: accidental handling occurs when an effect handler intercepts an effect that it was not meant to handle. For example, consider the function *bad_counter*, defined as follows:

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

(This example and the following ones in this section are written in a calculus whose syntax and semantics are introduced in Section 2.) This program uses an effect named *tick* to implement a memory cell in state-passing style. The application *bad_counter ff* is intended to produce a wrapped version of the second-order function *ff* that counts the number of times *ff* calls its argument. However, in some cases, the function *bad_counter* does not behave as intended, due to

accidental handling. Indeed, consider the following program:

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

The expected result of this program is $(((), 1), 1)$, because the function $(\lambda f.\ f\ ())$ calls its argument once, and its wrapped version, produced by the inner application of *bad_counter*, should preserve this property. However, the actual result is the value $(((), 2), 0)$. At runtime, two handlers are installed. Each handler counts how many *tick* effects it observes. The client, which is nested inside the two handlers, performs the effect *tick* twice. Each *tick* effect is intended to be a message to update each counter once, but the innermost handler erroneously intercepts both messages. The effect *tick* is *accidentally handled*, thus leading to unintended behavior.

To avoid accidental handling, precautions can be taken at several levels: the the syntax and dynamic semantics of the programming language can be altered so as to make programmer mistakes less likely; and/or a static type discipline can be imposed. At the language design level, the literature describes two mechanisms that are intended to help protect against accidental handling: (1) *lexically scoped handlers* [7, 9, 10] and (2) *effect coercions* [5].

Lexically scoped handlers. The characteristic intended feature of a "lexically scoped handler" is the fact that one can statically tell which handler is invoked when an effect is performed [38]. In our calculus, this can be achieved by installing a handler for a freshly-generated effect label, that is, a label that is generated immediately before the handler is installed. In other words, lexically scoped handlers can be simulated using ordinary effect handlers and dynamic generation of effect labels. The encoding is as follows:

$$\texttt{lex-handle}\ e\ \texttt{with}\ (h \mid r) = \texttt{effect}\ s\ \texttt{in}\ \texttt{handle}\ (e\ (\lambda x.\ \texttt{perform}\ s\ x))\ \texttt{with}\ (s\ \texttt{:}\ h \mid r) \qquad (1)$$

This code first generates a fresh effect label and binds the name s to this label. Then, it installs a handler, which monitors the application of the expression e to the function $\lambda x.\ \texttt{perform}\ s\ x$.

Coming back to the example of the function *bad_counter*, one can employ a lexically scoped handler to correct the code, as follows:

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

The variable *tick* now stands for a function that performs an effect named s. Every time *counter* is invoked, a fresh label ℓ is allocated, and the local name s is bound to this label. Therefore, the program:

$$\mathit{counter}\ (\mathit{counter}\ (\lambda f.\ f\ ())\ (\lambda\_.\ ())$$

reduces to the value $(((), 1), 1)$, as desired, because the two applications of *counter* install two handlers for two distinct dynamic labels.

Effect coercions. An effect coercion is an operation that modifies the manner in which a client that performs an effect is matched with one of the enclosing handlers. Perhaps the most illustrative example is that of the *lift* coercion [5]. Usually, performing an effect named s transfers control to the innermost enclosing handler that selects the name s. In the presence of a *lift* coercion, however, control is transferred instead to the *second* closest handler. To this end, the `perform` instruction must be wrapped in a *lift* coercion: `lift` s (`perform` $s\ v$). Here is how one could employ such a coercion to protect the function *bad_counter* against accidental handling:

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

As desired, the program:

$$\mathit{lift\_counter}\ (\mathit{lift\_counter}\ (\lambda f.\ f\ ())\ (\lambda\_.\ ())$$

reduces to the value $((),1),1)$, because among the two *tick* effects performed by the client, one *tick* effect is intercepted by the innermost handler, whereas the other is intercepted by the outermost handler thanks to a *lift* coercion.

Between these two mechanisms of protection against accidental handling, we argue in favor of lexically scoped handlers. However, as we have seen, lexically scoped handlers can be simulated in terms of dynamic allocation of effect labels and the ordinary effect handlers, which are independent language constructs. Therefore, we argue *against* restricting the programming language to lexically scoped handlers only, and *against* effect coercions.

Our argument against effect coercions is unwarranted complexity. Effect coercions complicate the dynamic semantics of the language, and are potentially difficult to explain to programmers. Dynamic allocation of effect labels allows avoiding accidental handling, while preserving a simple, standard dynamic semantics.

Argument against the restriction to lexically scoped handlers. Our argument against a restriction to lexically scoped handlers is threefold:

(1) **Unusual style.** Programming languages with support for exceptions, such as OCaml, Python, and Java, propose a set of globally defined exception names, to which every program has access. Programmers agree to throw each of these exceptions in specific situations – for example, when the task of searching for an element terminates unsuccessfully, or when an arithmetic computation causes a division by zero.
In a language with support for effect handlers, one can imagine a similar convention. Every program can have access to a set of globally defined effect names, and programmers can agree to perform each of these effects in specific situations. For example, when programming with *coroutines* [13], programmers might agree to perform a globally defined *yield* effect to produce elements. For example, the following *filter* function iterates over a list xs and "yields" the elements that satisfy a certain predicate f. By convention, an element is "yielded" by performing a *yield* effect.

$$\mathit{filter}\ xs\ f = \mathit{iter}\ xs\ (\lambda x.\ \texttt{if}\ f\ x\ \texttt{then perform}\ \mathit{yield}\ x) \tag{3}$$

We assume that *iter* is a higher-order iteration combinator, which applies a function in succession to every element of a list. Another function, *reassemble*, constructs a list of the elements yielded by a function g:

$$\mathit{reassemble}\, g = \texttt{handle}\ (g\ \texttt{()})\ \texttt{with}\ (\mathit{yield} : \lambda x\, k.\, x :: k\ \texttt{()} \mid \lambda\_.\ \texttt{[]})$$

Filtering a list to obtain a new list is just a matter of combining the previous two functions:

$$\mathit{reassemble}\,(\lambda\_.\ \mathit{filter}\ xs\ f) \tag{4}$$

With lexically scoped handlers, one can write similar programs, albeit in an unusual style. Since an effect label can be allocated only when the handler is installed, a client has to take an extra parameter representing the function that performs this effect. For example, here is how one could define *filter* in a language that is restricted to lexically scoped handlers:

$$\mathit{filter}'\ xs\ f\ \mathit{yield} = \mathit{iter}\ xs\,(\lambda x.\ \texttt{if}\ f\ x\ \texttt{then}\ \mathit{yield}\ x)$$

And here is how one could define *reassemble* using a lexically scoped handler:

$$\mathit{reassemble}'\ g = \texttt{lex-handle}\ g\ \texttt{with}\ (\lambda x\, k.\, x :: k\ \texttt{()} \mid \lambda\_.\ \texttt{[]})$$

This style becomes more cumbersome when a program performs multiple named effects: a client needs to take one extra parameter for each effect name.

An argument in favor of this style would be that *reassemble′* protects against accidental handling. Indeed, consider the following program:

$$\mathit{reassemble}'\,(\mathit{filter}'\,\mathit{xs}\,f) \tag{5}$$

Because *reassemble′* installs a lexically scoped handler, we are assured that this handler will not intercept the effects that f might perform. On the other hand, the handler installed by *reassemble*, in the program 4, might accidentally intercept a *yield* effect that f might perform. Our rebuttal is that, to protect against accidental handling in program 4, one can specify, as an assumption to the application of *filter*, that f must not perform *yield* effects. As we shall see in Section 3, the programmer can express such a specification using the type system introduced in this paper. Our system ensures that, in every application of *filter*, the function f does not perform *yield* effects.

(2) **Not backwards compatible.** A restriction to lexically scoped handlers runs contrary to the design choices made in OCaml 5 [39], which does not impose such a restriction. Indeed, OCaml supports unrestricted dynamic allocation of effect labels.

(3) **Theoretically unsatisfying.** Because a lexically scoped handler is simply the combination of the allocation of a fresh label and the installation of a handler for this label, it is theoretically unsatisfying to have a language that supports only lexically scoped handlers. With unrestricted dynamic allocation of effect labels, the programmer can choose between allocating effect labels globally (at the top level of the program) or locally.

Type system. Having decided on which semantic features we are interested in, namely multiple named effects and dynamic allocation of effect labels, we can now address the question of designing a type system. Let us discuss what type system features and what static guarantees are desired.

Desired features.

(1) **Annotated arrows.** In addition to an argument type α and a return type β, an annotated arrow includes a *row* ρ [19, 28, 37], which specifies the effects that a function might perform. Annotated arrows are a desired feature, because it is essentially impossible to achieve effect safety (discussed below) without them.

(2) **Effect polymorphism.** Effect polymorphism is the possibility to reuse a program component in several different contexts that are prepared to perform or handle different effects. A typical example is that of the higher-order iteration combinator *iter*, which applies its argument f to the elements of a collection. Effect polymorphism is a desired feature, because we would like to express that *iter* does not care what effects f might perform. More precisely, if the elements have type α, then we would like *iter* to have the type $\forall\theta.\ (\alpha \xrightarrow{\theta} ()) \xrightarrow{\theta} ()$, which is universally quantified over the effect of the function f.

Desired static guarantees. An important desired static guarantee is *effect safety*: that, during the execution of a complete (closed) program, no effect must be unhandled. In other words, performing an effect named s outside of the scope of a handler for this name must be statically forbidden. This safety guarantee is sometimes also known as *strong type safety*. In contrast, *weak type safety* rules out ordinary type errors, such as calling a function with an incorrect number of arguments, but allows unhandled effects.

The "absence of accidental handling", sometimes also referred to as "abstraction safety", might also be considered a desirable property. However, this property is only loosely defined in the literature. Zhang and Myers [44] seem to suggest that it is connected in some sense with *parametricity of effect polymorphism*. However, parametricity itself is loosely defined. Parametricity requires a universal quantifier in the syntax to be interpreted by a meta-level universal quantification over some universe of semantic types. However, *which* universe of semantic types is chosen matters,

and this creates a tension between conflicting goals: while a larger universe allows establishing more contextual equivalence laws, a smaller universe allows type-checking more programming language constructs, such as "dynamic-wind". We continue this discussion later in the paper (§5). For now, let us state that "absence of accidental handling" is not a well-defined goal. The existence of a sound semantic interpretation of types, and the ability to exploit parametricity to establish strong program equivalence laws, are more clearly-defined goals. In this paper, we define a sound semantic model for our type system, but leave the study of equivalence laws to future work.

This paper. The main contribution of this paper is the introduction of Tes, a type system for a language with support for effect handlers, multiple named effects, dynamic allocation of effect labels, and general references. The type system supports effect polymorphism and ensures the absence of unhandled effects. One key novelty of Tes is the meaning assigned to arrow types. An arrow type implicitly expresses the *requirement* that the effect labels in its row are pairwise distinct. In Section 3, we show how to exploit this feature in a number of examples. Moreover, Tes offers relaxed subsumption rules that allow, among other properties, the extension and permutation of rows. With simple yet flexible typing rules, Tes is the first system that accepts the program *counter* and that is not restricted to lexically scoped handlers. We give a more detailed comparison of Tes with previous work in Section 5. As a second contribution, in Section 4, we introduce a novel Separation Logic for reasoning about multiple named effects and multi-shot continuations. We use this logic as to offer a semantic interpretation of Tes and to prove that Tes ensures effect safety. As a third contribution, we study the combination of effects, mutable state, and polymorphism, and point out that the fundamental reason why this combination (if unrestricted) is unsound is the lack of commutation between the universal quantifier and the "update" modality. In Tes, typing judgments carry *purity attributes*, which indicate whether a program interacts with the store. A program that is considered *pure* by the type system satisfies a specification that does not involve an "update" modality; therefore, its type can be generalized. All of our results are formalized in Coq and are available online.

2 SYNTAX AND SEMANTICS

We introduce *λ-labels*, a calculus with mutable state, effect handlers, multiple named effects, dynamic allocation of effect labels, and multi-shot continuations.

2.1 Syntax

The syntax of values, expressions, and evaluation contexts is shown in Figure 1. Most of the constructs are standard. They include recursive functions, binary products, binary sums, binary and unary operations, lists, and references. We use infix notation when writing the application of a binary operation to a pair of arguments. We define a non-recursive function as an anonymous recursive function, $\lambda x.\, e \triangleq \texttt{rec}\ \_\ x.\, e$, and we encode let binding as function application, $\texttt{let}\ x = e_1\ \texttt{in}\ e_2 \triangleq (\lambda x.\, e_2)\ e_1$. Non-standard constructs include first-class continuations, $\texttt{cont}\ K$, and instructions to perform and handle effects. $\texttt{perform}\ n\ e$ and $\texttt{handle}\ e\ \texttt{with}\ (n : h \mid r)$, respectively. The argument e of the instruction to perform effects is called the effect *payload*. Both of the instructions to perform and handle effects carry an effect identifier n, which is either a variable s, an *effect name*, or a memory location ℓ, an *effect label*. Even though source programs use only effect names as identifiers, the syntax of expressions must allow an effect identifier to be a location so that the syntax is closed by the reduction relation. These memory locations correspond to fresh effect labels introduced by the construct $\texttt{effect}\ s\ \texttt{in}\ e$. Finally, the syntax includes *active effects*, $\texttt{eff}\ \ell\ v\ K$, which are also not part of source programs, but which play a role in the definition of the operational semantics as we shall explain in the next subsection.

Variables and effect identifiers

$$\mathit{Var} \ni f, x \qquad \mathit{EffId} \ni n ::= s \mid \ell \ (\in \mathit{Loc})$$

Values, expressions, and operations

$$\begin{aligned}
\mathit{Op} \ni \odot &::= \texttt{+} \mid \texttt{not} \mid \texttt{and} \mid \texttt{or} \mid \texttt{==} \\
\mathit{Val} \ni v &::= \texttt{()} \mid b \ (\in \mathit{Bool}) \mid i \ (\in \mathit{Int}) \mid \ell \ (\in \mathit{Loc}) \mid \odot \ (\in \mathit{Op}) \\
&\quad \mid \texttt{rec}\ f\ x.\ e \mid (v, v) \mid \texttt{inj}_i\ v \mid v :: v \mid \texttt{[]} \mid \texttt{cont}\ K \\
\mathit{Expr} \ni e &::= v \mid x \mid e\ e \mid e :: e \mid (e, e) \mid \texttt{proj}_i\ e \mid \texttt{inj}_i\ e \\
&\quad \mid \texttt{match}\ e\ \texttt{with}\ (v \mid v) \mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \mid \texttt{ref}\ e \mid \texttt{!}e \mid e \coloneqq e \\
&\quad \mid \texttt{effect}\ s\ \texttt{in}\ e \mid \texttt{perform}\ n\ e \mid \texttt{eff}\ \ell\ v\ K \mid \texttt{handle}\ e\ \texttt{with}\ (n\ \texttt{:}\ v \mid v)
\end{aligned}$$

Evaluation contexts

$$\begin{aligned}
\mathit{Ectx} \ni K &::= \bullet \mid e\ K \mid K\ v \mid K :: v \mid e :: K \mid (e, K) \mid (K, v) \mid \texttt{proj}_i\ K \mid \texttt{inj}_i\ K \\
&\quad \mid \texttt{match}\ K\ \texttt{with}\ (v \mid v) \mid \texttt{if}\ K\ \texttt{then}\ e\ \texttt{else}\ e \mid \texttt{ref}\ K \mid \texttt{!}K \mid e \coloneqq K \mid K \coloneqq v \\
&\quad \mid \texttt{perform}\ \ell\ K \mid \texttt{handle}\ K\ \texttt{with}\ (\ell\ \texttt{:}\ v \mid v)
\end{aligned}$$

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

Head-reduction relation $\boxed{e \mid \sigma \longrightarrow e \mid \sigma}$

$$\begin{aligned}
(\texttt{cont}\ K)\ v \mid \sigma &\longrightarrow K[v] \mid \sigma \\
\texttt{effect}\ s\ \texttt{in}\ e \mid \sigma &\longrightarrow e[\ell/s] \mid \sigma[\ell \mapsto \texttt{()}] \quad \ell \notin \operatorname{dom} \sigma \\
\texttt{perform}\ \ell\ v \mid \sigma &\longrightarrow \texttt{eff}\ \ell\ v\ \bullet \mid \sigma \\
(\texttt{eff}\ \ell\ v_1\ K)\ v_2 \mid \sigma &\longrightarrow \texttt{eff}\ \ell\ v_1\ (K\ v_2) \mid \sigma \\
e_1\ (\texttt{eff}\ \ell\ v_2\ K) \mid \sigma &\longrightarrow \texttt{eff}\ \ell\ v_2\ (e_1\ K) \mid \sigma \\
\texttt{handle}\ (\texttt{eff}\ \ell_2\ v\ K)\ \texttt{with}\ (\ell_1\ \texttt{:}\ h \mid r) \mid \sigma &\longrightarrow \texttt{eff}\ \ell_2\ v\ (\texttt{handle}\ K\ \texttt{with}\ (\ell_1\ \texttt{:}\ h \mid r)) \mid \sigma \quad \ell_1 \neq \ell_2 \\
\texttt{handle}\ (\texttt{eff}\ \ell\ v\ K)\ \texttt{with}\ (\ell\ \texttt{:}\ h \mid r) \mid \sigma &\longrightarrow h\ v\ (\texttt{cont}\ (\texttt{handle}\ K\ \texttt{with}\ (\ell\ \texttt{:}\ h \mid r))) \mid \sigma \\
\texttt{handle}\ v\ \texttt{with}\ (\ell_1\ \texttt{:}\ h \mid r) \mid \sigma &\longrightarrow r\ v \mid \sigma
\end{aligned}$$

Fig. 2. Selected head-reduction rules.

2.2 Semantics

The small-step operational semantics of *λ-labels* involves two relations: a reduction relation and an evaluation relation. Both of them act on pairs of a program and a *store* σ. A store is a finite map from memory locations to values. The *head-reduction relation*, of which an excerpt appears in Figure 2, governs how programs reduce and how they affect the store. A *reduction relation* is then defined by allowing reduction to take place under evaluation contexts.

The syntax of evaluation contexts defines a right-to-left evaluation order; this choice is arbitrary: it is inspired by HeapLang's [22] evaluation order, but the results of this paper also hold in a left-to-right evaluation order. The omitted reduction rules, such as β-reduction, and the rules for allocating, reading and writing references, are standard. Continuation resumption is simple: the application of a value v to a continuation $\texttt{cont}\ K$ reduces to $K[v]$, the term obtained by filling the hole of K with v. The remaining reduction rules in Figure 2 illustrate the subtle aspects of the calculus: introducing a fresh effect label, and performing and handling effects.

Type variables, row variables, and purity attributes

$$\mathit{TypeVar} \ni \alpha, \beta, \gamma \qquad \mathit{RowVar} \ni \theta \qquad \mathit{Attribute} \ni a ::= P \mid I$$

Types, rows, and signatures

$$\begin{aligned} \mathit{Type} \ni \tau, \upsilon, \kappa ::= {} & \texttt{()} \mid \texttt{bool} \mid \texttt{int} \mid \bot \mid \top \\ & \mid \alpha \mid \tau\ \texttt{list} \mid \tau * \tau \mid \tau + \tau \mid \tau\ \texttt{ref} \\ & \mid \tau \xrightarrow{\rho}_{a} \tau \mid \forall \alpha.\ \tau \mid \forall \theta.\ \tau \end{aligned} \qquad \begin{aligned} \mathit{Row} \ni \rho ::= {} & \langle\rangle \mid \sigma \cdot \rho \\ \mathit{Sig} \ni \sigma ::= {} & (s : \tau \Rightarrow \tau) \mid \theta \end{aligned}$$

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

Introducing a fresh effect label. Fresh effect labels are introduced using the store: the instruction `effect` s `in` e allocates a memory location ℓ, initialized with the value `()`, and performs the substitution of ℓ for the variable s in the expression e.

Performing and handling effects. Performing an effect transfers control to the handler. Indeed, the instruction `perform` $\ell\ v$ reduces to the active effect `eff` $\ell\ v\ \bullet$, which start the mechanism of capture of the evaluation context. An active effect swallows the evaluation context, frame by frame, until it reaches a handler. If the handler selects an effect label other than ℓ, then it continues the capture mechanism and swallows the handler. If the handler includes the effect label ℓ, then it transfers control to the effect branch h of handler. The effect branch h receives both the argument v with which the effect was performed, and the reification of the captured evaluation context K as a first-class continuation `cont` K. Notice that the reified continuation includes the effect handler. This is a *deep-handler semantics* [20]: resuming the continuation reinstalls the handler.

3 TYPE SYSTEM

In this section, we define Tes, a type system for *λ-labels*. First, we present the syntax of types. Second, we introduce the set of typing rules and the subsumption relation. Finally, we illustrate the strength and the subtleties of the system through a number of examples.

3.1 Syntax of types, rows, and signatures.

Figure 3 shows the syntax of types, rows, and signatures. The set *TypeVar* is an infinite set of type variables, ranged over by α, β, and γ, and *RowVar* is an infinite set of row variables, ranged over by θ. We also introduce the set *Attribute* of *purity attributes*. A purity attribute is either I for "impure" – to indicate that a program can interact with the store by introducing fresh effect labels, or by allocating, updating, or reading references – or P for "pure" – to indicate that a program cannot interact with the store. As we shall explain in the next subsection, the introduction of purity attributes is related to the introduction of polymorphic types and the *value restriction* [18, 42].

Most types are standard; they include the unit type `()`, top and bottom types, noted as $\top$ and $\bot$, respectively, the type of Booleans `bool`, the type of integers `int`, sum and product types, the type of lists, and reference types. The syntax also includes value-polymorphic types to universally quantify over type variables α. The two remaining types, annotated arrow types and effect-polymorphic types, are the most interesting, and we discuss them separately.

Annotated arrow types. In addition to an argument type υ and a return type τ, an arrow type $\upsilon \xrightarrow{\rho}_{a} \tau$ includes a purity attribute a and a *row* ρ. A row is a list of *signatures* σ. (The empty row $\langle\rangle$ denotes the empty list, and the the row composition $\_ \cdot \_$ denotes list cons.) A signature, in its turn, is either a *concrete signature* $(s : \upsilon' \Rightarrow \tau')$ or an *abstract signature* θ. A concrete signature $(s : \upsilon' \Rightarrow \tau')$

Typing judgment $\boxed{\Xi \mid \Delta \mid \Gamma \vdash_a e : \rho : \tau}$

UnitTyped
$$\Xi \mid \Delta \mid \Gamma \vdash_a () : \rho : ()$$

BoolTyped
$$\Xi \mid \Delta \mid \Gamma \vdash_a b : \rho : \texttt{bool}$$

IntTyped
$$\Xi \mid \Delta \mid \Gamma \vdash_a i : \rho : \texttt{int}$$

OpTyped
$$\frac{\vdash_{Op} \odot : v \to \tau}{\Xi \mid \Delta \mid \Gamma \vdash_a \odot : \rho : v \xrightarrow{\rho}_a \tau}$$

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

TLamTyped
$$\frac{\alpha \notin \Xi, \Gamma, \rho, \tau \qquad \Xi, \alpha \mid \Delta \mid \Gamma \vdash_P e : \rho : \tau}{\Xi \mid \Delta \mid \Gamma \vdash_a e : \rho : \forall \alpha.\, \tau}$$

RLamTyped
$$\frac{\theta \notin \Xi, \Gamma, \rho, \tau \qquad \Xi, \theta \mid \Delta \mid \Gamma \vdash_P e : \rho : \tau}{\Xi \mid \Delta \mid \Gamma \vdash_a e : \rho : \forall \theta.\, \tau}$$

ReadTyped
$$\frac{\Xi \mid \Delta \mid \Gamma \vdash_a e : \rho : \tau\ \texttt{ref}}{\Xi \mid \Delta \mid \Gamma \vdash_I !e : \rho : \tau}$$

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

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

WriteTyped
$$\frac{\Xi \mid \Delta \mid \Gamma \vdash_a e : \rho : \tau\ \texttt{ref} \qquad \Xi \mid \Delta \mid \Gamma \vdash_{a'} e' : \rho : \tau}{\Xi \mid \Delta \mid \Gamma \vdash_I e := e' : \rho : ()}$$

LetEffTyped
$$\frac{s \notin \Gamma, \rho, \tau \qquad \Xi \mid \Delta, s \mid \Gamma \vdash_a e : (s : \texttt{abs}) \cdot \rho : \tau}{\Xi \mid \Delta \mid \Gamma \vdash_I \texttt{effect}\ s\ \texttt{in}\ e : \rho : \tau}$$

AllocTyped
$$\frac{\Xi \mid \Delta \mid \Gamma \vdash_a e : \rho : \tau}{\Xi \mid \Delta \mid \Gamma \vdash_I \texttt{ref}\ e : \rho : \tau\ \texttt{ref}}$$

PerformTyped
$$\frac{\Xi \mid \Delta \mid \Gamma \vdash_a e : \rho : v \qquad (s : v \Rightarrow \tau) \in \rho \qquad s \in \Delta}{\Xi \mid \Delta \mid \Gamma \vdash_a \texttt{perform}\ s\ e : \rho : \tau}$$

RecTyped
$$\frac{\Xi \mid \Delta \mid f : v \xrightarrow{\rho}_a \tau, x : v, \Gamma \vdash_a e : \rho : \tau}{\Xi \mid \Delta \mid \Gamma \vdash_{a'} \texttt{rec}\ f\ x.\ e : \langle\rangle : v \xrightarrow{\rho}_a \tau}$$

HandleTyped
$$\frac{\sigma = (s : v \Rightarrow \tau) \qquad \sigma' = (s : v' \Rightarrow \tau') \qquad \rho' = \sigma' \cdot \rho \qquad s \in \Delta \qquad \Xi \mid \Delta \mid \Gamma \vdash_a e : \sigma \cdot \rho : \kappa \qquad \Xi \mid \Delta \mid \Gamma \vdash_a h : \rho' : v \to_a (\tau \xrightarrow{\rho'}_a \kappa') \xrightarrow{\rho'}_a \kappa' \qquad \Xi \mid \Delta \mid \Gamma \vdash_a r : \rho' : \kappa \xrightarrow{\rho'}_a \kappa'}{\Xi \mid \Delta \mid \Gamma \vdash_a \texttt{handle}\ e\ \texttt{with}\ (s : h \mid r) : \rho' : \kappa'}$$

MonotonicityTyped
$$\frac{a \leq_A a' \qquad \vdash_b \rho \leq_R \rho' \qquad \Xi \mid \Delta \mid \Gamma \vdash_a e : \rho : \tau \qquad \vdash \tau \leq_T \tau'}{\Xi \mid \Delta \mid \Gamma \vdash_{a'} e : \rho' : \tau'}$$

AppTyped
$$\frac{\Xi \mid \Delta \mid \Gamma \vdash_a e : \rho : v \xrightarrow{\rho}_a \tau \qquad \Xi \mid \Delta \mid \Gamma \vdash_a e' : \rho : v}{\Xi \mid \Delta \mid \Gamma \vdash_a e\ e' : \rho : \tau}$$

Fig. 4. Typing rules.

indicates that performing the effect s is analogous to calling a function of argument type v' and return type τ'. According to this reading, the signature $(s : \bot \Rightarrow \top)$, noted $(s : \texttt{abs})$, forbids f from performing s. We call this signature the *absence signature*. An abstract signature θ is simply a row variable: it stands for a row ρ' possibly containing multiple concrete and abstract signatures. Intuitively, a program f of type $v \xrightarrow{\rho}_a \tau$ is a function that, when applied to an argument of type v, will either return a result of type τ or perform an effect whose signature appears in the row ρ.

Moreover, if a is the impure attribute I, then this function can interact with the store during its evaluation; otherwise it cannot do so. Moreover, Tes introduces a novel aspect to the reading of an arrow: *Tes includes the requirement that the effect labels bound by signatures in ρ are pairwise distinct.* In other words, the program f can assume that the dynamic instances of the effects in ρ are distinct from one another. We say that a row ρ is *dynamically distinct* if it satisfies this separation requirement. The syntax of rows includes rows that are not dynamically distinct (for example, a row can have two occurrences of the same signature $(s : \_ \Rightarrow \_)$), however, if a function carries such a row, then this function cannot be called, because the separation requirement does not hold. Finally, we remark that, when the purity attribute is omitted from an arrow, the attribute I is chosen by default.

Effect-polymorphic types. A row might contain one or more row variables. An effect-polymorphic type adds the ability to quantify universally over such variables. Intuitively, a program of type $\forall\theta.\ \tau$ has a behavior that does not depend on the set of effects abstracted by θ. Recall the typical example of the effect-polymorphic *iter* function discussed in Section 1: a higher-order iteration method whose behavior is insensitive to the set of effects performed by its iteratee. An example of such an iteration method is the function *iter*, defined as follows:

$$\mathit{iter} = \texttt{rec}\ \mathit{iter}\ \mathit{xs}\ f.\ \texttt{match}\ \mathit{xs}\ \texttt{with}\ (\lambda x\ \mathit{xs}.\ f\ x\texttt{;}\ \mathit{iter}\ \mathit{xs}\ f \mid \lambda\_.\ \texttt{()}) \tag{6}$$

With effect (and value) polymorphism, we can assign the following type to *iter*:

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

This type tells that *iter* is independent both of the representation of the elements of *xs* and of the set of effects that f might perform. In particular, the function *iter* does not perform any effects and does not intercept the effects performed by f.

3.2 Typing judgment

A typing judgment in Tes depends on three environments: (1) a *row- and type-variable context* Ξ, which is a set of row and type variables θ and α, (2) an *effect-name context* Δ, which is a set of effect names, and (3) a *value context* Γ, which is a map from variables x to types τ. A typing judgment relates an expression e to a purity attribute a, a row ρ, and a type τ. Intuitively, such a judgment asserts that, during the evaluation of e, this program may perform effects according to ρ, interact with the store according a, and produce an output of type τ. Moreover, the program e can assume that ρ is dynamically distinct.

A selection of the typing rules appears in Figure 4. (The complete set of typing rules appears in the Appendix (§7).) These rules depend on the *operation judgment*, which associates an operation to a pair of an argument type υ and a return type τ. We write this pair as $\upsilon \to \tau$. (The derivation rules of the operation judgment appear in the Appendix (§7).)

We divide the remainder of this subsection into three parts: (1) purity attributes and the value restriction, (2) an overview of the main typing rules, and (3) the monotonicity rule and the subsumption relations.

3.2.1 Purity attributes and the value restriction. It is well-known that, in the presence of mutable state, the unrestricted introduction of polymorphic types is unsound [40]. One solution is the value restriction [18, 42]: to restrict polymorphism to values. With purity attributes, we adopt a slightly more general solution: we restrict the introduction of polymorphic types to *pure* expressions, that is, to expressions whose typing judgment carries the attribute P. Indeed, rules TLamTyped and RLamTyped allow the generalization of both row and type variables of a pure expression e. This solution is more general than the value restriction, because every value is a pure expression.

Subsumption relation on types $\boxed{D \vdash \tau \leq_T \tau}$

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

$$\textsc{TypeTrans}\quad \frac{D \vdash \tau \leq_T \tau' \qquad D \vdash \tau' \leq_T \tau''}{D \vdash \tau \leq_T \tau''} \qquad \textsc{Arrow}\quad \frac{D' = \langle \rho' \rangle \cup D \qquad a \leq_A a' \qquad D' \vdash \upsilon' \leq_T \upsilon \qquad D' \vdash \tau \leq_T \tau' \qquad D' \vdash_b \rho \leq_R \rho'}{D \vdash \upsilon \xrightarrow{\rho}_a \tau \leq_T \upsilon' \xrightarrow{\rho'}_{a'} \tau'}$$

Subsumption relation on signatures $\boxed{D \vdash \sigma \leq_S \sigma}$

$$\textsc{SigRefl}\quad D \vdash \sigma \leq_S \sigma \qquad \textsc{SigConc}\quad \frac{D \vdash \upsilon \leq_T \upsilon' \qquad D \vdash \tau' \leq_T \tau}{D \vdash (s : \upsilon \Rightarrow \tau) \leq_S (s : \upsilon' \Rightarrow \tau')}$$

Subsumption relation on rows $\boxed{D \vdash_b \rho \leq_R \rho}$

$$\textsc{Empty}\quad D \vdash_b \langle\rangle \leq_R \langle\rangle \qquad \textsc{RowCons}\quad D \vdash_b \rho \leq_R \sigma \cdot \rho \qquad \textsc{Skip}\quad \frac{D \vdash \sigma \leq_S \sigma' \qquad D \vdash_{\mathit{false}} \rho \leq_R \rho'}{D \vdash_b \sigma \cdot \rho \leq_R \sigma' \cdot \rho'}$$

$$\textsc{Swap}\quad D \vdash_b \sigma \cdot \sigma' \cdot \rho \leq_R \sigma' \cdot \sigma \cdot \rho$$

$$\textsc{Erase}\quad \frac{D \vdash s \notin \rho}{D \vdash_{\mathit{true}} (s : \mathtt{abs}) \cdot \rho \leq_R \rho} \qquad \textsc{RowTrans}\quad \frac{D \vdash_b \rho \leq_R \rho' \qquad D \vdash_b \rho' \leq_R \rho''}{D \vdash_b \rho \leq_R \rho''}$$

Fig. 5. Subsumption relations on types, signatures, and rows.

The only impure constructs are reading, writing, and allocating references, and allocation of effect labels. Therefore, even constructs such as handlers and effects can be considered pure expressions and can thus have their row and type variables generalized. Kammar and Pretnar study a similar system [24]. They show that, in the absence of references and allocation of effect labels, the unrestricted generalization of type variables is sound. The soundness of Tes subsumes this result. Finally, we remark that the reason why rule LetEffTyped carries an impure marker I is mainly technical: since the allocation of an effect label is implemented using memory allocation, such an expression interacts with the store. We believe that marking it as pure would not break the system's soundness. (Kammar and Pretnar [24] note on Paragraph 1 of the Conclusion Section that a proof of this statement remains an open problem.)

3.2.2 Overview of the main typing rules. We discuss the rules for allocating effect labels (LetEffTyped), performing effects (PerformTyped), and handling effects (HandleTyped).

Rule PerformTyped states that to perform an effect under the signature $(s : \upsilon \Rightarrow \tau)$, a program must produce a value of type υ, and, in return, it can expect a value of type τ. This typing rule confirms the intuitive idea that performing an effect is like calling a function of argument type υ and return type τ.

If we read rule LetEffTyped in the backwards direction (from the conclusion to the premise), then this rule states that allocating s gives e the ability to use s as an effect name. Initially, this name is attached to the absence signature. Because a row is supposed to be dynamically distinct, the introduction of $(s : \texttt{abs})$ in e's row is a nontrivial step, because we must prove that the extended row remains dynamically distinct: that if ρ satisfies the separation requirement, then so does the row $(s : \texttt{abs}) \cdot \rho$.

Rule HandleTyped, for installing a handler, expresses the idea that a handler establishes a boundary between the client e, which performs s effects according to σ, and the outer context in which the handler appears, where s effects abide by σ'.

Both the effect branch h and the return branch r can perform s effects according to σ'. Moreover, the continuation corresponds to h's second argument, whose type is $\tau \xrightarrow{\rho'}_a \kappa'$. It is interesting to remark that (1) the continuation has an arrow type, (2) this arrow is annotated by the row ρ', and (3) the return type of the continuation is the same as the one assigned to the handler. The first remark justifies that a delimited continuation can be seen as a function. The explanation for second and third remarks comes from the semantics of deep handlers: because the handler is reinstalled as the top frame of the continuation, s effects performed by the interrupted client are intercepted by the handler, who can introduce effects according to σ'.

3.2.3 Monotonicity rule and the subsumption relation. Rule MonotonicityTyped states the conditions under which one typing judgment subsumes another. These conditions are written in terms of subsumption relations on attributes, types, rows, and signatures. If two terms are related by a subsumption relation, then we say the term in the left-hand side is stronger than the one in the right-hand side.

The subsumption relation on attributes, noted $\_ \leq_A \_$, is the unique total order on *Attribute* where P is less than I. This relation can be defined as follows:

$$a \leq_A a' \triangleq (a = I \implies a' = I)$$

Figure 5 shows the definition of the remaining subsumption relations. Before we present these relations, let us introduce a notion on which they all depend: the notion of a *disjointness context*. A disjointness context D maps an effect name to a pair of a multiset of effect names S and a set of row variables V. Informally, a disjointness context D stores disjointness information. In particular, if D maps an effect name s to the pair (S, V), then the dynamic label bound by s is distinct from the dynamic labels bound by S and by V. We formally capture this description in Section 4, where we introduce the *semantic interpretation* of a disjointness context.

Why are disjointness contexts necessary? We introduce disjointness contexts to allow the sound *erasure of absence signatures* of a row. The erasure of an absence signature corresponds to the claim that $\sigma \cdot \rho \leq_R \rho$, where σ stands for $(s : \texttt{abs})$. However, this claim is not true in general. Indeed, if such subsumption relation was permitted, then the relation $\sigma \cdot \sigma \leq_R \sigma$ would be derivable. Intuitively, this relation says that a function f with row $\sigma \cdot \sigma$ can be seen as a function with row σ. However, the row $\sigma \cdot \sigma$ is not dynamically distinct, whereas the singleton row σ trivially is. Therefore, f has a false precondition, whereas a function with row σ does not. Assigning f the row σ would erase an unsatisfiable constraint, and thus lead to the acceptance of unsafe programs. An example of such an unsafe program is the following one:

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

Under the assumption that the relation $\sigma \cdot \sigma \leq_R \sigma$ is derivable, it is possible to show that this program is assigned the empty row and type `()`. The type derivation starts with the application of rule LetEffTyped, which introduces σ to the empty row. Then, the application of rule MonotonicityTyped, with the relation $\sigma \cdot \sigma \leq_R \sigma$ as the instance of the subsumption relation on rows, allows the type derivation of the program between lines 2–4 to be completed under the row $\sigma \cdot \sigma$. Because the row has two signatures for the same name s, we can install two handlers for the same effect s. The handler on line 2 allows its client – the program on line 3 – to perform s effects according to the signature (s : `()` ⇒ `()`). The handler on line 3 allows its client to perform s effects according to the signature (s : `bool` ⇒ `()`). Because the client of the handler on line 3 is also in the scope of the handler on line 2, it can perform s effects according to either one of these signatures. It then deviously chooses to perform an s effect according to the signature handled by the outermost handler. The innermost handler intercepts this effect and the mismatch of signatures leads to an error.

With disjointness contexts, the erasure of absence signatures can be soundly permitted. One can erase allow the erasure of s in (s : `abs`) $\cdot\ \rho$ under a disjointness context D, provided D guarantees that the dynamic label of s is different from the dynamic labels in ρ. We explain this idea in more detail when presenting the subsumption relation on rows.

Subsumption relation on types. The subsumption relation on types, noted $\_ \vdash \_ \leq_T \_$, is a relation parameterized by a disjointness context. The rules in Figure 5 state that this relation is reflexive and transitive and that it admits $\bot$ and $\top$ as bottom and top elements, respectively. Moreover, the relation is contravariant on the argument type of an arrow and covariant on the result type. The rule Arrow enriches the disjointness context. Intuitively, the rule exploits the assumption that ρ' is dynamically distinct to enrich the disjointness information stored in the current context D. The non-aliasing information learnt from the well-formedness of ρ' is represented by the disjointness context $\langle \rho' \rangle$. Its definition is written in terms of the functions *conc* and *abst*. The function *conc* computes the multiset of effect names of a row, whereas the function *abst* computes the set of row variables of a row. They are inductively defined as follows:

$$\begin{array}{rcl@{\qquad}rcl} \mathit{conc}(\langle\rangle) & \triangleq & \{\} & \mathit{abst}(\langle\rangle) & \triangleq & \{\} \\ \mathit{conc}((s : \_) \cdot \rho) & \triangleq & \{s\} \cup \mathit{conc}(\rho) & \mathit{abst}((s : \_) \cdot \rho) & \triangleq & \mathit{abst}(\rho) \\ \mathit{conc}(\theta \cdot \rho) & \triangleq & \mathit{conc}(\rho) & \mathit{abst}(\theta \cdot \rho) & \triangleq & \{\theta\} \cup \mathit{abst}(\rho) \end{array}$$

Notice that, because *conc* computes a multiset, the union, singleton set, and empty set have different meaning in each of the previous definitions. In the definition of *conc*, they are interpreted as multiset constructs, whereas, in the definition of *abst*, they are interpreted as set constructs.

The disjointness context $\langle \rho' \rangle$ can thus be defined as follows:

$$\langle \rho' \rangle \triangleq \bigcup_{s \in \mathit{conc}(\rho')} \{s \mapsto (\mathit{conc}(\rho') \setminus \{s\}, \mathit{abst}(\rho'))\}$$

The context $\langle \rho' \rangle$ maps every effect name s in ρ' to the pair of the multiset of effect names in ρ', excluding one occurrence of s, and the set of row variables in ρ'. The construction of this context exploits the assumption that ρ' is dynamically distinct: the dynamic label bound by s is distinct from the dynamic labels bound by $\mathit{conc}(\rho') \setminus \{s\}$ and by $\mathit{abst}(\rho')$.

To update a context D with $\langle \rho' \rangle$, it suffices to perform the *union* of these two contexts. The union of contexts D_1 and D_2 is defined as follows:

$$(D_1 \cup D_2)(s) \triangleq \begin{cases} D_1(s) \cup D_2(s) & \text{if } s \in \operatorname{dom} D_1 \cap \operatorname{dom} D_2 \\ D_i(s) & \text{if } s \in \operatorname{dom} D_i \\ \mathit{undefined} & \text{otherwise} \end{cases}$$

The union of pairs $D_1(s)$ and $D_2(s)$ is defined as the pairwise union.

Subsumption relation on signatures. The subsumption relation on signatures, noted $\_ \vdash \_ \leq_S \_$, is parameterized by a disjointness context. The rules in Figure 5 state that this relation is reflexive, and that, unlike an arrow constructor, the signature constructor $(\_ : \_ \Rightarrow \_)$ is covariant in its domain and contravariant in its range. (Transitivity is derivable.) The seemingly inverted variance in the case of a signature constructor matches the intuition that an effect signature $(s : v \Rightarrow \tau)$ is an arrow in the value context: a program that carries such a signature can perform s effects in the same way as it calls a function of type $v \to \tau$.

Subsumption relation on rows. The subsumption relation on rows, noted $\_ \vdash_\_ \_ \leq_R \_$, is parameterized by a disjointness context and a Boolean flag b. When true, this flag represents a permission to erase absence signatures. From the combination of rules in Figure 5, it follows that this relation is reflexive and transitive. Moreover, it follows that $b \vdash_\rho \rho' \leq_R$ is derivable for any row ρ' that includes ρ, regardless of the order of the signatures in ρ'. In particular, ρ is stronger than any of its permutations. The ability to freely permute entries in a row is present in systems that impose the restriction to lexically scoped handlers [7, 44], and in Links's type system [19], which, by means of a syntactic criteria, does not allow repeated occurrences of a label in a row. The systems studied in [5, 6, 29], on the other hand, do not support this feature.

Rule Erase depends on the following assertion:

$$D \vdash s \notin \rho \triangleq s \in \mathrm{dom}\, D \wedge (conc(\rho), abst(\rho)) \subseteq D(s)$$

This assertion claims that, from the disjointness information stored in D, one can derive that the label bound by s is distinct from the labels in ρ. Rule Erase also asks for the flag to be true: one must have the permission to erase signatures. This permission is lost when comparing rows of the form $\sigma \cdot \rho$ and $\sigma' \cdot \rho'$ through rule Skip. Intuitively, this restriction is necessary, because otherwise one would be able to erase a signature s from ρ without checking that the labels bound by s and σ are distinct. We omit the technical arguments for the sake of space, but it is possible to exhibit an unsafe program if we consider the subsumption rules without flags.

3.3 Examples

Now, we present a number of examples that illustrate the subtleties of the system: how to exploit the implicit assumption that rows are dynamically distinct in order to restrict the effects a program can perform, and how to exploit the subsumption rules.

3.3.1 Filter. As the first example, let us consider the function *filter* from Section 1. The goal is to show that *filter* is a well-typed program in Tes and to understand what restrictions are enforced by its type. In particular, we want to require that, during any application *filter xs f*, the effects performed by f do not include *yield* effects. Therefore, a *yield* handler will not accidentally intercept effects performed by f.

Recall the definition of *filter* (Eq. 3), which applies the predicate f to each element of *xs*, and *yields* those elements for which f returns `true`:

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

The definition depends on the higher-order iteration method *iter* (Eq. 6), which applies a user-provided function to each element of a list. Tes accepts *iter* and assigns to it the type

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

Tes also accepts *filter* and assigns to it the following type:

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

Since the body of *filter* is defined as an application of *iter*, to type check *filter*, it suffices to show that the arguments of this application respect *iter*'s type. The only nontrivial step in this task is to show that g has type $\alpha \xrightarrow{Y(\alpha)\cdot\theta} ()$, where α and θ are the variables quantified by *filter*'s type and that are introduced in the first steps of type checking, and where $Y(\alpha)$ is an abbreviation for the signature $(\mathit{yield} : \alpha \Rightarrow ())$. This step is nontrivial because f's row θ does not exactly match the row $Y(\alpha)\cdot\theta$ under which f is called and which must include the signature $Y(\alpha)$ since g might perform this effect. But this step is not a problem for Tes, it suffices to apply rule MonotonicityTyped, which allows the extension of rows: in particular, the relation $D \vdash_b \theta \leq_R Y(\alpha)\cdot\theta$ is derivable under any context D and flag b.

What does filter's type mean? The type of *filter* tells that *filter* behaves independently both on the representation of the list elements and on the set of effects that f might perform. Moreover, the row $Y(\alpha)\cdot\rho$ tells that the program *filter xs f* performs either *yield* effects or θ effects (introduced by f). Finally, *filter's type interdicts f from performing yield effects.* At first, it might seem strange that *filter*'s type is polymorphic on θ: what stops one from specializing θ to the singleton row $Y(\alpha)$, thus allowing f to perform *yield*? The answer is: nothing. However, such a specialized version of *filter* would be useless. Implicitly, the type of *filter* imposes the requirement that the row $Y(\alpha)\cdot\theta$ is dynamically distinct. This requirement appears when *filter* is fully applied. So, although one could specialize *filter* with the row $Y(\alpha)$, one would not be able to invoke this version of *filter* because the separation requirement for such a row would not hold.

3.3.2 Counter. In Section 1, we introduced the function *counter* (Eq. 2), which receives a second-order function *ff* as an argument and produces a version of this function that counts the number of times *ff* calls its argument function f. Type-checking *counter* depends on the guarantee that the effects performed by f are not intercepted by the handler installed by *counter*. Although, this behavior is clear from an operational-semantics point of view – the handler targets a fresh dynamic label – to statically ensure it is challenging. Previous type systems that accepted this program [7, 44] were designed for languages restricted to lexically scoped handlers. Therefore, type-safety of *counter* in such systems is a weaker result than the one obtained in Tes, because the language considered in such systems is less expressive than λ-*labels*, which supports traditional handlers and the unrestricted allocation of effect labels. We believe Tes is the only system that supports such a language and accepts *counter*. The closest related previous work of which we are aware is the system λ^{HEL}, devised by Biernacki et al. [6]. They consider a calculus that also supports unrestricted allocation of effect labels, and where the function *counter* could be similarly defined. However, in λ^{HEL}, the function *counter* is ill-typed. To satisfy the type checker, the programmer would have to place a *lift* coercion around the function call $f\ ()$. [1]

In Tes, the program *counter* can be assigned the following type:

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

This type means that *counter* works with any effect-polymorphic second-order function *ff* and that *counter ff* produces a function whose type is similar to *ff*'s type – the only difference is the return type $\gamma * \texttt{int}$, which is the product of *ff*'s return type and the type of integers. In particular, like the function *ff*, the result of *counter ff* is effect-polymorphic: it can be further applied to a function f regardless of the effects performed by this function.

3.3.3 Lexically scoped handlers. As the last example, let us consider the typing rule for lexically scoped handlers. This example illustrates an interesting application of the subsumption rule to erase an absence signature. Recall the definition of a lexically scoped handler (Eq. 1) in λ-*labels* as

[1] This claim was confirmed by the authors via personal communication.

a program that (1) generates a fresh effect label bound by s, and (2) installs a s handler over the application of e to a function that performs this effect:

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

Tes admits the following derived typing rule for this construct:

LexHandleTyped

$$\frac{\Xi \mid \Delta \mid \Gamma \vdash_a e : \rho : \forall \theta.\ (\upsilon \xrightarrow{\theta}_a \tau) \xrightarrow{\theta \cdot \rho}_a \kappa \qquad s \notin \Gamma, \rho, \upsilon, \tau, \kappa, \kappa' \qquad \Xi \mid \Delta \mid \Gamma \vdash_a h : \rho : \upsilon \to_a (\tau \xrightarrow{\rho}_a \kappa') \xrightarrow{\rho}_a \kappa' \qquad \Xi \mid \Delta \mid \Gamma \vdash_a r : \rho : \kappa \xrightarrow{\rho}_a \kappa'}{\Xi \mid \Delta \mid \Gamma \vdash_I \texttt{lex-handle}_s\ e\ \texttt{with}\ (h \mid r) : \rho : \kappa'}$$

(This rule is not original; it is similar to the rule for handlers presented in Figure 3 of the paper [7].) The expression e must be polymorphic in the effect θ: it must work independently of the label that is bound by s. The first step in the derivation of this rule consists of the application of rule LetEffTyped. This application is the reason that the conclusion of rule LexHandleTyped is marked as impure, and it is also the origin of the freshness condition on s that one sees in this same rule. It is unpleasant to have such a condition polluting the rule, but this condition can be easily satisfied, provided one is willing to rename s to a fresh s'. The second (and last) step in the derivation of LexHandleTyped is the application of rule HandleTyped. The main step to dispatch the premises of rule HandleTyped is to prove that one can introduce s in h's type:

$$\frac{\rho' = (s : \mathsf{abs}) \cdot \rho \qquad \Xi \mid \Delta \mid \Gamma \vdash_a h : \rho : \upsilon \to_a (\tau \xrightarrow{\rho}_a \kappa') \xrightarrow{\rho}_a \kappa'}{\Xi \mid \Delta \mid \Gamma \vdash_a h : \rho : \upsilon \to_a (\tau \xrightarrow{\rho'}_a \kappa') \xrightarrow{\rho'}_a \kappa'}$$

To establish this implication, we apply rule MonotonicityTyped. Then, it suffices to show the following chain of subsumption relations:

$$\upsilon \to_a (\tau \xrightarrow{\rho}_a \kappa') \xrightarrow{\rho}_a \kappa' \leq_T \upsilon \to_a (\tau \xrightarrow{\rho}_a \kappa') \xrightarrow{\rho'}_a \kappa' \leq_T \upsilon \to_a (\tau \xrightarrow{\rho'}_a \kappa') \xrightarrow{\rho'}_a \kappa'$$

The first step follows trivially from the ability to extend rows – by rule RowCons, the relation $\vdash_b \rho \leq_R \rho'$ holds for any b – and the second step follows from the ability to erase absence signatures – by rule Erase, the relation $\langle \rho' \rangle \vdash_{\mathit{true}} \rho' \leq_R \rho$ holds, and, since the right-most arrow includes the row ρ', we can satisfy the separation requirement stored in $\langle \rho' \rangle$.

4 METATHEORY

In this section, we prove strong type soundness of Tes: if a complete program is accepted by the system, then this program can be safely executed; in particular, no effect is left unhandled. To prove this statement, we interpret Tes typing judgments as specifications written in Hazel+, a novel Separation Logic [34] for reasoning about *λ-labels* programs. This approach to prove the soundness of a type system is known as the *semantic approach*[1, 26, 27]. This section is structured as follows: first, we present Hazel+; second, we introduce the interpretation of typing judgments; finally, we state and prove the soundness of Tes.

4.1 The Hazel+ Logic

A program logic is a pair of a specification language, to describe the behavior of programs, and a set of rules, to derive such specifications. Separation Logic is a program logic whose specification language features a separating conjunction. This logical connective describes states in which the memory is partitioned. Such a feature is particularly appealing when reasoning about programs that manipulate mutable data structures. This is the case for *λ-labels* programs, because the language has support for general references. Therefore, we choose to build Hazel+ as a Separation Logic.

Weakest precondition $\boxed{wp_a\ e\ \langle E\rangle\{\Phi\}}$

$$wp_a\ e\ \langle E\rangle\{\Phi\} \triangleq \mathit{ValidDistinct}\, E.1 \mathrel{-\!\!*} ewp_a\ e\ \langle E\rangle\{\Phi\}$$

Basic weakest precondition $\boxed{ewp_a\ e\ \langle E\rangle\{\Phi\}}$

$$ewp_a\ v\ \langle E\rangle\{\Phi\} \triangleq \mathit{if}\ a = P\ \mathit{then}\ \Phi(v)\ \mathit{else}\ \Rightarrow\!\!\!\!\!\!\cdot\ \Phi(v)$$

$$ewp_a\ (\texttt{eff}\ \ell\ v\ K)\ \langle E\rangle\{\Phi\} \triangleq \exists \Psi.\ (\ell, \Psi) \in E\ *\ \Psi\ \mathit{allows}\ \texttt{eff}\ v\ \{w.\ \triangleright\, ewp_a\ K[w]\ \langle E\rangle\{\Phi\})\}$$

$$ewp_I\ e\ \langle E\rangle\{\Phi\} \triangleq \forall \sigma.\, S(\sigma)\ {}^{\top}\!\Rrightarrow^{\emptyset} \begin{cases} \exists e', \sigma'.\ e\ /\ \sigma \longrightarrow e'\ /\ \sigma'\ * \\ \forall e', \sigma'.\ e\ /\ \sigma \longrightarrow e'\ /\ \sigma'\ {}^{\emptyset}\!\Rrightarrow^{\emptyset} \triangleright\, {}^{\emptyset}\!\Rightarrow^{\top} \\ \qquad S(\sigma')\ *\ ewp_I\ e'\ \langle E\rangle\{\Phi\} \end{cases}$$

$$ewp_P\ e\ \langle E\rangle\{\Phi\} \triangleq \forall \sigma.\, \begin{cases} \exists e'.\ e\ /\ \sigma \longrightarrow e'\ /\ \sigma\ * \\ \forall e'.\ e\ /\ \sigma \longrightarrow e'\ /\ \sigma \mathrel{-\!\!*} \triangleright\, ewp_P\ e'\ \langle E\rangle\{\Phi\} \end{cases}$$

Protocol agreement $\boxed{\Psi\ \mathit{allows}\ \texttt{eff}\ v\ \{\Phi\}}$

$$\Psi\ \mathit{allows}\ \texttt{eff}\ v\ \{\Phi\} \triangleq \exists Q.\ \Psi\, v\, Q\ *\ \Box \forall w.\, Q(w) \mathrel{-\!\!*} \Phi(w)$$

Fig. 6. Definition of *wp*, *ewp*, and *allows* `eff`.

Value
$$\frac{\Phi(v)}{wp_a\ v\ \langle E\rangle\{\Phi\}}$$

Monotonicity
$$\frac{a \preceq_A a' \qquad \Box \forall w.\, \Phi(w) \mathrel{-\!\!*} \Phi'(w) \qquad wp_a\ e\ \langle E\rangle\{\Phi\} \qquad E \preceq_L E'}{wp_{a'}\ e\ \langle E'\rangle\{\Phi'\}}$$

Bind
$$\frac{K\ \mathit{is\ neutral} \qquad wp_a\ e\ \langle E\rangle\{v.\ wp_a\ K[v]\ \langle E\rangle\{\Phi\}\}}{wp_a\ K[e]\ \langle E\rangle\{\Phi\}}$$

Perform
$$\frac{(\ell, \Psi) \in E \qquad \Psi\ \mathit{allows}\ \texttt{eff}\ v\ \{\Phi\}}{wp_a\ (\texttt{perform}\ \ell\ v)\ \langle E\rangle\{\Phi\}}$$

LetEff
$$\frac{\forall \ell.\ wp_a\ (e\,[\ell/s])\ \langle(\ell, \bot) :: E\rangle\{\Phi\}}{wp_I\ (\texttt{effect}\ s\ \texttt{in}\ e)\ \langle E'\rangle\{\Phi'\}}$$

Handle
$$\frac{\mathit{Handler}_a\ \langle\Psi\rangle\{\Phi\}\ (h \mid r)\ \langle E'\rangle\{\Phi'\} \qquad E' = (\ell, \Psi') :: E \qquad wp_a\ e\ \langle(\ell, \Psi) :: E\rangle\{\Phi\}}{wp_a\ (\texttt{handle}\ e\ \texttt{with}\ (\ell\ \texttt{:}\ h \mid r))\ \langle E'\rangle\{\Phi'\}}$$

Pure Infinitary Conjunction
$$\frac{E\ \mathit{is\ pure} \qquad \Box \forall x.\ wp_P\ e\ \langle E\rangle\{y.\ \Phi(x, y)\}}{wp_a\ e\ \langle E\rangle\{y. \forall x.\, \Phi(x, y)\}}$$

Fig. 7. Selected reasoning rules

More specifically, we build Hazel+ as an extension of Iris [22], an expressive Separation Logic, and borrowing many ideas from Hazel [14], a Separation Logic for effect handlers, unnamed effects, and one-shot continuations.

A brief introduction to Iris. In the interest of space, we give a concise explanation of the Iris logic: we explain the notions that are relevant to this paper. Birkedal and Bizjak [8] and Jung et al. [22] give a thorough introduction to Iris. Here is the syntax of a subset of Iris assertions:

$$\mathit{iProp} \ni P ::= P\ *\ P \mid P \mathrel{-\!\!*} P \mid P \wedge P \mid \Box P \mid \triangleright P \mid \Rightarrow\!\!\!\!\!\!\cdot\ P \mid \boxed{P}^{\iota} \mid \ell \mapsto v \mid \ell \mapsto_{\Box} v \mid \ldots$$

An Iris assertion holds relatively to a heap. The separating conjunction $P * Q$ holds of heaps composed of two disjoint parts, one satisfying P and one satisfying Q. The points-to assertion $\ell \mapsto v$ holds of heaps where the location ℓ stores v. The *persistent points-to assertion* [41] $\ell \mapsto_{\square} v$ holds of heaps where ℓ is a read-only location storing v. The *persistence modality* $\square$ is used to describe immutable regions of the heap. In particular, if the assertion $\square P$ holds, then P can be *duplicated*: $P * P$ holds. Moreover, this modality can be used to introduce the notion of a *persistent* assertion P: whenever P holds, the assertion $\square P$ holds. The *later modality* $\triangleright$ is used to construct recursive definitions. The *update modality* $\Rrightarrow$ is used to describe heap updates, such as the allocation of references, or writes to a memory location.

4.1.1 Specification language. The main ingredient of the specification language of Hazel+ is the weakest precondition $wp_a\ e\ \langle E \rangle \{\Phi\}$. Intuitively, if this assertion holds, then e can be safely executed. The purity attribute a indicates whether e interacts with the heap during its execution. The program e can either terminate with output v, in a state where $\Phi(v)$ holds, or perform an effect according to the *protocol list* E, which is a list of pairs of an effect label and a *protocol*. Intuitively, a protocol, noted Ψ, maps a value v to a set of answers Q that a program can expect in exchange for performing an effect with payload v. To formally capture this relation between a payload and a set of answers, we define protocols as inhabitants of the type $Protocol \triangleq Val \to (Val \to iProp) \to iProp$. We can thus say that Q is *a set of answers to* v *according to* Ψ if the assertion $\Psi\ v\ Q$ holds. The protocol list E then maps each effect label ℓ to a protocol capturing this payload-answer relation when performing effects labeled ℓ.

Figure 6 shows the definition of *wp*, which is written in terms of the *basic weakest precondition ewp* and the *protocol agreement allows* `eff`. The basic weakest precondition is (recursively) defined by case distinction on a and e. (Recursive calls are guarded by a later modality, thus ensuring that *ewp* is well-defined.) When a is P, neither the update modality nor the *state interpretation* S [2] As for the the case distinction on e, we have to consider the three following cases:

(1) *Value.* When e is a value v, this value must satisfy the postcondition Φ.
(2) *Effect.* When e is an active effect $\texttt{eff}\ \ell\ v\ K$, the label ℓ must be associated to a protocol Ψ, such that, for every possible answer w ascribed by Ψ to the request v, it is safe to resume K with w. This property is stated in terms of the protocol agreement: Ψ *allows* $\texttt{eff}\ v\ \{\Phi\}$ is the claim that Ψ assigns v to a set of answers included in Φ. The persistence modality in the definition of the protocol agreement ensures that a captured continuation can be resumed multiple times.
(3) *Reducible expression.* When e is neither a value nor an effect, then e must be a reducible expression and every possible reduction e' must be safe.

The weakest precondition *wp* is defined on top of *ewp* by adding the assumption that the list of labels in E is *valid and distinct*: these labels have been allocated and there is no aliasing among them. We write $E.1$ for the list of effect labels in E, that is, the projection of the first component of every pair in E. The valid-and-distinct property is captured by the predicate $ValidDistinct$:

$$ValidDistinct\ L \triangleq NoDup\ L \wedge (\forall \ell \in L.\ \ell \mapsto_{\square} ())$$

The assertion $NoDup\ L$ captures the non-aliasing claim: it states that there are no duplicates in the list L. The assertion $\ell \mapsto_{\square} ()$ is the claim that ℓ has been allocated. Because both of these assertions are persistent, so is the assertion $ValidDistinct\ L$. This is a crucial property, because, otherwise, the valid-and-distinct property would not be necessarily preserved during multiple invocations of a multi-shot continuation.

[2]The state interpretation is an invariant about the store. Its definition is identical to Jung et al.'s [22].

4.1.2 Reasoning rules. Figure 7 shows a subset of the reasoning rules of Hazel+. These are the most relevant rules to the soundness proof of Tes. We briefly discuss each one of them. We say that a reasoning rule *justifies* a typing rule to mean that this is the key rule to prove that the typing rule preserves the semantic interpretation of judgments.

Rule Value expresses the idea that a program can terminate by returning a value v that satisfies the postcondition. This rule justifies the typing rule ValueTyped.

Rule Monotonicity expresses the idea that, if a program e satisfies the specification $ewp_a\ e\ \langle E\rangle\{\Phi\}$, then it also satisfies a specification $ewp_{a'}\ e\ \langle E'\rangle\{\Phi'\}$, where where a', E', and Φ' are weaker than than a, E, and Φ, respectively. The predicate Φ' is weaker than Φ if $\Phi(v)$ implies $\Phi'(v)$ for every v. The premise of rule Monotonicity stating this implication is covered by a persistence modality, because, in the presence of multi-shot continuations, a program may terminate multiple times. This persistence modality is the reason why *the frame rule does not hold in Hazel+*; only a restricted version holds: one can apply the frame rule under the empty protocol list []. Intuitively, the empty protocol list states the absence of effects, and, under this restriction, a program can never be part of the context captured by a multi-shot continuation. The frame rule is also known to hold if continuations are one-shot [14]. The protocol list E' is weaker than E, noted $E \preceq_L E'$, if $E.1$ is valid and distinct, under the assumption that $E'.1$ is valid and distinct, and if for every Ψ in E there is a protocol Ψ' in E' that describes the same effect ℓ as Ψ and that is weaker than Ψ:

$$E \preceq_L E' \triangleq \begin{array}{l}(\mathit{ValidDistinct}\ E'.1 \mathrel{-\!\!*} \mathit{ValidDistinct}\ E.1) \wedge \\ (\square \forall (\ell, \Psi) \in E, v, \Phi.\ \Psi\ \mathit{allows}\ \texttt{eff}\ v\ \{\Phi\} \mathrel{-\!\!*} \exists \Psi'.\ (\ell, \Psi') \in E' \wedge \Psi'\ \mathit{allows}\ \texttt{eff}\ v\ \{\Phi\})\end{array}$$

The property of Ψ' being weaker than Ψ is the assertion that every pair of a request v and a predicate Φ that is allowed by Ψ is also allowed by Ψ'.

Rule Bind allows *context-local reasoning*: one can reason about the execution of a program independently of its surrounding evaluation context. This rule is surprising given the non-local nature of effects and handlers. There is however a side condition: the context K must be *neutral*, that is, K cannot contain a frame of the form $\texttt{handle \_ with } (\ell : h \mid r)$.

To reason about non-neutral contexts, one can employ rule Handle. This rule expresses the idea that to install a handler for a label ℓ around a program e, it suffices to know by which protocol e abides when performing effects labeled ℓ. Indeed, the rule asks for the verification of e according to a protocol list that associates ℓ to Ψ. The verification of r and h is delegated to the *handler judgment*, noted *Handler*, which compresses the specifications of r and h into a single assertion:

$$\begin{array}{ll}\mathit{Handler}_a\ \langle\Psi\rangle\{\Phi\}\ (h \mid r)\ \langle E'\rangle\{\Phi'\} \triangleq & \\ \quad \textit{(Return branch)} & (\square \forall v.\ \Phi(v) \mathrel{-\!\!*} wp_a\ (r\ v)\ \langle E'\rangle\{\Phi'\}) \quad \wedge \\ \quad \textit{(Effect branch)} & (\square \forall v, k.\Psi\ \mathit{allows}\ \texttt{eff}\ v\ \{w.\ wp_a\ (k\ w)\ \langle E'\rangle\{\Phi'\}\} \mathrel{-\!\!*} wp_a\ (h\ v\ k)\ \langle E'\rangle\{\Phi'\})\end{array}$$

Indeed, the handler judgment is the conjunction of r's and h's specifications. The specification of r assumes that v satisfies e's postcondition and claims that r satisfies the postcondition Φ' and protocol list E'. The specification of h requires h to satisfy postcondition Φ' and protocol list E' under two assumptions: (1) that v corresponds to the payload of a effect labeled ℓ performed by a program that abides by Ψ, and (2) that k represents this suspended program.

Rule Perform tells that one can reason about performing an effect as calling a function. A protocol Ψ dictates which answer a client can expect in exchange for a value v in a similar way as a program specification dictates which result one can expect from a function call.

Rule LetEff justifies the typing rule LetEffTyped. The protocol $\bot$, to which the label ℓ is initially associated, is the *empty protocol*. It is defined as $\lambda\_\_.$ False. A program that abides by this protocol performs no effect, because the assertion $\bot\ \mathit{allows}\ \texttt{eff \_ \{\_\}}$ never holds. The proof of rule LetEff is essentially the proof that, after the allocation of a fresh effect label, the list $E' = (\ell, \bot) :: E$ is

valid and distinct, given that E is valid and distinct. The key step in this proof is the introduction of a full points-to assertion $\ell \mapsto ()$, which we exploit to prove that ℓ is not an alias of a previously allocated label, and which we update to a persistent points-to assertion $\ell \mapsto_{\square} ()$ to complete the proof that E' is valid and distinct.

Rule Pure Infinitary Conjunction justifies the rules for introducing polymorphic types, namely rules TLamTyped and RLamTyped. This rule is atypical, because the unrestricted infinitary conjunction rule does not hold in Separation Logic [33]. From a technical point of view, the unrestricted infinitary conjunction rule does not hold in the particular case of Iris, because an universal quantification does not commute with the update modality. Here, we are able to prove a restricted version of the infinitary conjunction rule: the premise includes the attribute P, which excludes the update modality from the definition of wp_P. Another restriction is that the protocol list E must be *pure*. The list E is pure, if every protocol in E is *pure*. A protocol Ψ is pure if the following assertion holds:

$$\forall v.\ \exists Q.\ \forall Q'.\ \Psi\, v\, Q' \mathrel{-\!\!*} (\Psi\, v\, Q \ *\ (\forall w.\ Q\, w \mathrel{-\!\!*} \square\, Q'\, w))$$

This requirement appears as a solution for a step in the proof of Pure Infinitary Conjunction, where we need to commute an existential quantifier with an universal quantifier.

4.1.3 Soundness. The *adequacy theorem* gives the formal meaning of a weakest precondition:

Theorem 4.1 (Adequacy). *If $wp_a\ e\ \langle [\,] \rangle \{\Phi\}$ holds then e is safe.*

If one verifies e by proving a weakest precondition statement with the empty protocol list and an arbitrary postcondition Φ, then e is *safe*: the execution of e either diverges or terminates with a value. In particular, the execution does not crash, nor does it perform an unhandled effect.

4.2 Semantic Interpretation

4.2.1 Semantic interpretation of typing judgments. The semantic interpretation of judgments translates Tes typing judgments to Hazel+ specifications: it maps $\Xi \mid \Delta \mid \Gamma \vdash_a e : \rho : \tau$ to a *semantic judgment* $\Xi \mid \Delta \mid \Gamma \vDash_a e : \rho : \tau$. The definition of a semantic judgment is written in terms of the *semantic interpretation of types, signatures, and rows*. Types are interpreted as *semantic types* that inhabit $\mathit{SemType} \triangleq \{P : \mathit{Val} \to \mathit{iProp} \mid \forall v.\ P(v)\ \mathit{is\ persistent}\}$. Therefore, semantic types are *persistent predicates*, which can be seen as sets of values. The persistence requirement reflects the lack of a substructural discipline in Tes, that is, Tes is not an affine type system. Signatures are interpreted as *semantic signatures* that inhabit $\mathit{SemSig} \triangleq \{E : \mathit{List}\,(\mathit{Loc} \times \mathit{Protocol}) \mid E\ \mathit{is\ pure}\}$. Therefore, semantic signatures are pure protocol lists. Rows are interpreted as *semantic rows* that inhabit *SemRow*, which coincide with semantic signatures, that is, semantic rows are also pure protocol lists.

The interpretation of types, signatures, and rows appears in Figure 8. It has two parameters: (1) a row- and type-variable map η, which maps row variables to semantic rows and type variables to semantic types; and (2) an effect-name map δ, which maps effect names s to effect labels.

The interpretation of types follows the standard approach for the definition of *unary logical relations* in Iris [26]. The only case where our interpretation deviates from previous works is in the interpretation of arrow types. The interpretation of an arrow type $v \xrightarrow{\rho}_a \tau$ contains values v that satisfy the specification $\square \forall w.\, \mathcal{V}[\![v]\!]^{\delta}_{\eta}(w) \mathrel{-\!\!*} wp_a\ (v\ w)\ \langle \mathcal{R}[\![\rho]\!]^{\delta}_{\eta} \rangle \{y.\mathcal{V}[\![\tau]\!]^{\delta}_{\eta}(y)\}$. This specification states that v produces values in the interpretation of τ when applied to a value in the interpretation of v. The novel aspect of this interpretation is the use of our notion of the weakest precondition, wp, which describes the effects that v might perform. The description of these effects is given by the interpretation of the row ρ: it is the list concatenation of the interpretation of the signatures in ρ.

Interpretation of types. $\boxed{\mathcal{V}[\![\tau]\!]^\delta_\eta : \mathit{SemType}}$

$$
\begin{array}{rcl}
\mathcal{V}[\![\bot]\!]^\delta_\eta(\_) & \triangleq & \mathsf{False} \\
\mathcal{V}[\![\top]\!]^\delta_\eta(\_) & \triangleq & \mathsf{True} \\
\mathcal{V}[\![()]\!]^\delta_\eta(v) & \triangleq & (v = ()) \\
\mathcal{V}[\![\texttt{bool}]\!]^\delta_\eta(v). & \triangleq & \exists b.\, v = b \\
\mathcal{V}[\![\texttt{int}]\!]^\delta_\eta(v) & \triangleq & \exists i.\, v = i \\
\mathcal{V}[\![\alpha]\!]^\delta_\eta(v) & \triangleq & \eta(\alpha)(v) \\
\mathcal{V}[\![\tau\ \texttt{list}]\!]^\delta_\eta(v) & \triangleq & v = [] \lor \exists u, us.\, v = u :: us * \mathcal{V}[\![\tau]\!]^\delta_\eta(u) * \triangleright \mathcal{V}[\![\tau\ \texttt{list}]\!]^\delta_\eta(us) \\
\mathcal{V}[\![v * \tau]\!]^\delta_\eta(v) & \triangleq & \exists w, w'.\, v = (w, w') * \mathcal{V}[\![v]\!]^\delta_\eta(w) * \mathcal{V}[\![\tau]\!]^\delta_\eta(w') \\
\mathcal{V}[\![v + \tau]\!]^\delta_\eta(v) & \triangleq & \exists w.\, (v = \mathtt{inj}_1\ w * \mathcal{V}[\![v]\!]^\delta_\eta(w)) \lor (v = \mathtt{inj}_2\ w * \mathcal{V}[\![\tau]\!]^\delta_\eta(w')) \\
\mathcal{V}[\![\tau\ \texttt{ref}]\!]^\delta_\eta(v) & \triangleq & \exists \ell.\, v = \ell * \boxed{\exists w.\, \ell \mapsto w * \mathcal{V}[\![\tau]\!]^\delta_\eta(w)}^{\mathcal{N}.\ell} \\
\mathcal{V}[\![v \xrightarrow{\rho}_a \tau]\!]^\delta_\eta(v) & \triangleq & \Box \forall w.\, \mathcal{V}[\![v]\!]^\delta_\eta(w) \mathrel{-\!\!*} wp_a\ (v\ w)\ \langle \mathcal{R}[\![\rho]\!]^\delta_\eta \rangle \{y.\mathcal{V}[\![\tau]\!]^\delta_\eta(y)\} \\
\mathcal{V}[\![\forall \alpha.\ \tau]\!]^\delta_\eta(v) & \triangleq & \forall A.\, \mathcal{V}[\![\tau]\!]^\delta_{\eta, \alpha \mapsto A}(v) \\
\mathcal{V}[\![\forall \theta.\ \tau]\!]^\delta_\eta(v) & \triangleq & \forall E.\, \mathcal{V}[\![\tau]\!]^\delta_{\eta, \theta \mapsto E}(v)
\end{array}
$$

Interpretation of rows. $\boxed{\mathcal{R}[\![\rho]\!]^\delta_\eta : \mathit{SemRow}}$

$$\mathcal{R}[\![\rho]\!]^\delta_\eta \quad \triangleq \quad \bigcup_{\sigma \in \rho} \mathcal{S}[\![\sigma]\!]^\delta_\eta$$

Interpretation of signatures. $\boxed{\mathcal{S}[\![\sigma]\!]^\delta_\eta : \mathit{SemSig}}$

$$
\begin{array}{rcl}
\mathcal{S}[\![(s : v \Rightarrow \tau)]\!]^\delta_\eta & \triangleq & (\delta(s),\ \lambda v\, Q.\, \mathcal{V}[\![v]\!]^\delta_\eta(v) * \Box \forall w.\, \mathcal{V}[\![\tau]\!]^\delta_\eta(w) \mathrel{-\!\!*} Q(w)) \\
\mathcal{S}[\![\theta]\!]^\delta_\eta & \triangleq & \eta(\theta)
\end{array}
$$

Interpretation of typing judgments. $\boxed{\Xi \mid \Delta \mid \Gamma \vDash_a e : \rho : \tau}$

$$
\begin{array}{l}
\Xi \mid \Delta \mid \Gamma \vDash_a e : \rho : \tau \triangleq \\
\quad \forall \eta, \delta, vs.\, (\forall \{x \mapsto v\} \subseteq \Gamma.\, \mathcal{V}[\![v]\!]^\delta_\eta(vs(x))) \mathrel{-\!\!*} wp_a\ (e[vs][\delta])\ \langle \mathcal{R}[\![\rho]\!]^\delta_\eta \rangle \{y.\, \mathcal{V}[\![\tau]\!]^\delta_\eta(y)\}
\end{array}
$$

Fig. 8. Interpretation of types, rows, signatures, and typing judgments.

A row variable θ is interpreted as the semantic row $\eta(\theta)$, and a concrete signature $(s : v' \Rightarrow \tau')$ is interpreted as the (singleton list containing the) pair of the dynamic label to which s is bound and the protocol dictating that the answer to a value in the interpretation of v' is a value in the interpretation of τ'. Therefore, v's specification states that v performs effects according to the interpretation of ρ. Moreover, the implicit valid-and-distinct property in the definition of *wp* unfolds as the assertion $\mathit{ValidDistinct}\, \mathcal{R}[\![\rho]\!]^\delta_\eta.1$. This assertion means that the list of dynamic labels bound by the effect names in ρ is valid and distinct, or, using the terminology introduced in Section 3, that ρ is dynamically distinct.

The meaning of the semantic judgment $\Xi \mid \Delta \mid \Gamma \vDash_a e : \rho : \tau$ can be given as follows: for every row- and type-variable map η, effect-name map δ, and for every substitution vs of variables x in dom Γ with values in the interpretation of $\Gamma(x)$, the complete program $e[vs][\delta]$ produces a value in the semantic type $\mathcal{V}[\![\tau]\!]^\delta_\eta$ and performs effects according to the semantic row $\mathcal{R}[\![\rho]\!]^\delta_\eta$.

Interpretation of disjointness contexts. $\boxed{[\![D]\!]^{\delta}_{\eta} : \mathit{iProp}}$

$$[\![D]\!]^{\delta}_{\eta} \triangleq (\forall s \in \mathrm{dom}\ D.\ \delta(s) \mapsto_{\square} ()) \wedge (\forall \{s \mapsto (S, V)\} \subseteq D.\ \delta(s) \notin \delta(S) \wedge \delta(s) \notin \eta(V).1)$$

Interpretation of the subsumption relation on types. $\boxed{D \vDash \tau \leq_T \tau}$

$$D \vDash v \leq_T \tau \triangleq \square \forall \eta, \delta.\ [\![D]\!]^{\delta}_{\eta} \mathrel{-\!\!*} \forall v.\ \mathcal{V}[\![v]\!]^{\delta}_{\eta}(v) \mathrel{-\!\!*} \mathcal{V}[\![\tau]\!]^{\delta}_{\eta}(v)$$

Interpretation of the subsumption relation on rows. $\boxed{D \vDash_b \rho \leq_R \rho}$

$$D \vDash_b \rho \leq_R \rho' \triangleq \square \forall \eta, \delta.\ [\![D]\!]^{\delta}_{\eta} \mathrel{-\!\!*} (\mathcal{R}[\![\rho]\!]^{\delta}_{\eta} \leq_L \mathcal{R}[\![\rho']\!]^{\delta}_{\eta} \wedge (b = \mathit{false} \mathrel{-\!\!*} \mathcal{R}[\![\rho]\!]^{\delta}_{\eta}.1 \subseteq_m \mathcal{R}[\![\rho']\!]^{\delta}_{\eta}.1))$$

Interpretation of the subsumption relation on signatures. $\boxed{D \vDash \sigma \leq_S \sigma'}$

$$D \vDash \sigma \leq_S \sigma' \triangleq \square \forall \eta, \delta.\ [\![D]\!]^{\delta}_{\eta} \mathrel{-\!\!*} (\mathcal{S}[\![\sigma]\!]^{\delta}_{\eta} \leq_L \mathcal{S}[\![\sigma']\!]^{\delta}_{\eta} \wedge \mathcal{S}[\![\sigma]\!]^{\delta}_{\eta}.1 = \mathcal{S}[\![\sigma']\!]^{\delta}_{\eta}.1)$$

Fig. 9. Interpretation of disjointness contexts and of the subsumption relation.

4.2.2 Semantic interpretation of the subsumption relation. The *semantic interpretation of a subsumption relation on types, signatures, or rows* is a Hazel+ assertion relating the interpretation of types, signatures, or rows, respectively. Like the semantic judgment, the semantic interpretation of a subsumption relation employs a double turnstile symbol $\vDash$ to differentiate it from the syntactic subsumption relation. Its definition appears in Figure 9. This definition depends on the *interpretation of disjointness contexts*.

The interpretation of a disjointness context D asserts, for every effect name s in the domain of D, that s has been allocated, the assertion $\delta(s) \mapsto_{\square} ()$ holds, and that D maps s to a pair of a set of names S and a set of row variables V, such that there is no aliasing between s and S, $\delta(s)$ does not belong to the image of δ over S, nor between s and V, $\delta(s)$ does not belong to the image of η over V.

The interpretation of the relation $D \vdash v \leq_T \tau$ asserts that, under the non-aliasing assumption $[\![D]\!]^{\delta}_{\eta}$, every value in the interpretation of v belongs to the interpretation of τ. The interpretation of the relation $D \vdash_b \rho \leq_R \rho'$ asserts that the interpretation of ρ' is weaker than the interpretation of ρ (according to the order $\_ \leq_L \_$, defined in Subsection 4.1.2). Moreover, it also asserts that, if the permission b to erase absence signatures is off, then every label appears in $\mathcal{R}[\![\rho]\!]^{\delta}_{\eta}$ with multiplicity equal to, or less than, its multiplicity in $\mathcal{R}[\![\rho']\!]^{\delta}_{\eta}$. We express this property by implicitly regarding lists as multisets and by exploiting the subset relation on multisets $\_ \subseteq_m \_$. The interpretation of the relation $D \vdash \sigma \leq_S \sigma'$ asserts that (1) the interpretation of σ' is weaker than the interpretation of σ and that (2) the list of labels in $\mathcal{S}[\![\sigma]\!]^{\delta}_{\eta}.1$ is equal to $\mathcal{S}[\![\sigma']\!]^{\delta}_{\eta}.1$. This equality expresses the fact that the subsumption relation allows one to weaken/strengthen the types that appear in a signature, but not their effect names.

4.3 Soundness of Tes

The *Soundness Theorem* states that, if Tes accepts a program e, then e is safe (that is, e either diverges or terminates with a value):

Theorem 4.2 (Soundness). *If the judgment* $\emptyset \mid \emptyset \mid \emptyset \vdash_a e : \langle\rangle : ()$ *is derivable, then* e *is safe.*

To prove this theorem, we first establish the *Fundamental Theorem*:

Theorem 4.3 (Fundamental Theorem). *If the syntactic typing judgment* $\Xi \mid \Delta \mid \Gamma \vdash_a e : \rho : \tau$ *is derivable, then its semantic interpretation* $\Xi \mid \Delta \mid \Gamma \vDash_a e : \rho : \tau$ *holds.*

By exploiting Theorem 4.3, it follows that, if Tes assigns a program e the typing judgment $\emptyset \mid \emptyset \mid \emptyset \vdash_a e : \langle\rangle : ()$, then e enjoys the following specification: $wp_a\ e\ \langle \mathcal{R}[\![\langle\rangle]\!]^{\emptyset}_{\emptyset} \rangle \{ \mathcal{V}[\![()]\!]^{\emptyset}_{\emptyset} \}$.

To complete the proof of Theorem 4.2, it suffices to simplify $\mathcal{R}[\![\langle\rangle]\!]^{\emptyset}_{\emptyset}$ as [], and to apply Theorem 4.1.

The proof of Theorem 4.3 proceeds by induction on the derivation of typing judgments. Each typing rule gives rise to the proof obligation that the interpretation of judgments in the premise implies the interpretation of the judgment in the conclusion. For each typing rule, there is a well-suited reasoning rule allowing this proof obligation to be completed inside the Hazel+ logic. The case of the typing rule MonotonicityTyped relies on the *Fundamental Theorem of The Subsumption Relation*:

Theorem 4.4 (Fundamental Theorem of The Subsumption Relation). *(1) If $D \vdash \upsilon \leq_T \tau$ is derivable then $D \vDash \upsilon \leq_T \tau$ holds; (2) If $D \vdash \sigma \leq_S \sigma'$ is derivable then $D \vDash \sigma \leq_S \sigma'$ holds; (3) If $D \vdash_b \rho \leq_R \rho'$ is derivable then $D \vDash_b \rho \leq_R \rho'$ holds.*

Each of the statements in Theorem 4.4 is proved by induction on the subsumption derivation.

5 RELATED WORK

Hillerström and Lindley [19] study a core formal calculus that serves as a model of Links [12], a functional programming language for web applications that the authors extend with support for effect handlers. Taking advantage of Links's existing row-based approach to type-checking records, the authors consider a similar approach to the type-checking effectful programs: they annotate an arrow with a row of labels denoting the effects that a function might perform. Moreover, the system has support for row polymorphism following Rémy's discipline [37]: the kind system, in combination with a syntactic well-formedness criteria, ensures that labels in a row are pairwise distinct. Links does not support the dynamic generation of effect labels.

Bauer and Pretnar [2] present the theoretical foundation of *Eff*, a programming language with support for handlers, and dynamic generation of effect labels. However, the authors consider a subset of *Eff* that excludes the generation of effect labels. They endow this core *Eff* calculus with a row-based type system that ensures strong type safety. Later [3], the same authors extend this core calculus with support for dynamic generation of effect labels. However, in this extended calculus, strong type safety no longer holds: well-typed programs can perform unhandled effects. The system has support for value-polymorphic types, whose introduction applies only to programs that do not perform effects.

Leijen [29] formalizes a subset of the Koka language [30]. This formalization consists of a calculus with support for handlers and globally defined effects, of a type system with support for both value and effect polymorphism, and of a compilation strategy for explicitly typed programs. This strategy relies on a *selective CPS transformation* [32], which Leijen extends with support for effect-polymorphic programs. As in Tes, an arrow type in Leijen's type system is annotated with a row of effects. Unlike Tes, a row in Leijen's system is *univariate*: it can have at most one row variable. In Tes, the ability to have multiple variables in a row is important, for example, in the statement of the typing rule of a lexically scoped handler. Indeed, the client of such a handler has an effect-polymorphic type $\forall\theta.\ (\alpha \xrightarrow{\theta} \beta) \xrightarrow{\theta \cdot \rho} \tau$, where θ stands for the effect that is allocated by the handler. If row were univariate, then the row ρ could contain no variables, because, otherwise, the row $\theta \cdot \rho$ would not be univariate. Tes does not impose such a restriction. Another difference is the introduction of polymorphic types: in Tes, the generalization of type and row variables applies to programs that perform effects, whereas Leijen argues that, in a possible extension with primitive references, generalization would apply only to *total programs*: programs type-checked under the empty row, and, consequently, that do not perform effects. Tes shows that this restriction

is unnecessary: even in the presence of references, it is sound to generalize the type of a program that performs effects, provided that it does not interact with the store, that is, it neither uses references nor allocate labels.

A notable omission from Leijen's formalization is Koka's `inject` [28], a construct that works as a *lift* coercion. Biernacki et al. [5] are the first authors to provide a formal treatment of a calculus with such a construct. In addition to its formal operational semantics, they design a row-based type system (with support for effect polymorphism through univariate rows) for this calculus. They conceive the first binary logical relations for handlers, and they apply these relations to prove that their system is sound. In a later paper [6], the same authors introduce λ^{HEL}, a calculus with support for both the 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) the composition of coercions. These coercions do not add expressiveness to the language: they can all be written in terms of *lift*. Still, they facilitate how the programmer can modify the way in which an effect dynamically searches for a handler. They equip this calculus with a type system with support for effect-polymorphic, value-polymorphic, and existential types. Although the function *counter*, discussed in Sections 1 and 3, is expressible in λ^{HEL}, the type system designed by the authors does not accept this program. (This has been confirmed by the authors via personal communication.) The technical reason why this program is rejected is that the subsumption rules of their system are not flexible enough: a call to a function f with an abstract row θ cannot occur in a context whose row is not exactly θ. (It is not trivial how to overcome this issue, because, in their system, the interpretation of a signature depends on the signature's position in the row.) In Tes, such a function call can occur in any context whose row includes the variable θ.

Zhang and Myers [44] coin the term *accidental handling*, and introduce the *tunneling semantics* as a novel operational semantics that avoids accidental handling by construction. This semantics, however, is presented only informally as a program transformation. This transformation is not presented in the setting of $\lambda_{\Downarrow\Uparrow}$, a formal calculus introduced by the authors. Furthermore, as noted by Biernacki et al. [7], there is a discrepancy between the paper presentation of $\lambda_{\Downarrow\Uparrow}$ and its Coq formalization. In the article, there is no dynamic generation of fresh labels, whereas, in the Coq formalization, one finds a calculus with support for this feature through a construct that corresponds to a lexically scoped handler: it introduces a fresh effect label and installs a handler for this label. Therefore, if we take the Coq formalization as the main reference, then Zhang and Myers's work consists of the introduction of a calculus for lexically scoped handlers, of a type system for this calculus with support for effect polymorphism, and the soundness proof of this system using binary logical relations. They apply these binary logical relations to show interesting program equivalences. One of these equivalences, for example, shows that an effect-polymorphic function cannot intercept effects abstracted by a row variable. This property seems aligned with the intuitive idea of *absence of accidental handling*, but a formal definition of this term still does not exist. Zhang and Myers and other authors [11] suggest that the absence of accidental handling is equivalent to the parametricity of effect polymorphism. However, this definition is unsatisfactory, because there are systems both with parametric polymorphism and that allow programs to intercept effects abstracted by row variables. One example is the system Tes+Wind obtained by extending *λ-labels* with a "dynamic-wind" construct `dynamic-wind` $p\ e\ q$ [17], which monitors the execution of e by invoking the thunk p when control enters e (at the beginning of e's execution and every time e is resumed) and by invoking the thunk q when control leaves e (at the end of e's execution and every time e performs an unhandled effect). Then, we extend Tes with the following typing rule:

$$\textsc{DynamicWindTyped}\quad \frac{\Xi \mid \Delta \mid \Gamma \vdash_a e : \rho : \tau \qquad \Xi \mid \Delta \mid \Gamma \vdash_a p : \rho : () \to_a () \qquad \Xi \mid \Delta \mid \Gamma \vdash_a q : \rho : () \to_a ()}{\Xi \mid \Delta \mid \Gamma \vdash_a \texttt{dynamic-wind}\, p\, e\, q : \rho : \tau}$$

This rule preserves the interpretation presented in Section 4. Therefore, Tes+Wind has parametric effect polymorphism, because an effect-polymorphic type is interpreted by a universal quantifier, but it accepts the program `dynamic-wind` $p\ e\ q$, which breaks Zhang and Myers's equivalences and is thus an example of accidental handling.

Despite of their previous formal study of effect coercions, in the paper [7], Biernacki et al. argue against these constructs, which they deem as impractical for real-world programming, and conceive a type system for a language with support for lexically scoped handlers only. The authors present two semantics for this language: (1) the *open semantics*, where effect names are not substituted with labels, and evaluation is defined among open terms in a capture-avoiding way, and (2) the *generative semantics*, where, as in *λ-labels*, effect names are substituted with dynamic labels. By means of binary logical relations, the authors show that the system is sound and that these semantics are equivalent.

Kammar and Pretnar [24] show that a handler calculus without references and without allocation of effect labels admits a type system with unrestricted introduction of value-polymorphic types. In particular, the generalization of type variables applies to a program that handles and performs effects. Kammar and Pretnar establish the soundness of their system through the syntactic approach [43], whereas we establish Tes's soundness through the semantic approach: we provide a semantic interpretation of judgments as specifications in a program logic. This change in perspective leads to a key observation: from a semantic point of view, combining polymorphism with general references is akin to commuting a universal quantifier with an update modality.

6 CONCLUSION

With this paper, we are taking part in a long-ongoing debate on the most convenient constructs for programming with effect handlers and multiple named effects. We have argued in favor of dynamic generation of effect labels. We have formalized the dynamic semantics of this construct in the setting of *λ-labels*, a calculus with handlers, dynamic effect labels, multi-shot continuations, and general references. We have equipped this calculus with Tes, a simple yet expressive type system. Three main characteristic features in Tes are: (1) its interpretation of arrow types, which includes a separation requirement, stating that the dynamic labels denoted by the static row must be pairwise distinct; (2) its original subsumption rules, which allow extending a row, permuting rows, and erasing absence signatures, under a set of disjointness hypotheses; (3) its purity attributes, which indicate if a program interacts with the store, and allow restricting polymorphism to pure expressions. Tes supports lexically scoped handlers, but is not restricted to lexically scoped handlers. It is the first system that accepts *counter* and that is not restricted to lexically scoped handlers. We have proved that Tes is sound by following a semantic approach: we have introduced Hazel+, a novel Separation Logic for handlers and multiple named effects, and provided an interpretation of typing judgments as Hazel+ specifications. We have argued that "parametricity of effect polymorphism" and "absence of accidental handling" are both loosely-defined properties and that they are not necessarily equivalent. We argue that proving contextual equivalence laws is ultimately the only reliable way of showing absence of accidental handling, and we point out a tension, as the desire for strong equivalence laws rules out certain potentially desirable combinators, such as "catch-all" or "dynamic-wind" combinators. In future work, we intend to study binary logical relations and to establish contextual equivalence laws.

REFERENCES

[1] Amal J. Ahmed, Matthew Fluet, and Greg Morrisett. 2005. A step-indexed model of substructural state. In *International Conference on Functional Programming (ICFP)*. 78–91.

[2] Andrej Bauer and Matija Pretnar. 2014. An Effect System for Algebraic Effects and Handlers. *Logical Methods in Computer Science* 10, 4 (2014).

[3] Andrej Bauer and Matija Pretnar. 2015. Programming with algebraic effects and handlers. *Journal of Logical and Algebraic Methods in Programming* 84, 1 (2015), 108–123.

[4] Andrej Bauer and Matija Pretnar. 2020. Eff. http://www.eff-lang.org/.

[5] Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. 2018. Handle with care: relational interpretation of algebraic effects and handlers. *Proceedings of the ACM on Programming Languages* 2, POPL (2018), 8:1–8:30.

[6] Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. 2019. Abstracting algebraic effects. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 6:1–6:28.

[7] Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. 2020. Binders by day, labels by night: effect instances via lexically scoped handlers. *Proceedings of the ACM on Programming Languages* 4, POPL (2020), 48:1–48:29.

[8] Lars Birkedal and Aleš Bizjak. 2018. Lecture Notes on Iris: Higher-Order Concurrent Separation Logic. (Dec. 2018). Lectures notes.

[9] Jonathan Immanuel Brachthäuser and Philipp Schuster. 2017. Effekt: extensible algebraic effects in Scala. In *Symposium on Scala*. 67–72.

[10] Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. 2020. Effects as capabilities: effect handlers and lightweight effect polymorphism. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 126:1–126:30.

[11] Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. 2020. Effekt: Capability-passing style for type- and effect-safe, extensible effect handlers in Scala. *Journal of Functional Programming* 30 (2020), e8.

[12] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. 2006. Links: Web Programming Without Tiers. In *Formal Methods for Components and Objects (Lecture Notes in Computer Science, Vol. 4709)*. Springer, 266–296.

[13] Ana Lúcia de Moura and Roberto Ierusalimschy. 2009. Revisiting Coroutines. *ACM Transactions on Programming Languages and Systems* 31, 2 (Feb. 2009), 1–31.

[14] Paulo Emílio de Vilhena and François Pottier. 2021. A Separation Logic for Effect Handlers. *Proceedings of the ACM on Programming Languages* 5, POPL (Jan. 2021).

[15] Stephen Dolan, Spiros Eliopoulos, Daniel Hillerström, Anil Madhavapeddy, K. C. Sivaramakrishnan, and Leo White. 2017. Concurrent System Programming with Effect Handlers. In *Trends in Functional Programming (TFP) (Lecture Notes in Computer Science, Vol. 10788)*. Springer, 98–117.

[16] Andrzej Filinski. 1996. *Controlling Effects*. Ph.D. Dissertation. School of Computer Science, Carnegie Mellon University.

[17] Matthew Flatt, Gang Yu, Robert Bruce Findler, and Matthias Felleisen. 2007. Adding Delimited and Composable Control to a Production Programming Environment. In *International Conference on Functional Programming (ICFP)*. 165–176.

[18] Jacques Garrigue. 2004. Relaxing the Value Restriction. In *Functional and Logic Programming (Lecture Notes in Computer Science, Vol. 2998)*. Springer, 196–213.

[19] Daniel Hillerström and Sam Lindley. 2016. Liberating effects with rows and handlers. In *International Workshop on Type-Driven Development (TyDe@ICFP)*. 15–27.

[20] Daniel Hillerström and Sam Lindley. 2018. Shallow Effect Handlers. In *Asian Symposium on Programming Languages and Systems (APLAS) (Lecture Notes in Computer Science, Vol. 11275)*. Springer, 415–435.

[21] Daniel Hillerström, Sam Lindley, and Robert Atkey. 2020. Effect handlers via generalised continuations. *Journal of Functional Programming* 30 (2020), e5.

[22] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming* 28 (2018), e20.

[23] Ohad Kammar, Sam Lindley, and Nicolas Oury. 2013. Handlers in action. In *International Conference on Functional Programming (ICFP)*. 145–158.

[24] Ohad Kammar and Matija Pretnar. 2017. No value restriction is needed for algebraic effects and handlers. *Journal of Functional Programming* 27 (2017), e7.

[25] Oleg Kiselyov. 2014. Undelimited continuations are co-values rather than functions. (Oct. 2014). https://okmij.org/ftp/continuations/undelimited.html.

[26] Robert Krebbers, Amin Timany, and Lars Birkedal. 2017. Interactive proofs in higher-order concurrent separation logic. In *Principles of Programming Languages (POPL)*.

[27] Mortern Krogh-Jespersen, Kasper Svendsen, and Lars Birkedal. 2017. A Relational Model of Type-and-Effects in Higher-Order Concurrent Separation Logic. In *Principles of Programming Languages (POPL)*. 218–231.

[28] Daan Leijen. 2014. Koka: Programming with Row Polymorphic Effect Types. In *Workshop on Mathematically Structured Functional Programming (MSFP)*, Vol. 153. 100–126.

[29] Daan Leijen. 2017. Type directed compilation of row-typed algebraic effects. In *Principles of Programming Languages (POPL)*. 486–499.

[30] Daan Leijen. 2020. Koka. https://www.microsoft.com/en-us/research/project/koka/.

[31] Sam Lindley, Conor McBride, and Craig McLaughlin. 2017. Do Be Do Be Do. In *Principles of Programming Languages (POPL)*.

[32] Lasse R. Nielsen. 2001. A Selective CPS Transformation. *Electronic Notes in Theoretical Computer Science* 45 (Nov. 2001), 311–331.

[33] Peter W. O'Hearn. 2007. Resources, Concurrency and Local Reasoning. *Theoretical Computer Science* 375, 1–3 (May 2007), 271–307.

[34] Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. 2001. Local Reasoning about Programs that Alter Data Structures. In *Computer Science Logic (Lecture Notes in Computer Science, Vol. 2142)*. Springer, 1–19.

[35] Benjamin C. Pierce. 2002. *Types and Programming Languages*. MIT Press.

[36] Gordon D. Plotkin and Matija Pretnar. 2009. Handlers of Algebraic Effects. In *European Symposium on Programming (ESOP) (Lecture Notes in Computer Science, Vol. 5502)*. Springer, 80–94.

[37] Didier Rémy. 1989. Type checking records and variants in a natural extension of ML. In *Principles of Programming Languages (POPL)*. 77–88.

[38] Philipp Schuster, Jonathan Immanuel Brachthäuser, Marius Müller, and Klaus Ostermann. 2022. A Typed Continuation-Passing Translation for Lexical Effect Handlers. In *Programming Language Design and Implementation (PLDI)*. 566–579.

[39] K. C. Sivaramakrishnan, Stephen Dolan, Leo White, Tom Kelly, Sadiq Jaffer, and Anil Madhavapeddy. 2021. Retrofitting effect handlers onto OCaml. In *Programming Language Design and Implementation (PLDI)*. 206–221.

[40] Mads Tofte. 1990. Type Inference for Polymorphic References. *Information and Computation* 89, 1 (1990), 1–34.

[41] Simon Friis Vindum and Lars Birkedal. 2021. Contextual refinement of the Michael-Scott queue. In *Certified Programs and Proofs (CPP)*. 76–90.

[42] Andrew K. Wright. 1995. Simple Imperative Polymorphism. *Lisp and Symbolic Computation* 8, 4 (Dec. 1995), 343–356.

[43] Andrew K. Wright and Matthias Felleisen. 1994. A Syntactic Approach to Type Soundness. *Information and Computation* 115, 1 (Nov. 1994), 38–94.

[44] Yizhou Zhang and Andrew C. Myers. 2019. Abstraction-safe effect handlers via tunneling. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 5:1–5:29.

7 TECHNICAL APPENDIX

Typing judgment $\boxed{\Xi \mid \Delta \mid \Gamma \vdash_a e : \rho : \tau}$

UnitTyped
$$\Xi \mid \Delta \mid \Gamma \vdash_a () : \rho : ()$$

BoolTyped
$$\Xi \mid \Delta \mid \Gamma \vdash_a b : \rho : \texttt{bool}$$

IntTyped
$$\Xi \mid \Delta \mid \Gamma \vdash_a i : \rho : \texttt{int}$$

NilTyped
$$\Xi \mid \Delta \mid \Gamma \vdash_a [] : \rho : \tau\ \texttt{list}$$

ConsTyped
$$\frac{\Xi \mid \Delta \mid \Gamma \vdash_a h : \rho : \tau \qquad \Xi \mid \Delta \mid \Gamma \vdash_a t : \rho : \tau\ \texttt{list}}{\Xi \mid \Delta \mid \Gamma \vdash_a h :: t : \rho : \tau}$$

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

ListMatchTyped
$$\frac{\Xi \mid \Delta \mid \Gamma \vdash_a e : \rho : \tau\ \texttt{list} \qquad \Xi \mid \Delta \mid \Gamma \vdash_a l : \rho : \tau \to_a \tau\ \texttt{list} \xrightarrow{\rho}_a \tau' \qquad \Xi \mid \Delta \mid \Gamma \vdash_a r : \rho : \tau'}{\Xi \mid \Delta \mid \Gamma \vdash_a \texttt{match}\ e\ \texttt{with}\ (l \mid r) : \rho : \tau'}$$

AppTyped
$$\frac{\Xi \mid \Delta \mid \Gamma \vdash_a e : \rho : \upsilon \xrightarrow{\rho}_a \tau \qquad \Xi \mid \Delta \mid \Gamma \vdash_a e' : \rho : \upsilon}{\Xi \mid \Delta \mid \Gamma \vdash_a e\ e' : \rho : \tau}$$

TLamTyped
$$\frac{\alpha \notin \Xi, \Gamma, \rho, \tau \qquad \Xi, \alpha \mid \Delta \mid \Gamma \vdash_P e : \rho : \tau}{\Xi \mid \Delta \mid \Gamma \vdash_a e : \rho : \forall \alpha.\, \tau}$$

RLamTyped
$$\frac{\theta \notin \Xi, \Gamma, \rho, \tau \qquad \Xi, \theta \mid \Delta \mid \Gamma \vdash_P e : \rho : \tau}{\Xi \mid \Delta \mid \Gamma \vdash_a e : \rho : \forall \theta.\, \tau}$$

ReadTyped
$$\frac{\Xi \mid \Delta \mid \Gamma \vdash_a e : \rho : \tau\ \texttt{ref}}{\Xi \mid \Delta \mid \Gamma \vdash_I !e : \rho : \tau}$$

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

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

WriteTyped
$$\frac{\Xi \mid \Delta \mid \Gamma \vdash_a e : \rho : \tau\ \texttt{ref} \qquad \Xi \mid \Delta \mid \Gamma \vdash_{a'} e' : \rho : \tau}{\Xi \mid \Delta \mid \Gamma \vdash_I e := e' : \rho : ()}$$

LetEffTyped
$$\frac{s \notin \Gamma, \rho, \tau \qquad \Xi \mid \Delta, s \mid \Gamma \vdash_a e : (s : \texttt{abs}) \cdot \rho : \tau}{\Xi \mid \Delta \mid \Gamma \vdash_I \texttt{effect}\ s\ \texttt{in}\ e : \rho : \tau}$$

AllocTyped
$$\frac{\Xi \mid \Delta \mid \Gamma \vdash_a e : \rho : \tau}{\Xi \mid \Delta \mid \Gamma \vdash_I \texttt{ref}\ e : \rho : \tau\ \texttt{ref}}$$

PerformTyped
$$\frac{\Xi \mid \Delta \mid \Gamma \vdash_a e : \rho : \upsilon \qquad s \in \Delta \qquad (s : \upsilon \Rightarrow \tau) \in \rho}{\Xi \mid \Delta \mid \Gamma \vdash_a \texttt{perform}\ s\ e : \rho : \tau}$$

RecTyped
$$\frac{\Xi \mid \Delta \mid f : \upsilon \xrightarrow{\rho}_a \tau, x : \upsilon, \Gamma \vdash_a e : \rho : \tau}{\Xi \mid \Delta \mid \Gamma \vdash_{a'} \texttt{rec}\ f\ x.\ e : \langle\rangle : \upsilon \xrightarrow{\rho}_a \tau}$$

HandleTyped
$$\frac{\sigma = (s : \upsilon \Rightarrow \tau) \qquad \sigma' = (s : \upsilon' \Rightarrow \tau') \qquad \rho' = \sigma' \cdot \rho \qquad s \in \Delta \qquad \Xi \mid \Delta \mid \Gamma \vdash_a e : \sigma \cdot \rho : \kappa \qquad \Xi \mid \Delta \mid \Gamma \vdash_a h : \rho' : \upsilon \to_a (\tau \xrightarrow{\rho'}_a \kappa') \xrightarrow{\rho'}_a \kappa' \qquad \Xi \mid \Delta \mid \Gamma \vdash_a r : \rho' : \kappa \xrightarrow{\rho'}_a \kappa'}{\Xi \mid \Delta \mid \Gamma \vdash_a \texttt{handle}\ e\ \texttt{with}\ (s : h \mid r) : \rho' : \kappa'}$$

MonotonicityTyped
$$\frac{\Xi \mid \Delta \mid \Gamma \vdash_a e : \rho : \tau \qquad a \preceq_A a' \qquad \vdash_b \rho \preceq_R \rho' \qquad \vdash \tau \preceq_T \tau'}{\Xi \mid \Delta \mid \Gamma \vdash_{a'} e : \rho' : \tau'}$$

$$\textsc{MatchTyped}\quad \frac{\Xi \mid \Delta \mid \Gamma \vdash_a e : \rho : \tau_1 + \tau_2 \qquad \Xi \mid \Delta \mid \Gamma \vdash_a e_i : \rho : \tau_i \xrightarrow{\rho}_a \tau}{\Xi \mid \Delta \mid \Gamma \vdash_a \texttt{match}\ e\ \texttt{with}\ (e_1 \mid e_2) : \rho : \tau}$$

$$\textsc{InjTyped}\quad \frac{\Xi \mid \Delta \mid \Gamma \vdash_a e : \rho : \tau_i}{\Xi \mid \Delta \mid \Gamma \vdash_a \texttt{inj}_i\ e : \rho : \tau_1 + \tau_2}$$

$$\textsc{PairTyped}\quad \frac{\Xi \mid \Delta \mid \Gamma \vdash_a e_i : \rho : \tau_i}{\Xi \mid \Delta \mid \Gamma \vdash_a (e_1, e_2) : \rho : \tau_1 * \tau_2}$$

$$\textsc{ProjTyped}\quad \frac{\Xi \mid \Delta \mid \Gamma \vdash_a e : \rho : \tau_1 * \tau_2}{\Xi \mid \Delta \mid \Gamma \vdash_a \texttt{proj}_i\ e : \rho : \tau_i}$$

$$\textsc{IfThenElseTyped}\quad \frac{\Xi \mid \Delta \mid \Gamma \vdash_a e_b : \rho : \texttt{bool} \qquad \Xi \mid \Delta \mid \Gamma \vdash_a e_i : \rho : \tau}{\Xi \mid \Delta \mid \Gamma \vdash_a \texttt{if}\ e_b\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \rho : \tau}$$

$$\textsc{OpTyped}\quad \frac{\vdash_{Op} \odot : v \longrightarrow \tau}{\Xi \mid \Delta \mid \Gamma \vdash_a \odot : \rho : v \xrightarrow{\rho}_a \tau}$$

Fig. 10. Typing rules

Operation judgment $\boxed{\vdash_{Op} \odot : v \longrightarrow \tau}$

$$\vdash_{Op} \texttt{+} : \texttt{int} * \texttt{int} \longrightarrow \texttt{int} \qquad \vdash_{Op} \texttt{not} : \texttt{bool} \longrightarrow \texttt{bool}$$

$$\frac{\odot \in \{\texttt{and}, \texttt{or}\}}{\vdash_{Op} \odot : \texttt{bool} * \texttt{bool} \longrightarrow \texttt{bool}} \qquad \frac{\tau \in \{(), \texttt{bool}, \texttt{int}\}}{\vdash_{Op} \texttt{==} : \tau * \tau \longrightarrow \texttt{bool}}$$

Fig. 11. Operation rules.

Head-reduction relation $\boxed{e \mid \sigma \longrightarrow e \mid \sigma}$

$$\begin{array}{rcl}
\texttt{ref}\ v \mid \sigma & \longrightarrow & \ell \mid \sigma[\ell \mapsto v] \quad \ell \notin \mathrm{dom}\ \sigma \\
!\,\ell \mid \sigma & \longrightarrow & v \mid \sigma \quad \sigma(\ell) = v \\
\ell := v \mid \sigma & \longrightarrow & () \mid \sigma[\ell \mapsto v] \quad \ell \in \mathrm{dom}\ \sigma \\
(\texttt{rec}\ f\ x.\ e)\ v \mid \sigma & \longrightarrow & e[(\texttt{rec}\ f\ x.\ e)/f][v/x] \mid \sigma \\
(\texttt{cont}\ K)\ v \mid \sigma & \longrightarrow & K[v] \mid \sigma \\
\texttt{effect}\ s\ \texttt{in}\ e \mid \sigma & \longrightarrow & e[\ell/s] \mid \sigma[\ell \mapsto ()] \quad \ell \notin \mathrm{dom}\ \sigma \\
\texttt{perform}\ \ell\ v \mid \sigma & \longrightarrow & \texttt{eff}\ \ell\ v\ \bullet \mid \sigma \\
K[\texttt{eff}\ \ell\ v\ K'] \mid \sigma & \longrightarrow & \texttt{eff}\ \ell\ v\ (K[K']) \mid \sigma \quad K \textit{ is neutral} \\
\texttt{handle}\ (\texttt{eff}\ \ell_2\ v\ K)\ \texttt{with}\ (\ell_1 : h \mid r) \mid \sigma & \longrightarrow & \texttt{eff}\ \ell_2\ v\ (\texttt{handle}\ K\ \texttt{with}\ (\ell_1 : h \mid r)) \mid \sigma \quad \ell_1 \neq \ell_2 \\
\texttt{handle}\ (\texttt{eff}\ \ell\ v\ K)\ \texttt{with}\ (\ell : h \mid r) \mid \sigma & \longrightarrow & h\ v\ (\texttt{cont}\ (\texttt{handle}\ K\ \texttt{with}\ (\ell : h \mid r))) \mid \sigma \\
\texttt{handle}\ v\ \texttt{with}\ (\ell_1 : h \mid r) \mid \sigma & \longrightarrow & r\ v \mid \sigma
\end{array}$$

Fig. 12. Head-reduction rules.