Backwards-Compatible Row-Based Exceptions in ML

SIMCHA VAN COLLEM, Radboud University Nijmegen, The Netherlands
PAULO EMÍLIO DE VILHENA, Imperial College London, United Kingdom
ROBBERT KREBBERS, Radboud University Nijmegen, The Netherlands

We introduce a type system that provides strong types for exception tracking in languages in the ML family. Our type system employs a rich notion of row polymorphism and subtyping to ensure backwards compatibility, making sure that code without exception tracking remains to work and can be generalized gracefully to support exception tracking. We study the safety and abstraction guarantees of our type system, in particular the role of local exceptions for data abstraction. We formulate these claims using binary logical relations in a novel relational separation logic for exceptions, an independent contribution of this paper. We support a realistic subset of features from ML, such as extensible variant types and local exceptions, both of which are not supported by prior type systems and logics with exception tracking. We exercise our type system and logic on a number of challenging examples taken from the OCaml standard library, from one of Jane Street's OCaml libraries, and from Filinksi's PhD thesis. All our results are mechanized in the Rocq prover using Iris.

1 Introduction

Exceptions are widely used for error handling and writing efficient and concise programs using non-local control flow. Exceptions are also error-prone. Programmers should ensure exceptions are handled, to avoid unexpected crashes, and that exception names do not collide, to avoid catching more exceptions than intended. In this paper, we develop a type system that provides programmers with strong types to control the correct usage of exceptions and thereby avoid such programming mistakes. We formulate these claims in a novel relational separation logic for exceptions, the first of this kind. Moreover, we consider a realistic subset of ML-style languages, such as Standard ML and OCaml, and pay especial attention to the compatibility issues between the standard typing discipline of such languages and our design. In sum, we focus on the following aspects.

Feature support. We should support two key features of exceptions in ML: the ability to treat exceptions as first-class data through the *extensible variant type* **exn**, which is important, for example, to catch an exception in one thread and re-raise it in another thread; and the ability to declare exceptions locally. **Backwards compatibility.** We should allow programmers to opt out from exception tracking in (parts of) their program. To achieve this, our system should enjoy two properties. First, it should be possible to encode the standard ML types and typing rules so that code without exception tracking remains to work. Second, it should be possible to generalize types one at a time without breaking old code. We call this second property *graceful generalization.* **Safety and abstraction theorems.** Our type system should ensure type and exception safety: a program with a sufficiently strong type does not raise uncaught exceptions. Moreover, it should guarantee *representation independence* [45, 52], which roughly says that, if a program uses a library or function through an abstract interface, then changes to the internal representation of this library or function cannot affect the program's behavior. Representation independence has been studied extensively, particularly in the context of local state [2, 50, 56], but not in the context of both extensible variant types and exceptions. Finally, the type system should guarantee that, if a program uses an exception abstractly, then locally declared exceptions cannot be used to catch this exception.

Our type system. We extend a language in the style of ML with rows [51, 61] and row polymorphism to enable exception tracking. As standard in type systems based on row typing [21, 32, 33, 39],

Authors' Contact Information: Simcha Van Collem, simcha.vancollem@ru.nl, Radboud University Nijmegen, Nijmegen, The Netherlands; Paulo Emílio de Vilhena, p.de-vilhena@imperial.ac.uk, Imperial College London, London, United Kingdom; Robbert Krebbers, mail@robbertkrebbers.nl, Radboud University Nijmegen, Nijmegen, The Netherlands.

function types are of the shape $\tau \to_\rho \sigma$, where the *row* ρ describes which exceptions can be raised. What is novel about our type system is that it is backwards compatible with ML in two ways. First, every typing rule from an ordinary ML type system can be derived from our generalized typing rules. The key idea to enable this encoding is (1) to introduce a *top row* $\top$, stating any exception can be raised and thus representing the lack of exception tracking, and (2) to see the ordinary function type $\tau \to \sigma$ as $\tau \to_\top \sigma$. Second, thanks to carefully designed yet simple subtyping rules, our system allows for a *graceful transition* to exception tracking, where programs can be ported one function at a time. Let us illustrate this second aspect through the following example:

$$\texttt{with\_return} : \forall \alpha.\, ((\alpha \to \bot) \to \alpha) \to \alpha$$
$$\texttt{with\_return}\, f \triangleq \textbf{let exn}\, \mathit{Return}\, \textbf{in try}\, f\, (\lambda x.\, \textbf{raise}\, (\mathit{Return}\, x))\, \textbf{with}\, \mathit{Return}\, x \Rightarrow x$$
$$\textbf{mapM}_{\textsf{exn}} : \forall \alpha, \beta.\, (\alpha \to \textbf{option}\, \beta) \to \textbf{list}\, \alpha \to \textbf{option}\, (\textbf{list}\, \beta)$$
$$\textbf{mapM}_{\textsf{exn}}\, f\, xs \triangleq \texttt{with\_return}\, (\lambda \mathit{return}.\, \textbf{Some}\, (\texttt{List.map}\, (\lambda x.\, \textbf{match}\, f\, x\, \textbf{with Some}\, y \Rightarrow y \mid \textbf{None} \Rightarrow \mathit{return}\, \textbf{None})\, xs))$$

The higher-order function $\textbf{mapM}_{\textsf{exn}}$ maps a function f over a list xs using **List.map** of type $\forall \alpha, \beta.\, (\alpha \to \beta) \to \textbf{list}\, \alpha \to \textbf{list}\, \beta$ from the OCaml standard library. The function f might fail by returning **None**, in which case $\textbf{mapM}_{\textsf{exn}}$ breaks out of **List.map** and returns **None**. Instead of using exceptions directly, the implementation relies on **with_return** (a simplification of the homonymous function from Jane Street [23]), which uses a *local exception Return* to implement a *return* functionality, a common feature in imperative languages.

The function $\textbf{mapM}_{\textsf{exn}}$ type checks in our system with the specified signature. Because ordinary types are implicitly annotated with $\top$, this signature does not track exceptions. How to update $\textbf{mapM}_{\textsf{exn}}$'s type to track exceptions without breaking clients of $\textbf{mapM}_{\textsf{exn}}$'s original type? A similar question applies to how the types of **List.map** and **with_return** can be generalized without making $\textbf{mapM}_{\textsf{exn}}$ ill-typed. In our system, all these types can be generalized one at a time, without invalidating the types remaining to be generalized. The key idea is that, when generalizing a type to a row-polymorphic one, the original type can still be obtained either by simple specialization of the row-polymorphic type to $\top$ or via subtyping rules. Indeed, after generalizing **List.map**'s type to $\forall \theta, \alpha, \beta.\, (\alpha \to_\theta \beta) \to_{\langle\rangle} \textbf{list}\, \alpha \to_\theta \textbf{list}\, \beta$, for example, the function $\textbf{mapM}_{\textsf{exn}}$ remains typeable because we can instantiate θ with $\top$ and subtype the empty row $\langle\rangle$ to $\top$. The same applies to the generalization of **with_return**'s type to $\forall \theta, \alpha.\, (\forall \theta'.\, (\alpha \to_{\theta'} \bot) \to_{\theta \cdot \theta'} \alpha) \to_\theta \alpha$, where the row θ' abstracts the local exception *Return*, and to the generalization of $\textbf{mapM}_{\textsf{exn}}$'s type to $\forall \theta, \alpha, \beta.\, (\alpha \to_\theta \textbf{option}\, \beta) \to_{\langle\rangle} \textbf{list}\, \alpha \to_\theta \textbf{option}\, (\textbf{list}\, \beta)$.

Of course, we are not the first to extend ML languages with exception tracking or to consider associated compatibility issues. Leroy and Pessaux [39] design an extension of OCaml with a powerful inference engine to provide a fine analysis of exceptions including the values with which they are raised (to support legacy uses of exceptions where error codes are represented by exception arguments). Our focus is on a declarative type system with a rich notion of polymorphism and subtyping and on the formalization of the system's abstraction guarantees. New et al. [47] apply gradual typing to address the migration from unchecked to checked effects. Similar to our top row $\top$, their row ? accounts for the absence of effect tracking. Our approaches to backwards compatibility are otherwise different. They allow for (dynamically checked) *down casts* to turn a function with row ? into a function with a concrete row. We only allow *up casts* and instead use a rich notion of polymorphism and subtyping instead of dynamic checks. See §6 for more details.

Local exceptions. A key feature that is missing in prior work are local exceptions. Why are they challenging? To see why, consider the following example:

$$\texttt{with\_return}\, (\lambda \mathit{return}_1.\, 20 + \texttt{with\_return}\, (\lambda \mathit{return}_2.\, \mathit{return}_1\, 22))$$

The call to $return_1$ raises an exception that should be caught by the outer **with_return**, so this program should return 22. This behavior relies on **with_return** using a local exception and on the semantics of local exceptions in ML. If both instances of **with_return** used a global exception *Return*, then the names would collide and the program would return 42. Semantically, exceptions in Standard ML [44, §2.6] and OCaml [53] are not represented by their name, but by a fresh exception label that is dynamically generated by the exception declaration at runtime. One challenge that this entails is how to define the meaning of rows, in general, and of $\top$, in particular. Unlike Leroy and Pessaux [39], we cannot define the row $\top$ as the set of all exception names that statically appear in the program: we should be careful about the scope of each exception.

Extensible variant types. Standard ML [44, §2.6] and OCaml [34] provide the *dynamically extensible variant type* **exn** to treat exceptions as first-class data. Each time a new exception is declared, it is added as a new variant constructor of **exn**. The **raise** construct takes any value of type **exn**. Using **try** e **with** $x \Rightarrow e$ one can catch *any* exception (the bound variable x has type **exn**) and one can **match** on values of type **exn** to determine which exception is at hand.

Similar to Leroy and Pessaux [39], we annotate the $\mathbf{exn}_\rho$ type with a row ρ. This extension allows us to assign types to functions which use exceptions as first-class data. For example, the **Lazy** [37] and **Domain** [35] libraries in OCaml allow one to suspend computations or to execute computations in a new thread (called a *domain* in OCaml). In both cases, if the computation f raises *any* exception, this exception is stored in a reference and re-raised each time the computation is forced/joined. Using the $\mathbf{exn}_\rho$ type for the reference allows us to assign these libraries types that are polymorphic in the exceptions raised by f. Our general types for these libraries, in turn, can be subtyped to the ordinary OCaml types without exception tracking.

Safety and abstraction theorems. Using the Rocq prover and the Iris framework [25–30] we prove that our type system enjoys type and exception safety and representation independence. Since our type system is backwards compatible with ML, our proofs extend to ordinary ML with local exceptions and dynamically extensible variant types. Representation independence of ML with neither of these features has been considered in prior work.

Let us consider some examples to see how representation independence is relevant. The $\mathbf{mapM}_{\mathsf{exn}}$ function we previously discussed should be *contextually equivalent* to a version $\mathbf{mapM}_{\mathsf{opt}}$ (with the same type) which internally uses the option monad instead of exceptions:

$$\begin{aligned}\mathbf{mapM}_{\mathsf{opt}}\ f\ xs \triangleq\ & \mathbf{match}\ xs\ \mathbf{with}\ [\,] \Rightarrow \mathbf{Some}\ [\,] \mid x :: xs' \Rightarrow \\ & f\ x \gg= \mathbf{fun}\ y \Rightarrow (\mathbf{mapM}_{\mathsf{opt}}\ xs'\ f) \gg= \mathbf{fun}\ ys \Rightarrow \mathbf{Some}\ (y :: ys)\end{aligned}$$

Contextual equivalence of e_1 and e_2 means that e_1 can be replaced with e_2 in any typed program e without affecting e's behavior. The equivalence between $\mathbf{mapM}_{\mathsf{exn}}$ and $\mathbf{mapM}_{\mathsf{opt}}$ relies on the exception *Return* (used in **with_return**) being private and, by consequence, on the function argument f not being able to throw *Return*. If *Return* were not private (that is, if it were declared as a global exception) or if the operational semantics did not guarantee freshness of exception labels, then the equivalence would fail: in $\mathbf{mapM}_{\mathsf{exn}}$, a *Return* exception thrown by f would be handled accidentally by the **try** in **with_return**, whereas in $\mathbf{mapM}_{\mathsf{opt}}$ such an exception would propagate.

There is a close relationship between local references and local exceptions. It is known that keeping local references private is important for representation independence [2, 50, 56]. Consider:

$$\mathbf{let}\ r = \mathbf{ref}\ 10\ \mathbf{in}\ f\ ();\ !r\ =_{\mathsf{ctx}}\ f\ ();10 \qquad \mathbf{let\ exn}\ E\ \mathbf{in\ try}\ f\ ()\ \mathbf{with}\ E\ x \Rightarrow e\ =_{\mathsf{ctx}}\ f\ ()$$

The first (well-known) equivalence holds because the reference r is local and the (unknown) function f is thus unable to modify it. Similarly, we observe that the second equivalence holds because the exception E is local and the (unknown) function f cannot raise it. The more complicated equivalence between $\mathtt{mapM}_{\mathsf{exn}}$ and $\mathtt{mapM}_{\mathsf{opt}}$ crucially relies on equivalences of the second form.

We develop a binary logical relations model from which safety and representation independence follows. We show that our model is rich enough to prove non-trivial contextual equivalences and refinements. We prove equivalences between $\mathtt{mapM}_{\mathsf{exn}}$ and $\mathtt{mapM}_{\mathsf{opt}}$ and that a version of the **Lazy** module from the OCaml standard library is a refinement of the **Domain** module. Finally, we prove various equivalences involving interesting implementations of extensible variant types introduced by Filinski in his PhD thesis [17, §4.5, Remark 4.30].

Our relational separation logic. We define our binary logical relations in a novel relational separation logic for exceptions. Although the approach of using a relational logic for defining binary logical relations is not new [7, 16, 56, 59], our logic is the first that supports (local) exceptions and extensible variant types. Our program logic has a number of novel ingredients.

First, inspired by Simuliris [20], we generalize the postcondition to range over expressions instead of values. This allows us to obtain simple reasoning rules and state simple specifications to relate code that raises exceptions to code that does not—a crucial ability for modular verification. In contrast to previous relational logics for effects [13] based on *biorthogonality* [49], which have a similar goal, our logic keeps a simple refinement judgment (without an extra parameter).

Second, we show that the generalization of the postcondition allows us to state a single invariant rule. In contrast, prior relational logics with support for invariants such as ReLoC [18, 19], have a separate invariant rules for each atomic instruction (such as, load, store, and **cas**).

Third, based on Iris's higher-order ghost state [25] and on Timany et al. [57]'s *monotone partial bijections*, we develop a theory of *exception-signature assertions* to model rows and dynamically extensible variant types. Our theory supports variant constructors that refer to the variant type itself, even in negative position. We show how it enables reasoning about freshness of exception labels and and how it extends to Filinski [17]'s implementations of extensible variant types.

Contributions. In sum, our contributions are as follows:

- We define ExceptionLang, a language with higher-order references, polymorphism, concurrency, and a type system based on rows for (local) exceptions (§2). We provide a *top row* $\top$ and a rich notion of subtyping to ensure backwards compatibility with ML.
- We develop a relational separation program logic for compositional verification of ExceptionLang programs. The logic employs a generalized form of postcondition suitable to relate code that raises an exception to code that returns normally (§3). The generalized postcondition additionally enables us to state a single proof rule for Iris-style invariants.
- We develop a logical relations model for the ExceptionLang type system on top of our program logic (§4). The key ingredient is a higher-order ghost theory to concisely model rows and the extensible variant type **exn**. Using our logical relations model, we prove type and exception safety, as well as representation independence (§5).
- We mechanize all our results in the Rocq prover using Iris [5].

2 Language and Type System

We present the syntax and semantics of ExceptionLang (§2.1 and 2.2), its type system (§2.3), our approach to backwards compatibility (§2.4 and 2.5), and the safety and abstraction theorems (§2.6).

2.1 Syntax

Figure 1a shows the syntax of values, expressions, and evaluation contexts in ExceptionLang. The syntax of evaluation contexts reveals a call-by-value semantics with right-to-left evaluation order.

$$v ::= () \mid \mathbf{rec}\, f\, x = e \mid C\, v \mid \ell \mid l\, v \mid \ldots \qquad N ::= E \mid l$$
$$e ::= v \mid x \mid e\, e \mid C\, e \mid \mathbf{match}\, e\, \mathbf{with}\, \overline{C\, x \Rightarrow e} \mid \mathbf{ref}\, e \mid !\, e \mid e \leftarrow e \mid \mathbf{cas}\ e\, e\, e \mid \mathbf{fork}\, \{e\}$$
$$\mid\ \mathbf{let\, exn}\, E\, \mathbf{in}\, e \mid N\, e \mid \mathbf{raise}\, e \mid \mathbf{try}\, e\, \mathbf{with}\, x \Rightarrow e \mid \mathbf{match}\, e\, \mathbf{with}\, N\, x \Rightarrow e \mid y \Rightarrow e \mid \ldots$$
$$K ::= [] \mid e\, K \mid K\, v \mid l\, K \mid \mathbf{raise}\, K \mid \mathbf{try}\, K\, \mathbf{with}\, x \Rightarrow e \mid \mathbf{match}\, K\, \mathbf{with}\, l\, x \Rightarrow e \mid y \Rightarrow e \mid \ldots$$

(a) Syntax.

$$\begin{array}{rl} (\text{if } \mathit{neutral}\, K)\ \ K[\mathbf{raise}\, v] & \longrightarrow_{\text{p}}\ \mathbf{raise}\, v \\ \mathbf{try}\, v\, \mathbf{with}\, x \Rightarrow h & \longrightarrow_{\text{p}}\ v \\ \mathbf{try}\, (\mathbf{raise}\, v)\, \mathbf{with}\, x \Rightarrow h & \longrightarrow_{\text{p}}\ h[v/x] \\ \mathbf{match}\, (l\, v)\, \mathbf{with}\, l\, x \Rightarrow h \mid y \Rightarrow h' & \longrightarrow_{\text{p}}\ h[v/x] \\ \mathbf{match}\, (l'\, v)\, \mathbf{with}\, l\, x \Rightarrow h \mid y \Rightarrow h' & \longrightarrow_{\text{p}}\ h'[l'\, v/y] \end{array}$$

$$\begin{array}{rl} (\mathbf{let\, exn}\, E\, \mathbf{in}\, e, s) & \longrightarrow_{\text{t}}\ (e[l/E], \{l\} \uplus s, \epsilon) \\ (\mathbf{fork}\, \{e\}, s) & \longrightarrow_{\text{t}}\ ((), s, [e]) \end{array}$$

Pure-Step
$$\frac{e_1 \longrightarrow_{\text{p}} e_2}{(e_1, s) \longrightarrow_{\text{t}} (e_2, s, \epsilon)}$$

Bind-Step
$$\frac{(e_1, s_1) \longrightarrow_{\text{t}} (e_2, s_2, \vec{e}_f)}{(K[e_1], s_1) \longrightarrow_{\text{t}} (K[e_2], s_2, \vec{e}_f)}$$

TP-Step
$$\frac{(e_1, s_1) \longrightarrow_{\text{t}} (e_2, s_2, \vec{e}_f)}{(\vec{e} +\!\!+ e_1 :: \vec{e}', s_1) \longrightarrow_{\text{tp}} (\vec{e} +\!\!+ e_2 :: \vec{e}' +\!\!+ \vec{e}_f, s_2)}$$

(b) Operational semantics.

Fig. 1. Syntax and operational semantics of ExceptionLang.

Most of the value and expression constructs are standard; they closely follow the syntax of ML. In the following paragraphs, we discuss the unusual elements of the syntax.

Variants. The syntax $C\, e$ is used to construct values using one of the constructors C of a user-defined variant type.[1] Such values can be pattern matched using the syntax $\mathbf{match}\, e\, \mathbf{with}\, \overline{C\, x \Rightarrow e}$.

Function declarations. Function declarations $\mathbf{rec}\, f\ x = e$ have one formal argument x and include a binder f denoting the function itself to allow recursive declarations. We obtain a standard syntax via syntactic sugar: non-recursive functions $\mathbf{fun}\, x.\, e$ are defined as $\mathbf{rec}\, \_\ x = e$ and functions with multiple formal parameters $\mathbf{fun}\, x \ldots y.\, e$ are defined as $\mathbf{fun}\, x.\, \ldots\, \mathbf{fun}\, y.\, e$.

Exceptions. The exception handler $\mathbf{try}\, e\, \mathbf{with}\, x \Rightarrow h$ handles any exception raised by e. Specifically, if e raises an exception with $\mathbf{raise}\, v$, then control is transferred to the *handler branch* h with x bound to v. Any expression e can be used as the argument of **raise**, however, as we are going to see (§ 2.3), the type system allows only $E\, e$ expressions to be used as such. The term E is an *exception name*. At runtime, an exception name is bound to an *exception label* l.[2] The *local-exception declaration* $\mathbf{let\, exn}\, E\, \mathbf{in}\, e$ introduces an exception name E whose scope is e. At runtime, this instruction allocates a fresh exception label l to which E is bound. The *exception-matching construct* $\mathbf{match}\, e\, \mathbf{with}\, E'\ x \Rightarrow h \mid y \Rightarrow h'$ expects e to reduce to an expression of the form $E\, v$. At runtime, if E and E' are bound to the same exception label, then the branch h is selected with x bound to v, otherwise the branch h' is selected with y bound to $E\, v$. Our syntax allows the same exception name E to be introduced in different parts of a program with different purposes. It is therefore important that the comparison be at the level of exception labels and that each occurrence

[1] Variant types do not exist at runtime. The constructor C is represented as a tag that, statically, is used to indicate the expected type of its argument, and that, at runtime, is used to indicate the correct pattern in a **match**.
[2] Exception labels are used only in the definition of the semantics (§ 2.2); they are not part of the surface language (as indicated by the different color).

of E be bound to a fresh label. Finally, we note that it is easy to define a construct for catching a specific exception as syntactic sugar (similar to Pierce [48, §14.3, Ex. 4]):

$$\mathbf{try}\ e\ \mathbf{with}\ E\ x \Rightarrow h \ \triangleq\ \mathbf{try}\ e\ \mathbf{with}\ x \Rightarrow (\mathbf{match}\ x\ \mathbf{with}\ E\ x \Rightarrow h \mid y \Rightarrow \mathbf{raise}\ y) \qquad (1)$$

Absence of common constructs. Although, for brevity, the presented syntax omits usual constructs such as integers, Booleans, pairs, usual unary and binary operations, **if** statements, and **let** bindings, they are all part of our Rocq mechanization [5].

2.2 Operational Semantics

Figure 1b shows the definition of the semantics of ExceptionLang. The semantics is captured by a small-step *thread-pool reduction* $\rightarrow_{\mathrm{tp}}$, a relation between two pairs (of the same type) consisting of a *thread pool* $\vec{e_1}$, simply formalized as a list of expressions, and a *heap* s, formalized as a finite map where keys can be either *memory locations* ℓ that map to values, or exception labels l that map to a reserved symbol ‡. Intuitively, the assertion $(\vec{e_1}, s_1) \rightarrow_{\mathrm{tp}} (\vec{e_2}, s_2)$ means that one of the expressions in $\vec{e_1}$ takes one step changing the state from s_1 to s_2. This is captured by Rule TP-Step, which stratifies the construction of $\rightarrow_{\mathrm{tp}}$ into the construction of the *thread-local reduction* $\rightarrow_{\mathrm{t}}$, a relation formalizing the reduction of a single expression. Specifically, the thread-local reduction $\rightarrow_{\mathrm{t}}$ is a relation between (1) a pair of an expression and a heap and (2) a tuple of an expression, a heap, and a list of expressions. Intuitively, the assertion $(e_1, s_1) \rightarrow_{\mathrm{t}} (e_2, s_2, \vec{e})$ means that e_1 reduces to e_2 in one step, updating the state from s_1 to s_2, and spawning new threads initialized with each of the expressions in $\vec{e}$. The construction of such assertions is stratified by Rule Bind-Step, which allows one to focus on a subexpression e of $K[e]$, or by Rule Pure-Step, which allows one to ignore the state s in case of a *pure step* $\rightarrow_{\mathrm{p}}$. Some of the base cases of $\rightarrow_{\mathrm{p}}$ and $\rightarrow_{\mathrm{t}}$ are shown in Figure 1b. They rephrase, mathematically, our informal discussion from §2.1 about the constructs of ExceptionLang. The condition *neutral* K means that no exception handler ($\mathbf{try}\ K'\ \mathbf{with}\ x \Rightarrow e$) occurs in K, and ensures that exceptions are always caught by the nearest handler. The notation $\{l\} \uplus s$ corresponds to the insertion of the key-value binding $(l, ‡)$ in s plus the condition that l is not a key of s.

2.3 Type System

The formal syntax of types and exception rows is defined in Figure 2a.[3] Recall that a row ρ intuitively denotes a set of exceptions and that, syntactically, it can either be empty $\langle\rangle$; a singleton E; a row variable θ; the top row $\top$; or the union $\rho_1 \cdot \rho_2$ of ρ_1 and ρ_2. An alternative (equivalent) syntactic view of a row ρ is as a pair of a set of exception names $\mathit{Exns}\,\rho$ and a set of row variables $\mathit{RVars}\,\rho$. Such a view can be obtained by *flattening* unions in ρ. This alternative view of rows is used in the subtyping rules (Figure 2c), which we discuss soon.

A selection of typing rules is shown in Figure 2b. We selected rules where rows play a non-trivial role. A typing judgment has the shape $\Gamma \vdash e : \rho : \tau$, where Γ is a *typing context* that maps variables x and exception names E to types. Its informal meaning is that e either diverges, or returns an output of type τ, or raises an exception included in ρ.

Rule Forall-Row-Intro introduces a row-polymorphic type $\forall\theta.\ \tau$. The rule applies only to values. This restriction is known as the *value restriction*, which is necessary in the presence of both polymorphism and mutable references [62].[4] Because values do not raise exceptions, the row is $\langle\rangle$.

Rule Rec-Typed shows that a function $\mathbf{rec}\ f\ x = e$ has type $\tau \rightarrow_\rho \sigma$ if its body e has type σ and row ρ in an extended typing context where x has type τ and f has type $\tau \rightarrow_\rho \sigma$.

[3]See our Rocq mechanization [5] for omitted types such as integers, references, value-polymorphism and recursive types.
[4]In Rocq, we formalize the value restriction as a separate typing judgment.

$$\tau ::= \textbf{unit} \mid \tau \to_\rho \sigma \mid \overline{C \text{ of } \tau} \mid \forall\theta.\,\tau \mid \textbf{exn}_\rho \mid \ldots \qquad \rho ::= \langle\rangle \mid \theta \mid \top \mid E \mid \rho \cdot \rho$$

(a) Types and rows.

$$\textsc{Forall-Row-Intro}\ \frac{\Gamma \vdash e : \langle\rangle : \tau \qquad \textit{value}\ e}{\Gamma \vdash e : \langle\rangle : \forall\theta.\,\tau} \qquad \textsc{Rec-Typed}\ \frac{(f : \tau \to_\rho \sigma), (x : \tau), \Gamma \vdash e : \rho : \sigma}{\Gamma \vdash \textbf{rec}\ f\ x = e : \langle\rangle : \tau \to_\rho \sigma}$$

$$\textsc{Let-Exn-Typed}\ \frac{E \text{ fresh in } \Gamma, \rho, \tau \qquad (E : \sigma), \Gamma \vdash e : \rho : \tau}{\Gamma \vdash \textbf{let exn}\ E\ \textbf{in}\ e : \rho : \tau} \qquad \textsc{Exn-Typed}\ \frac{(E : \sigma) \in \Gamma \qquad \Gamma \vdash e : \rho : \sigma}{\Gamma \vdash E\ e : \rho : \textbf{exn}_E} \qquad \textsc{Raise-Typed}\ \frac{\Gamma \vdash e : \rho : \textbf{exn}_\rho}{\Gamma \vdash \textbf{raise}\ e : \rho : \tau}$$

$$\textsc{Match-Exn-Typed}\ \frac{(E : \sigma) \in \Gamma \qquad \Gamma \vdash e : \rho : \textbf{exn}_{E \cdot \rho'} \qquad (x : \sigma), \Gamma \vdash h : \rho : \tau \qquad (y : \textbf{exn}_{\rho'}), \Gamma \vdash h' : \rho : \tau}{\Gamma \vdash (\textbf{match}\ e\ \textbf{with}\ E\ x \Rightarrow h \mid y \Rightarrow h') : \rho : \tau}$$

$$\textsc{Try-Typed}\ \frac{\Gamma \vdash e : \rho : \tau \qquad (x : \textbf{exn}_\rho), \Gamma \vdash h : \rho' : \tau}{\Gamma \vdash \textbf{try}\ e\ \textbf{with}\ x \Rightarrow h : \rho' : \tau} \qquad \textsc{Sub-Typed}\ \frac{\Gamma \vdash e : \rho : \tau \qquad \rho <: \rho' \qquad \tau <: \tau'}{\Gamma \vdash e : \rho' : \tau'}$$

(b) Typing rules.

$$\textsc{Row-Sub}\ \frac{\textit{Exns}\ \rho \subseteq \textit{Exns}\ \rho' \qquad \textit{RVars}\ \rho \subseteq \textit{RVars}\ \rho'}{\rho <: \rho'} \qquad \textsc{Arrow-Sub}\ \frac{\tau' <: \tau \qquad \rho <: \rho' \qquad \sigma <: \sigma'}{\tau \to_\rho \sigma <: \tau' \to_{\rho'} \sigma'}$$

$$\textsc{Exn-Sub}\ \frac{\rho <: \rho'}{\textbf{exn}_\rho <: \textbf{exn}_{\rho'}} \qquad \textsc{Forall-Row-Elim-Sub}\ \ \forall\theta.\,\tau <: \tau[\rho/\theta] \qquad \textsc{Forall-Row-Intro-Sub}\ \frac{\tau <: \sigma \qquad \theta \notin \tau}{\tau <: \forall\theta.\,\sigma}$$

(c) Subtyping rules.

Fig. 2. Type system of ExceptionLang.

Rule Let-Exn-Typed extends the typing context with the exception name E at an arbitrary type σ. We call σ the *type of* E. This type is used to constrain the type of e' in an exception value $E\ e'$. The rule requires E to be *fresh* to avoid clashes of exception names.

Rule Exn-Typed places two conditions to the typing of an exception value $E\ e$: the exception name E must be present in the typing context and e must have the type σ to which E is bound. The concluding type $\textbf{exn}_E$ is the most precise, but, as we will see, it can be weakened by subtyping.

Rule Raise-Typed is relatively simple: $\textbf{raise}\ e$ can be typed at any type τ, provided e has type $\textbf{exn}_\rho$. Thanks to this type, only the constructors in ρ could have been used to build the exception value e. It is therefore sound to assign the row ρ to $\textbf{raise}\ e$.

Rule Match-Exn-Typed shows the typing of an exception-matching construct. It assumes the scrutinee e is an exception value of type $\textbf{exn}_{E \cdot \rho'}$ and that the branch h can be typed in an extended context where x has type σ (the type of E). Crucially, the rule also assumes that h' can be typed in an extended context where y has type $\textbf{exn}_{\rho'}$. Intuitively, the typing of h' can exploit the fact that, if h' is selected, then the exception value returned by e could not have been constructed by E.

Rule Try-Typed shows that the type τ of an exception handler is the same as the type of its handlee e and the type of its handler branch h. Because every exception in e is captured by the exception handler, the handler raises exceptions only during the execution of h. This is why the concluding judgment has row ρ'. The typing of h can rely on a typing context extended with $(x : \mathbf{exn}_\rho)$, because every exception raised by e uses one of the constructors in ρ.

Rule Sub-Typed adds the possibility to weaken the type τ and the row ρ of a derived typing judgment to a type σ and row ρ' according to the subtyping relations $\rho <: \rho'$ and $\tau <: \sigma$. A selection of the subtyping rules appear in Figure 2c. Again, we select those in which rows are important. Rule Row-Sub rephrases the row subtyping relation $\rho <: \rho'$ as two set inclusions: $\mathit{Exns}\,\rho \subseteq \mathit{Exns}\,\rho'$ and $\mathit{RVars}\,\rho \subseteq \mathit{RVars}\,\rho'$. If $\top$ occurs in ρ', these inclusions are vacuously true. If $\top$ occurs in ρ, then these inclusions demand that $\top$ occur in ρ' as well. Rule Arrow-Sub adapts the standard subtyping of arrow types to include rows. Rule Exn-Sub allows one to loose the information of the constructors used to build an exception value. Finally, Rules Forall-Row-Elim-Sub and Forall-Row-Intro-Sub are the elimination and introduction subtyping rules for polymorphic types.

The following typing rule for the construct defined in Equation (1) can be derived:

$$\textsc{Try-Exn-Typed}\;\frac{(E : \sigma) \in \Gamma \qquad \Gamma \vdash e : E \cdot \rho : \tau \qquad (x : \sigma), \Gamma \vdash h : \rho : \tau}{\Gamma \vdash \mathbf{try}\; e \;\mathbf{with}\; E\,x \Rightarrow h : \rho : \tau}$$

2.4 Backwards Compatibility of ML Typing Rules

We derive the ordinary typing rules of ML, where we define the exception type as $\mathbf{exn} \triangleq \mathbf{exn}_\top$, the function type as $\tau \to \sigma \triangleq \tau \to_\top \sigma$ and the typing judgment as $\Gamma \vdash e : \tau \triangleq \Gamma \vdash e : \top : \tau$. In most of our rules, the typing judgments all have the same row, so the derivation of the ML typing rule with $\top$ is direct. Let us consider a rule where also subsumption is needed:

$$\textsc{Match-Exn-ML-Typed}\;\frac{(E : \sigma) \in \Gamma \qquad \Gamma \vdash e : \mathbf{exn} \qquad (x : \sigma), \Gamma \vdash h : \tau \qquad (y : \mathbf{exn}), \Gamma \vdash h' : \tau}{\Gamma \vdash \mathbf{match}\; e \;\mathbf{with}\; E\,x \Rightarrow h \mid y \Rightarrow h' : \tau}$$

To derive this rule from Match-Exn-Typed, we use Sub-Typed on $\Gamma \vdash e : \mathbf{exn}$ and have to show that $\top <: E \cdot \top$. Recall that this subtyping holds vacuously because the row on the right includes $\top$.

2.5 Backwards Compatibility via Subtyping and Other Examples

Recall the definition and signature of **with_return** from §1:

$$\mathbf{with\_return} : \forall \theta, \alpha.\, (\forall \theta'.\, (\alpha \to_{\theta'} \bot) \to_{\theta \cdot \theta'} \alpha) \to_\theta \alpha$$
$$\mathbf{with\_return}\; f \triangleq \mathbf{let\;exn}\; \mathit{Return}\; \mathbf{in\;try}\; f\; (\lambda x.\, \mathbf{raise}\; (\mathit{Return}\; x))\; \mathbf{with}\; \mathit{Return}\; x \Rightarrow x$$

A call to **with_return** f, executes f applied to a *return* function. Consequently, f can terminate (1) with a *return* x call, (2) normally, or (3) by raising an exception. Raised exceptions by f are not caught by **with_return** as it only handles the local *Return* exception, which is used to implement the *return* call. The rank-2 row quantifier θ' in the type of **with_return** hides the use of a local exception and allows f to only raise it via a call to the *return* function.

To derive **with_return**'s ordinary type $\forall \alpha.\, ((\alpha \to \bot) \to \alpha) \to \alpha$, we use Sub-Typed and should show the subtyping $(\forall \theta.\, \forall \alpha.\, (\forall \theta'.\, (\alpha \to_{\theta'} \bot) \to_{\theta \cdot \theta'} \alpha) \to_\theta \alpha) <: (\forall \alpha.\, ((\alpha \to \bot) \to \alpha) \to \alpha)$. We first apply Forall-Row-Elim-Sub to instantiate θ with $\top$, followed by a congruence rule for type polymorphism and Arrow-Sub. Next, we show $((\alpha \to \bot) \to \alpha) <: (\forall \theta'.\, (\alpha \to_{\theta'} \bot) \to_{\top \cdot \theta'} \alpha)$, where the direction has swapped due to the contravariant nature of Arrow-Sub. We now apply Forall-Row-Intro-Sub to introduce the universal quantifier on the right. The rest of the derivation boils down to showing $\theta' <: \top$ and $\top <: \top \cdot \theta'$, which hold vacously because $\top$ appears on the right.

We implement a version of OCaml's **Domain** library [35] as follows:

$$\texttt{Domain.spawn} : \forall\theta.\,\forall\alpha.\,(\texttt{unit} \to_\theta \alpha) \to_{\langle\rangle} (\texttt{unit} \to_\theta \alpha)$$
$$\texttt{Domain.spawn}\ f \triangleq \texttt{let}\ r = \texttt{ref Wait in fork}\ \{\texttt{try}\ r \leftarrow \texttt{Val}\ (f\ ())\ \texttt{with}\ ex \Rightarrow r \leftarrow \texttt{Exn}\ ex\};\ \lambda\_.\ \texttt{Domain.join}\ r$$

$$\texttt{Domain.join} : \forall\theta.\,\forall\alpha.\ \texttt{ref}\ \{\texttt{Wait} \mid \texttt{Val of}\ \alpha \to_\theta \alpha \mid \texttt{Exn of exn}_\theta\}$$
$$\texttt{Domain.join}\ r \triangleq \texttt{match}\ !r\ \texttt{with Wait} \Rightarrow \texttt{Domain.join}\ r \mid \texttt{Val}\ res \Rightarrow res \mid \texttt{Exn}\ ex \Rightarrow \texttt{raise}\ ex$$

The function **Domain.spawn** takes a closure, runs it in a new thread, and returns a *join handler h*. Calling h () joins the thread and returns its result or re-raises any exception raised by f (). This behavior is evident from its type: the returned join handler is annotated with the same row θ as the given closure. The derivation of the ordinary ML type $\forall\alpha.\ (\texttt{unit} \to \alpha) \to (\texttt{unit} \to \alpha)$ follows by instantiating θ with $\top$ via Forall-Row-Elim-Sub and Arrow-Sub.

Internally, **Domain.spawn** uses the native **fork** of ExceptionLang to run f () in a new thread. The reference r is used to track whether computation is pending (**Wait**), has terminated normally (**Val**), or with an exception (**Exn**). The returned join handler uses the internal function **Domain.join** to perform a busy loop until the computation finished (is no longer **Wait**). To type check this implementation, it is crucial that the exception type $\texttt{exn}_\theta$ is annotated with the row θ.

We implement a version of OCaml's **Lazy** library [37] as follows:

$$\texttt{Lazy.from\_fun} : \forall\theta.\,\forall\alpha.\,(\texttt{unit} \to_\theta \alpha) \to_{\langle\rangle} (\texttt{unit} \to_\theta \alpha)$$
$$\texttt{Lazy.from\_fun}\ f \triangleq \texttt{let}\ r = \texttt{ref Wait in}\ \lambda\_.\ \texttt{Lazy.force}\ f\ r$$

$$\texttt{Lazy.force} : \forall\theta.\,\forall\alpha.\,(\texttt{unit} \to_\theta \alpha) \to_{\langle\rangle} \texttt{ref}\ \{\texttt{Wait} \mid \texttt{Lock} \mid \texttt{Val of}\ \alpha \mid \texttt{Exn of exn}_\theta\} \to_\theta \alpha$$
$$\begin{aligned}\texttt{Lazy.force}\ f\ r \triangleq\ & \texttt{match}\ !r\ \texttt{with} \\ & \mid \texttt{Wait} \Rightarrow \texttt{if not (cas}\ r\ \texttt{Wait Lock) then Lazy.force}\ f\ r \\ & \qquad \texttt{else try let}\ res = f\ ()\ \texttt{in}\ r \leftarrow \texttt{Val}\ res;\ res \\ & \qquad\quad \texttt{with}\ ex \Rightarrow r \leftarrow \texttt{Exn}\ ex;\ \texttt{raise}\ ex \\ & \mid \texttt{Lock} \Rightarrow \texttt{Lazy.force}\ f\ r \mid \texttt{Exn}\ ex \Rightarrow \texttt{raise}\ ex \mid \texttt{Val}\ res \Rightarrow res\end{aligned}$$

The function **Lazy.from_fun** has the same type as **Domain.spawn** and a similar behavior. The key differences is that the computation f () is not performed in a new thread, but upon the first call to the returned *suspension h*. Also similar to **Domain.spawn**, if f () raises an exception, then every call to h () will raise that same exception, as evident from the row θ in the signature. The implementation uses an internal reference r to track the computation's state. To ensure that the computation f () is performed at most once, we add a **Lock** state. We use the atomic compare-and-set operation **cas** to ensure that only a single thread can move from the **Wait** to the **Lock** state, and perform f (). Other threads perform a busy loop until the result is stored (is no longer **Lock**).

We implement a version of OCaml's **Fun.protect** function [36] as follows:

$$\texttt{Fun.protect} : \forall\theta.\,\forall\alpha.\,(\texttt{unit} \to_{\langle\rangle} \texttt{unit}) \to_{\langle\rangle} (\texttt{unit} \to_\theta \alpha) \to_\theta \alpha$$
$$\texttt{Fun.protect}\ \mathit{finally}\ \mathit{work} \triangleq \texttt{let}\ y = (\texttt{try}\ \mathit{work}\ ()\ \texttt{with}\ x \Rightarrow \mathit{finally}\ ();\ \texttt{raise}\ x)\ \texttt{in}\ \mathit{finally}\ ();\ y$$

A call to **Fun.protect** *finally work* first performs *work* (), and then performs *finally* () regardless of whether *work* () returned normally or raised an exception. OCaml considers exceptions raised by *finally* a programming error, and performs a runtime check to handle these in a special way. We do not implement such a runtime check and instead rule out these cases statically by restricting the row of *finally* to $\langle\rangle$. To see **Fun.protect** in action, we implement a spin-lock:

$$\texttt{new\_lock} : \texttt{unit} \to_{\langle\rangle} \forall\theta.\,\forall\alpha.\,(\texttt{unit} \to_\theta \alpha) \to_\theta \alpha$$
$$\texttt{new\_lock}\ () \triangleq \texttt{let}\ r = \texttt{ref false in}\ \lambda f.\ \texttt{acquire}\ r;\ \texttt{Fun.protect}\ (\lambda\_.\ \texttt{release}\ r)\ f$$

A call to **new_lock** () returns a lock lk. Calling $lk\ f$ executes f () inside a critical section. The lock is implemented using a reference r that is **false** (unlocked) or **true** (locked). The internal function **acquire** performs a busy loop until it can update r from **false** to **true**, and **release** resets r to **false**. We wrap the call to f inside **Fun.protect** to prevent deadlocks when f raises an exception.

Finally, we consider three implementations of a module for dynamically extensible variant types:

$$\mathbf{new\_dyn}_{\mathsf{exn}}, \mathbf{new\_dyn}_{\mathsf{ref}} : \mathbf{unit} \rightarrow_{\langle\rangle} \exists \alpha.\, \forall \beta.\, \mathbf{unit} \rightarrow_{\langle\rangle} (\beta \rightarrow_{\langle\rangle} \alpha) \times (\alpha \rightarrow_{\langle\rangle} \mathbf{option}\ \beta)$$

$$\begin{aligned}
\mathbf{new_dyn}_{\mathsf{exn}}\ () &\triangleq \mathbf{pack}\ \langle \mathbf{exn}_\top \rangle\ (\lambda_.\ \mathbf{let\ exn}\ E\ \mathbf{in} \\
&\qquad ((\lambda x.\ E\ x), (\lambda d.\ \mathbf{match}\ d\ \mathbf{with}\ E\ x \Rightarrow \mathbf{Some}\ x \mid _ \Rightarrow \mathbf{None}))) \\
\mathbf{new_dyn}_{\mathsf{ref}}\ () &\triangleq \mathbf{let}\ lk = \mathbf{new_lock}\ ()\ \mathbf{in\ pack}\ \langle \mathbf{unit} \rightarrow_\top \mathbf{unit} \rangle\ (\lambda_.\ \mathbf{let}\ r = \mathbf{ref\ None\ in} \\
&\qquad ((\lambda x.\ \lambda_.\ r \leftarrow \mathbf{Some}\ x), (\lambda d.\ lk\ (\lambda_.\ r \leftarrow \mathbf{None};\ d\ ();\ !r)))) \\
\mathbf{new_dyn}_{\mathsf{cntr}}\ () &\triangleq \mathbf{let}\ c = \mathbf{ref}\ 0\ \mathbf{in\ pack}\ \langle \mathbf{int} \times \mathbf{Obj.t} \rangle\ (\lambda_.\ \mathbf{let}\ i = \mathbf{FAA}\ c\ 1\ \mathbf{in} \\
&\qquad ((\lambda x.\ (i, x)), (\lambda (j, x).\ \mathbf{if}\ i = j\ \mathbf{then\ Some}\ x\ \mathbf{else\ None})))
\end{aligned}$$

The implementations $\mathbf{new\_dyn}_{\mathsf{exn}}$ and $\mathbf{new\_dyn}_{\mathsf{ref}}$ are taken from Filinski [17, §4.5, Remark 4.30], while $\mathbf{new\_dyn}_{\mathsf{cntr}}$ is an *unsafe* implementation we wrote ourselves. All of these three functions generate a new dynamically extensible variant module, modeled as a closure that can be used to install new *constructor-destructor* pairs (c, d). The destructor d is akin to a **match**. That is, a call to $d\ y$ returns **Some** x if x was obtained through the corresponding constructor $c\ x$, and returns **None** when the x has been obtained via a different constructor. For example:

$$\begin{aligned}
&\mathbf{let}\ dyn = \mathbf{new_dyn}_{\mathsf{exn}}\ ()\ \mathbf{in\ let}\ (c_1, d_1) = dyn\langle \mathbf{int} \rangle\ ()\ \mathbf{in\ let}\ (c_2, d_2) = dyn\langle \mathbf{bool} \rangle\ ()\ \mathbf{in} \\
&d_1\ (c_2\ \mathbf{true})\ (\text{* returns } \mathbf{None} \text{ *});\ d_1\ (c_1\ 37)\ (\text{* returns } \mathbf{Some}\ 37 \text{ *})
\end{aligned}$$

The implementation $\mathbf{new\_dyn}_{\mathsf{exn}}$ uses the **exn** type to represent variant constructors, and implements the destructor using **match**. The implementation $\mathbf{new\_dyn}_{\mathsf{ref}}$ uses a local reference r for each variant constructor. The constructor c returns a closure that sets the reference r to the argument value, and the destructor calls that closure and reads the reference r. If the reference contains **None** it means the constructor of a different variant was used. Compared to Filinski [17] we consider a concurrent language and need to wrap the destructor in a lock to ensure thread safety. Finally, $\mathbf{new\_dyn}_{\mathsf{cntr}}$ represents variant constructors as a pairs of an integer tag (which we atomically increment for each new constructor) and the constructor argument. This implementation is not statically typeable in our language and requires the unsafe **Obj** module in OCaml [38]. Even though $\mathbf{new\_dyn}_{\mathsf{cntr}}$ is not typeable, one can reason about it semantically using logical relations [24, 56], as shown in §5.

2.6 Safety and Abstraction Guarantees

Safety. A program is *safe* if it either diverges or terminates with a result that is either a value or an unhandled exception (that is, a **raise** expression). A program is *exception safe*, if, on top of being type safe, it does not raise unhandled exceptions. A well-typed program is always type safe, and, if it type checks at an empty row, it is also exception safe:

Theorem 2.1 (Safety). *If* $\vdash e : \rho : \tau$, *then* e *is safe, and, if* $\rho = \langle\rangle$, *then* e *is also exception safe.*

Abstraction. A key abstraction property enjoyed by well-typed programs is representation independence: changing the representation of a type does not affect programs that use this type abstractly. The main ingredient in formulating such abstraction properties is *contextual equivalence.* Informally, a program e_1 is *contextually equivalent* to e_2, if, for any program e, replacing every occurrence of e_2 with e_1 in e does not change the behavior of e. Formally, contextual equivalence (and contextual refinement) is defined as follows:

$$\Gamma \vDash e_1 \leq_{\text{ctx}} e_2 : \tau : \rho \triangleq \forall (C : (\Gamma : \rho : \tau) \rightsquigarrow (\emptyset : \langle\rangle : \texttt{unit})), s.\ (C[e_1], s) \downarrow \implies (C[e_2], s) \downarrow$$

$$\Gamma \vDash e_1 =_{\text{ctx}} e_2 : \rho : \tau \triangleq \Gamma \vDash e_1 \leq_{\text{ctx}} e_2 : \rho : \tau \wedge \Gamma \vDash e_2 \leq_{\text{ctx}} e_1 : \rho : \tau$$

In words, an expression e_1 contextually refines e_2 if, for any *typed program context* C and heap s, the termination of $(C[e_1], s)$ implies the termination of $(C[e_2], s)$. Termination is captured by the assertion $(e_1, s) \downarrow$, stating that the main thread e_1 terminates to either a value or a **raise** expression. The relation $C : (\Gamma : \rho : \tau) \rightsquigarrow (\Gamma' : \rho' : \sigma)$ expresses that, for any e, if the judgment $\Gamma \vdash e : \rho : \tau$ is derivable, then $\Gamma \vdash C[e] : \rho' : \sigma$ is derivable. Program contexts C are more general than evaluation contexts K, because the hole $[]$ in C does not need to appear in evaluation position.

In §5 we show that our type system (and by backwards compatibility, ordinary ML) satisfies many interesting equivalences and refinements. We first show simple equations indicating that local exceptions are private, before proving refinements between the examples in this section. Due to the quantification over *all* program contexts C, contextual refinement is, however, very hard to prove. As is standard in the literature, we prove refinements indirectly via logical relations (§4), which we define by means of our relational separation logic for exceptions (§3).

3 Relational Separation Logic for Exceptions

We present the refinement judgment and adequacy theorem (§3.1) and proof rules (§3.2) of our relational logic for exceptions. We conclude with the support for concurrency (§3.3).

3.1 Refinement Judgment and Adequacy

The *refinement judgment* $e_1 \precsim e_2 \{\Phi\}$ is the mechanism through which the behaviors of programs e_1 and e_2 can be compared. In its most general form, the *postcondition* Φ is a binary relation on expressions. The judgment can then be intuitively read as asserting that either (1) the expression e_1 diverges or (2) both e_1 and e_2 respectively reduce to expressions e_1' and e_2' such that $\Phi\, e_1'\, e_2'$ holds.

Our general type of postconditions allows a kind of *asymmetric context-local reasoning*, which we explain during the discussion of the *bind rule* (§3.2). However, it is often the case that, when stating the refinement between two programs, we would like the postcondition to express a relation between the *results* of these programs rather than arbitrary expressions. In a language with exceptions, the result of a program is either a value or a **raise** expression. Precisely because the type of our postconditions is so general, it is easy to define specialized forms of the judgment:

$$e_1 \precsim e_2 \{r_1, r_2.\, R\} \quad \triangleq \quad e_1 \precsim e_2 \left\{e_1', e_2'.\ e_1', e_2' \in \mathit{Result} \wedge R[e_1'/r_1, e_2'/r_2]\right\}$$

$$e_1 \precsim e_2 \left\{\Phi_E \,\middle|\, \Phi_V\right\} \quad \triangleq \quad e_1 \precsim e_2 \left\{ r_1, r_2.\ \vee \begin{cases} \exists v_1, v_2.\ r_1 = v_1 \wedge r_2 = v_2 \wedge \Phi_V\, v_1\, v_2 \\ \exists l_1, l_2, v_1, v_2.\ \wedge \begin{cases} r_1 = \texttt{raise}\ (l_1\, v_1) \\ r_2 = \texttt{raise}\ (l_2\, v_2) \\ \Phi_E\, l_1\, l_2\, v_1\, v_2 \end{cases} \end{cases} \right\}$$

The specialized form $e_1 \precsim e_2 \{r_1, r_2.\, R\}$ restricts the postcondition to a binary relation on expressions of type $\mathit{Result} \triangleq \mathit{Val} \cup \{\texttt{raise}\ v \mid v \in \mathit{Val}\}$. The binders r_1 and r_2, which we use exclusively to denote results, disambiguate between the general refinement judgment and this specialized form.

The specialized form $e_1 \precsim e_2 \left\{\Phi_E \,\middle|\, \Phi_V\right\}$ intuitively states that either (1) e_1 diverges or (2) both e_1 and e_2 return values related by Φ_V or (3) both e_1 and e_2 reduce to **raise** expressions related by Φ_E.

Model. The definition of the refinement judgment $e_1 \precsim e_2 \{\Phi\}$ is a generalization of the definition of *observational refinement* from ReLoC [19]. The key difference is that we use an *extended weakest precondition* with support for exceptions and an expression postcondition. For concision, we omit

$$\textsc{Step-L}\ \frac{e_1 \rightarrow_{\mathrm{p}} e_1' \qquad \triangleright\, e_1' \precsim e_2\ \{\Phi\}}{e_1 \precsim e_2\ \{\Phi\}}* \qquad \textsc{Step-R}\ \frac{e_2 \rightarrow_{\mathrm{p}} e_2' \qquad e_1 \precsim e_2'\ \{\Phi\}}{e_1 \precsim e_2\ \{\Phi\}}* \qquad \textsc{Base}\ \frac{\Phi\, e_1\, e_2}{e_1 \precsim e_2\ \{\Phi\}}*$$

$$\textsc{Wand}\ \frac{e_1 \precsim e_2\ \{\Phi\} \qquad \forall e_1', e_2'.\ \Phi\, e_1'\, e_2' \mathrel{-\!\!*} \Phi'\, e_1'\, e_2'}{e_1 \precsim e_2\ \{\Phi'\}}* \qquad \textsc{Bind}\ \frac{e_1 \precsim e_2\ \left\{e_1', e_2'.\ K_1[e_1'] \precsim K_2[e_2']\ \{\Phi\}\right\}}{K_1[e_1] \precsim K_2[e_2]\ \{\Phi\}}*$$

$$\textsc{Let-Exn-L}\ \frac{\forall l_1.\ exnLbl_{\mathrm{i}}\ l_1 \mathrel{-\!\!*} \triangleright\, e_1[l_1/E] \precsim e_2\ \{\Phi\}}{\texttt{let exn}\ E\ \texttt{in}\ e_1 \precsim e_2\ \{\Phi\}}* \qquad \textsc{Let-Exn-R}\ \frac{\forall l_2.\ exnLbl_{\mathrm{s}}\ l_2 \mathrel{-\!\!*} e_1 \precsim e_2[l_2/E]\ \{\Phi\}}{e_1 \precsim \texttt{let exn}\ E\ \texttt{in}\ e_2\ \{\Phi\}}*$$

Fig. 3. Reasoning rules.

the formal definition of refinement and refer the interested reader to the Rocq mechanization [5]. Using the model we obtain the adequacy theorem, which shows the logic is sound:

Theorem 3.1 (Adequacy). *If* $\vdash e_1 \precsim e_2\ \{\phi\}$*, then (1)* e_1 *is safe, (2) if* $(e_1, s)\downarrow$ *then* $(e_2, s)\downarrow$*, and (3) if, for every* r_1 *and* r_2*, the assertion* $\phi\, r_1\, r_2$ *implies* $r_1 \in Val$*, then* e_1 *is exception safe.*

3.2 Reasoning Rules

Figure 3 shows the reasoning rules of the logic. We omit the rules for reasoning about memory accesses (such as, load, store, and **cas**), which are standard in relational separation logic. Our rules for reasoning about concurrency constructs are less standard. We present them in detail in §3.3.

Rules Step-L and Step-R enable the expressions in either side of the refinement to be partially executed, provided these executions are pure.[5]

Rule Base lets one terminate a proof of refinement whenever the expressions e_1 and e_2 satisfy Φ. It is used to reason about expressions that return a value or raise an exception. For example, consider the following *derived* rules $\Phi_V\, v_1\, v_2 \ \vdash\ v_1 \precsim v_2\ \left\{\Phi_E \,\middle|\, \Phi_V\right\}$ and $\Phi_E\, l_1\, l_2\, v_1\, v_2\ \vdash$ $\texttt{raise}\ (l_1\, v_1) \precsim \texttt{raise}\ (l_2\, v_2)\ \left\{\Phi_E \,\middle|\, \Phi_V\right\}$. Both rules are straightforward applications of Base.

Reading from top to bottom, Rule Wand allows the postcondition Φ in a refinement judgment to be weakened to Φ'. The weakening condition is expressed as $\forall e_1', e_2'.\ \Phi\, e_1'\, e_2' \mathrel{-\!\!*} \Phi'\, e_1'\, e_2'$. Intuitively, Rule Wand captures the principle that resources available before the execution of a program are still available afterwards provided the program does not interfere with these resources. This principle is more clearly visible in the *frame rule*, $R * e_1 \precsim e_2\ \{\Phi\} \vdash e_1 \precsim e_2\ \left\{e_1', e_2'.\ R * \Phi\, e_1'\, e_2'\right\}$, which can be stated as a *derived* rule proved by a straightforward application of Wand.

Rule Bind enforces *local-context reasoning*: it is sound to reason about the expressions e_1 and e_2 independently of the contexts in which they occur. There are two interesting aspects about Bind.

First, because the postcondition is not restricted to values (more generally, not even to results), when applying Bind, it is not needed that e_1 and e_2 terminate synchronously. This generalization allows a form of *asymmetric local-context reasoning*, whereby one can focus on a subexpression in the implementation side regardless of the specification side (and vice-versa). The following rule (and its specification-side counterpart) can be derived: $e_1 \precsim e_2\ \left\{e_1', e_2'.\ K[e_1'] \precsim e_2'\ \{\Phi\}\right\} \vdash K[e_1] \precsim e_2\ \{\Phi\}$.

[5] To reason about a recursive function f, Iris employs a reasoning principle (called *Löb's induction*) whereby f's own specification can be introduced as an assumption to reason about recursive uses of f. To avoid cyclic proofs, where the assumed specification of f is immediately used to prove itself, Iris guards the assumed specification by a *later modality* $\triangleright$ [7, 46]. To use the assumed specification, the modality must be eliminated. The $\triangleright$ in front of the refinement judgment in Step-L is one way in which $\triangleright$ can be eliminated from assumptions in the proof context. Intuitively, the presence of $\triangleright$ in Step-L (and its absence in Step-R) corresponds to the informal reading of the judgment, which allows e_1 to diverge regardless of e_2.

Second, unlike previous relational logics for non-local control effects where the bind rule is restricted to contexts K that do not contain handlers for the effects the subexpressions might perform, Rule Bind applies to any context, even to contexts that include **try-with** constructs. Indeed, the following rule (and its specification-side counterpart) can be derived:

$$\textsc{Try-With-L}\ \frac{e_1 \precsim e_2 \left\{e_1', e_2'.\ \left(e_1' \in Val \land e_1' \precsim e_2' \{\Phi\}\right) \lor \left(\exists v.\ e_1' = \mathbf{raise}\, v \land h_1[v/x] \precsim e_2' \{\Phi\}\right)\right\}}{\mathbf{try}\, e_1\, \mathbf{with}\, x \Rightarrow h_1 \precsim e_2 \{\Phi\}}*$$

Finally, Rule Let-Exn-L can be used to reason about the allocation of exceptions, which corresponds in the logic to the introduction of a resource $exnLbl_{\mathsf{i}}\ l$. Intuitively, $exnLbl_{\mathsf{i}}\ l$ asserts that l is fresh: $exnLbl_{\mathsf{i}}\ l * exnLbl_{\mathsf{i}}\ l' \vdash l \neq l'$. By default, $exnLbl_{\mathsf{i}}\ l$ is non-duplicable, but it can be updated to a persistent assertion $exnLbl_{\mathsf{i}}^{\square}\ l$, which can still be used for establishing freshness of exception labels: $exnLbl_{\mathsf{i}}\ l * exnLbl_{\mathsf{i}}^{\square}\ l' \vdash l \neq l'$. The reading of Rule Let-Exn-R and $exnLbl_{\mathsf{s}}$ is analogous.[6]

Example. We illustrate how our reasoning rules can be used to derive the following rule:

$$\textsc{With-Return-L}\ \frac{\forall return, l. \left(\begin{array}{l} exnLbl_{\mathsf{i}}\ l \twoheadrightarrow (\square\, \forall v, \Phi', e'.\mathbf{raise}(l\ v) \precsim e' \{\Phi'\} \twoheadrightarrow return\, v \precsim e' \{\Phi'\}) \twoheadrightarrow \\ main\ return \precsim e \left\{ r_1, r_2.\ \lor \left\{ \begin{array}{l} \exists v.\ r_1 = \mathbf{raise}\ (l\ v) \ \land \Phi\, v\, r_2 \\ (\forall v.\ r_1 \neq \mathbf{raise}\ (l\ v)) \land \Phi\, r_1\, r_2 \end{array} \right. \right\} \end{array} \right)}{\mathbf{with\_return}\ main \precsim e \{r_1, r_2.\ \Phi\, r_1\, r_2\}}$$

This rule allows one to reason about occurrences of **with_return** on the left side of the judgment.[7] Specifically, to reason about the application of **with_return** to a client function *main*, it suffices to reason about the application of *main* to a function *return*, which stands for the function in **with_return** that raises the locally declared exception *Return*. Rule With-Return-L exposes the label l to which *Return* is bound at runtime to assert its freshness using $exnLbl_{\mathsf{i}}\ l$. The postcondition for the refinement between *main return* and e reflects the behavior of the exception handler in the implementation of **with_return**: the case of a **raise** expression with label l is especially handled to allow the user to unwrap the payload v before proving the results are related by Φ. In addition to $exnLbl_{\mathsf{i}}\ l$, Rule With-Return-L also provides a specification of *return* that, in a sense, inlines the underlying definition of *return*: the application $return\, v$ refines any expression e' at any postcondition Φ' provided that **raise** $(l\ v)$ refines e' at Φ'. Iris's persistently modality $\square$ allows this specification to be used as many times as *return* occurs in the body of *main*.

The main rules used in the derivation of With-Return-L are Rule Let-Exn-L, to introduce l and $exnLbl_{\mathsf{i}}\ l$; and Rule Try-With-L, to reason about the exception handler installed by **with_return**.

3.3 Concurrency and Iris-Style Invariants

We explain how our relational logic supports concurrency by showing that **Lazy.from_fun** refines **Domain.spawn**, our versions of the OCaml libraries we discussed in §2.5.[8] To prove this refinement, we use Iris's *invariants* $\boxed{P}$ to assert that P holds at any moment. We show that, by leveraging the expressivity of postconditions ranging over expressions, we can state a single invariant-opening rule that applies to any atomic instruction. In contrast, prior relational logics such as ReLoC [18, 19] count with one invariant-opening rule *per* atomic instruction (see §6 for details).

[6] Under the hood, the *exnLbl* predicates are defined as points-to predicates with a discardable fractional permission [60].
[7] An analogous (derived) rule for occurrences of **with_return** on the right side of the judgment exists.
[8] We can prove a refinement in this direction because the execution of the source is angelic: we can execute the spawned source thread from **Domain.spawn** entirely at the moment the suspension is forced in **Lazy.from_fun**. The other direction does not hold because the scheduling of the target is demonic: the execution of the spawned thread could be interleaved arbitrarily, which is not possible for the computation of the suspension.

$$\textsc{Fork-R}\quad \frac{\forall j.\, j \Rrightarrow e_2 \twoheadrightarrow e_1 \precsim ()\ \{\Phi\}}{e_1 \precsim \texttt{fork}\,\{e_2\}\ \{\Phi\}}* \qquad \textsc{Inv-Open}\quad \frac{\text{atomic}(e_1) \quad \boxed{P} \quad \triangleright P \twoheadrightarrow e_1 \precsim e_2 \left\{e_1', e_2'.\, e_1' \in \mathit{Result} * \triangleright P * \Phi\, e_1'\, e_2'\right\}}{e_1 \precsim e_2\ \{\Phi\}}*$$

Fig. 4. Reasoning rules for concurrency.

We state the refinement between **Lazy.from_fun** and **Domain.spawn** as follows:[9]

$$\forall \Phi_E, \Phi_V, f_1, f_2.\ \left(\begin{array}{l} f_1\ () \precsim f_2\ () \left\{\Box\, \Phi_E \,\middle|\, \Box\, \Phi_V\right\} \twoheadrightarrow \\ \texttt{Lazy.from\_fun}\ f_1 \precsim \texttt{Domain.spawn}\ f_2 \left\{g_1, g_2.\ \Box\, g_1\ () \precsim g_2\ () \left\{\Box\, \Phi_E \,\middle|\, \Box\, \Phi_V\right\}\right\} \end{array} \right)$$

It is assumed that f_1 and f_2 either both raise an exception or both return a value. The refinement states that the functions g_1 and g_2 satisfy the same relational specification as f_1 and f_2: from a logical point of view, g_1 and g_2 behave like f_1 and f_2. This reading corresponds to the intuition that, under the hood, g_1 forces the execution of f_1 and g_2 reads the result of executing f_2. The occurrences of the $\Box$ modalities allow g_1 and g_2 to be called repeatedly and their results to be shared freely.

We now briefly explain the proof. After the allocation of a reference r_2 by **Domain.spawn**, we must reason about a **fork** instruction. To this end, we use Rule Fork-R, which lets us introduce a *ghost thread-pool assertion* [58] to represent the newly spawned thread with a fresh id j running the exception handler: $j \Rrightarrow (\texttt{try}\ r_2 \leftarrow \texttt{Val}\ (f_2\ ())\ \texttt{with}\ ex \Rightarrow r_2 \leftarrow \texttt{Exn}\ ex)$. After the allocation of r_1 by **Lazy.from_fun**, both **Lazy.from_fun** and **Domain.spawn** reach a point in which they return a value. Rule Base can then be used to reduce the proof to:

$$\Box(\lambda\_.\, \texttt{Lazy.force}\ f_1\ r_1)\ () \precsim (\lambda\_.\, \texttt{Domain.join}\ r_2)\ () \left\{\Box\, \Phi_E \,\middle|\, \Box\, \Phi_V\right\}$$

The modality $\Box$ prevents us from keeping exclusive ownership of resources (in particular, the points-to assertions and the ghost thread-pool assertion). This is problematic because **Lazy.force**, for example, accesses the location r_1. Fortunately, it is possible to express a relation among the contents of r_1 and r_2, and the state of the running thread j as an Iris assertion that holds at any moment. Iris allows us to express the property that it holds at any moment by means of the *invariant assertion* $\boxed{P}$. To introduce this assertion, the ownership of the resources in P is relinquished. In exchange, because $\boxed{P}$ asserts only the *knowledge* that P holds at any point, it does not need to be exclusively owned: it is in fact persistent and can be freely shared.

The relation among r_1, r_2, and j is formalized as the assertion Inv shown in Figure 5. It represents the state system informally discussed in §2.5 as a disjunction of four cases. Case (1) is a snapshot of the current resources, case (2) asserts that someone currently owns the lock and is executing f_1, f_2, and cases (3,4) state that the computations have terminated with either related exceptions or values. By relinquishing the resources we own, we can prove the first case, introduce $\boxed{\mathit{Inv}}$ and keep it even after removing the $\Box$ modality to simplify the goal to:

$$\texttt{Lazy.force}\ f_1\ r_1 \precsim \texttt{Domain.join}\ r_2 \left\{\Box\, \Phi_E \,\middle|\, \Box\, \Phi_V\right\}$$

To access r_1, **Lazy.force** must claim its ownership back from Inv. It is possible to reclaim ownership of the resources in Inv (or to *open* the invariant $\boxed{\mathit{Inv}}$) for the exact duration of a single execution step. This is sound because, as long as Inv is *preserved*, that is, as long as it holds at the beginning and at the end of each step, Inv holds at any point between two steps (as asserted by $\boxed{\mathit{Inv}}$). Rule Inv-Open captures this intuition precisely. Access to P is only granted for the duration of one step

[9] This can be seen as a *multishot* refinement where we can repeatedly call the returned closures. In our Rocq mechanization we have also proven a *oneshot* refinement, where all persistence modalities are removed.

$$Inv \triangleq \bigvee \begin{cases} \left(\begin{array}{l} r_1 \mapsto_{\mathsf{i}} \texttt{Wait} * r_2 \mapsto_{\mathsf{s}} \texttt{Wait} * f_1\,() \precsim f_2\,()\ \{\Box\,\Phi_E \mid \Box\,\Phi_V\} * \\ j \Rrightarrow (\texttt{try}\ r_2 \leftarrow \texttt{Val}\,(f_2\,())\ \texttt{with}\ ex \Rightarrow r_2 \leftarrow \texttt{Exn}\,ex) \end{array} \right) & (1) \\ (r_1 \mapsto_{\mathsf{i}}^{1/2} \texttt{Lock}) & (2) \\ (\exists l_1, l_2, v_1, v_2.\ r_1 \mapsto_{\mathsf{i}}^{\Box} \texttt{Exn}\,(l_1\,v_1) * r_2 \mapsto_{\mathsf{s}}^{\Box} \texttt{Exn}\,(l_2\,v_2) * \Box\,\Phi_E\ l_1\ l_2\ v_1\ v_2) & (3) \\ (\exists v_1, v_2.\ r_1 \mapsto_{\mathsf{i}}^{\Box} \texttt{Val}\,v_1 * r_2 \mapsto_{\mathsf{s}}^{\Box} \texttt{Val}\,v_1 * \Box\,\Phi_V\ v_1\ v_2) & (4) \end{cases}$$

Fig. 5. The invariant for the refinement between **Lazy.from_fun** and **Domain.spawn**.

by requiring the expression on the implementation side to be atomic.[10] If this expression is not atomic, the user can use Bind to focus on an atomic subexpression before applying Inv-Open. The statement, in a relational setting, of a single general invariant-opening rule applicable to any atomic expression is an original contribution of ours.

Using Rule Inv-Open, the rest of the proof is straightforward. If the **cas** operation succeeds, we can take out the resources of case (1) and execute f_1, f_2, using de Vilhena et al. [13]'s logical fork rule. Finally, we update the invariant to either case (3) or (4), depending on whether f_1, f_2 terminated with an exception or normally. Subsequent calls to g_1, g_2 are in case (3) or (4), and can duplicate the contents of the persistently stored postconditions to finish the proof.

4 Logical Relations

To formally justify the safety and abstraction guarantees discussed in §2.6, we develop a *logical interpretation* of the type system through *logical relations*, where types are interpreted as value relations and typing judgments as relational specifications (§4.1). Based on the *logical approach* [56], we define our logical relations in terms of our relational program logic. Consequently, our safety theorem as well as standard properties of logical relations follow from adequacy of our logic (§4.2). Moreover, the abstraction guarantees follow from properties of the interpretation of types (§5). For now, it suffices to know that the abstraction guarantees are formulated as assertions relating the execution of two programs, so it is key that our logical relations be *binary*. Finally, we note that our interpretation of the extensible variant type **exn** and the interpretation of rows make use of a novel theory of *exception-signature assertions* (§4.3), otherwise we follow the setup of Timany et al. [56].

4.1 Semantic Interpretation of the Type System

We interpret a typing judgment $\Gamma \vdash e : \rho : \tau$ as a *semantic typing judgment* $\Gamma \vDash e : \rho : \tau$ and the subtyping relations $\tau <: \tau'$ and $\rho <: \rho'$ respectively as *semantic subtyping judgments* $\tau \prec: \tau'$ and $\rho \prec: \rho'$. These semantic judgments are logical assertions defined in terms of the interpretation of types and rows. A type τ is interpreted as a *semantic type* $[\![\tau]\!]$, a binary value relation such that the assertion $[\![\tau]\!]\ v_1\ v_2$ is persistent for every v_1 and v_2. Intuitively, the relation $[\![\tau]\!]$ represents the results of two related expressions of type τ. The persistence condition allows values to be freely shared as permitted by the unrestricted nature of the type system. A row ρ is interpreted as a *semantic row* $[\![\rho]\!]$, a relation on a label pair and a value pair. Intuitively, the relation $[\![\rho]\!]$ describes the exceptions raised by two related expressions with row ρ.

Figure 6a defines the semantic judgments. We explain the semantic subtyping judgments when discussing the interpretation of types and rows. A semantic typing judgment $\Gamma \vDash e : \rho : \tau$ is defined as a *logical refinement* $\Gamma \vDash e \leq_{\log} e : \rho : \tau$. A logical refinement between two expressions e_1 and e_2

[10] For soundness, Iris also uses *invariant masks* $\mathcal{E}$ and *namespaces* $\mathcal{N}$ to prevent an invariant being opened twice in a nested fashion, which we omit for brevity.

$$
\begin{aligned}
\Gamma \vDash e : \rho : \tau &\triangleq \Gamma \vDash e \leq_{\log} e : \rho : \tau \\
\Gamma \vDash e_1 \leq_{\log} e_2 : \rho : \tau &\triangleq \forall \eta, \gamma_1, \gamma_2.\ [\![\Gamma]\!]_{\eta\,\gamma_1\,\gamma_2} \twoheadrightarrow \gamma_1\, e_1 \precsim \gamma_2\, e_2 \left\{ [\![\rho]\!]_{\eta\,\gamma_1\,\gamma_2} \,\middle|\, [\![\tau]\!]_{\eta\,\gamma_1\,\gamma_2} \right\} \\
\tau <: \tau' &\triangleq \forall \eta, \gamma_1, \gamma_2, v_1, v_2.\ [\![\tau]\!]_{\eta\,\gamma_1\,\gamma_2}\, v_1\, v_2 \twoheadrightarrow [\![\tau']\!]_{\eta\,\gamma_1\,\gamma_2}\, v_1\, v_2 \\
\rho <: \rho' &\triangleq \forall \eta, \gamma_1, \gamma_2.\ (\!|\rho|\!)_{\eta\,\gamma_1\,\gamma_2} \subseteq (\!|\rho'|\!)_{\eta\,\gamma_1\,\gamma_2}
\end{aligned}
$$

(a) Interpretation of typing and subtyping judgments.

$$
[\![\Gamma]\!]_{\eta\,\gamma_1\,\gamma_2} \triangleq \left(\mathop{\ast}_{(x:\tau)\,\in\,\Gamma} .\ [\![\tau]\!]_{\eta\,\gamma_1\,\gamma_2}\, (\gamma_1\, x)\, (\gamma_2\, x)\right) \ast \left(\mathop{\ast}_{(E:\sigma)\,\in\,\Gamma} .\ (\gamma_1\, E, \gamma_2\, E) \ltimes [\![\sigma]\!]_{\eta\,\gamma_1\,\gamma_2}\right)
$$

(b) Interpretation of typing contexts.

$$
[\![\rho]\!]_{\eta\,\gamma_1\,\gamma_2}\, l_1\, l_2\, v_1\, v_2 \triangleq (l_1, l_2) \in (\!|\rho|\!)_{\eta\,\gamma_1\,\gamma_2} \ast \exists \Phi.\ (l_1, l_2) \ltimes \Phi \ast \Box\, \Phi\, v_1\, v_2
$$

$$
\begin{aligned}
(\!|\theta|\!)_{\eta\,\gamma_1\,\gamma_2} &\triangleq \eta\, \theta & (\!|\langle\rangle|\!)_{\eta\,\gamma_1\,\gamma_2} &\triangleq \emptyset & (\!|\rho_1 \cdot \rho_2|\!)_{\eta\,\gamma_1\,\gamma_2} &\triangleq (\!|\rho_1|\!)_{\eta\,\gamma_1\,\gamma_2} \cup (\!|\rho_2|\!)_{\eta\,\gamma_1\,\gamma_2} \\
(\!|E|\!)_{\eta\,\gamma_1\,\gamma_2} &\triangleq \{(\gamma_1\, E, \gamma_2\, E)\} & (\!|\top|\!)_{\eta\,\gamma_1\,\gamma_2} &\triangleq \top
\end{aligned}
$$

(c) Interpretation of rows.

$$
\begin{aligned}
[\![\mathbf{unit}]\!]_{\eta\,\gamma_1\,\gamma_2}\, v_1\, v_2 &\triangleq v_1 = v_2 = () \\
[\![\overline{C \text{ of } \tau}]\!]_{\eta\,\gamma_1\,\gamma_2}\, v_1\, v_2 &\triangleq \exists (C \text{ of } \sigma) \in \overline{C \text{ of } \tau}, w_1, w_2.\ v_1 = C\, w_1 \ast v_2 = C\, w_2 \ast [\![\sigma]\!]_{\eta\,\gamma_1\,\gamma_2}\, w_1\, w_2 \\
[\![\tau \to_\rho \sigma]\!]_{\eta\,\gamma_1\,\gamma_2}\, v_1\, v_2 &\triangleq \Box\, \forall w_1, w_2.\ [\![\tau]\!]_{\eta\,\gamma_1\,\gamma_2}\, w_1\, w_2 \twoheadrightarrow v_1\, w_1 \precsim v_2\, w_2 \left\{ [\![\rho]\!]_{\eta\,\gamma_1\,\gamma_2} \,\middle|\, [\![\sigma]\!]_{\eta\,\gamma_1\,\gamma_2} \right\} \\
[\![\forall \theta.\ \tau]\!]_{\eta\,\gamma_1\,\gamma_2}\, v_1\, v_2 &\triangleq \forall S.\ [\![\tau]\!]_{([\theta \mapsto S]\eta)\,\gamma_1\,\gamma_2}\, v_1\, v_2 \\
[\![\mathbf{exn}_\rho]\!]_{\eta\,\gamma_1\,\gamma_2}\, v_1\, v_2 &\triangleq \exists l_1, l_2, w_1, w_2.\ v_1 = l_1\, w_1 \ast v_2 = l_2\, w_2 \ast [\![\rho]\!]_{\eta\,\gamma_1\,\gamma_2}\, l_1\, l_2\, w_1\, w_2
\end{aligned}
$$

(d) Interpretation of types.

Fig. 6. Semantic interpretation of the ExceptionLang type system.

is defined as a relational specification stating that, for all two substitutions γ_1 and γ_2 of values for variables and of exception labels for exception names, the expression $\gamma_1\, e_1$, obtained by applying the substitution γ_1 to e_1, refines the expression $\gamma_2\, e_2$. For the interpretation of the type system, the expressions e_1 and e_2 in a logical refinement are always the same. However, in §5, we use logical refinement as a way to establish contextual refinement (via Theorem 4.1, to be explained in §4.2). To this end, it is important for logical refinement to be able to relate two different expressions. The substitutions γ_1 and γ_2, as well as the map η of sets of label pairs for row variables, are parameters in the interpretation of types and rows. The substitutions γ_1 and γ_2 are needed to interpret exception names and η is needed to interpret row variables. We omit them in the text for brevity.

The postcondition of $\Gamma \vDash e_1 \leq_{\log} e_2 : \rho : \tau$ concludes that $\gamma_1\, e_1$ and $\gamma_2\, e_2$ either raise $[\![\rho]\!]$-related exceptions or return $[\![\tau]\!]$-related values. The precondition of $\Gamma \vDash e_1 \leq_{\log} e_2 : \rho : \tau$ requires γ_1 and γ_2 to be related, as defined in Figure 6b: for each variable x bound to τ in Γ, the values $\gamma_1\, x$ and $\gamma_2\, x$ are related by $[\![\tau]\!]$ and, for each exception name E bound to σ in Γ, the labels $\gamma_1\, E$ and $\gamma_2\, E$ and the interpretation $[\![\sigma]\!]$ are related by the connective $\ltimes$. The *exception-signature* assertion $(l_1, l_2) \ltimes \Phi$ means l_1 and l_2 are valid labels uniquely associated with Φ. The interpretation of rows relies on this unique association to assert that values raised with these labels are related by Φ.

Figure 6c shows the interpretation of rows. First, the interpretation of a row ρ asserts (l_1, l_2) belongs to $(\!|\rho|\!)$, which corresponds to the set of label pairs denoted by ρ. Its recursive definition is straightforward. Second, the interpretation of ρ asserts v_1 and v_2 are allowed payloads of the exceptions l_1 and l_2. This is done by asserting the existence of a predicate Φ for which the exception-signature assertion holds and by which v_1 and v_2 are related.

The semantic subtyping judgment $\rho <: \rho'$ means exceptions raised in the context of row ρ can be safely raised in a context of row ρ'. Such a property holds whenever $[\![\rho]\!]$-related exceptions are also related by $[\![\rho']\!]$. It suffices to assert $(\!|\rho|\!)$ is included in $(\!|\rho'|\!)$, as the rest of the interpretation does not depend on rows.

Figure 6d shows the interpretation of types. The interpretation of **unit** relates values equal to `()`. The interpretation of a variant type $\overline{C \texttt{ of } \tau}$ relates values built with one of the constructors $C \texttt{ of } \sigma$ of the type using $[\![\sigma]\!]$-related arguments. The interpretation of a function type $\tau \to_\rho \sigma$ relates functions v_1 and v_2 that, when applied to $[\![\tau]\!]$-related values, either return $[\![\sigma]\!]$-related values or raise $[\![\rho]\!]$-related exceptions. The modality $\square$ ensures the interpretation is persistent. The interpretation of a row polymorphic type $\forall\theta.\ \tau$ binds θ to a universally quantified set of label pairs S, thereby translating a *syntactic* $\forall$ into a *semantic* $\forall$. The interpretation of $\textbf{exn}_\rho$ relates exception values $l_1\ w_1$ and $l_2\ w_2$ such that l_1, l_2, w_1, and w_2 are related by $[\![\rho]\!]$, which, to recall, means (1) (l_1, l_2) belongs to the set of label pairs denoted by ρ and (2) w_1 and w_2 are values that can respectively be raised with l_1 and l_2. The interpretation of the omitted types (such as, integers, references, and value-polymorphic types) follows the usual definition of logical relations in Iris [56].

The semantic subtyping judgment $\tau <: \tau'$ means values returned in a context of type τ can also be returned in a context of type τ'. Such a property holds whenever $[\![\tau]\!]$-related values are also related by $[\![\tau']\!]$. This is exactly the definition of $\tau <: \tau'$.

4.2 Soundness and Fundamental Theorem

The following theorem shows logical refinement can be used to derive contextual refinement:

Theorem 4.1 (Sound Logical Relations). *If* $\Gamma \vDash e_1 \leq_{\text{log}} e_2 : \rho : \tau$, *then* $\Gamma \vDash e_1 \leq_{\text{ctx}} e_2 : \rho : \tau$.

Its proof relies on adequacy of our relational logic (Theorem 3.1) among other theorems. It is used extensively in the next section (§5) to show contextual refinements.

The *Fundamental Theorem* shows that a well-typed program indeed meets the relational specification given by the semantic interpretation of its typing judgment:

Theorem 4.2 (Fundamental Theorem). *If* $\Gamma \vdash e : \rho : \tau$, *then* $\Gamma \vDash e : \rho : \tau$.

This theorem creates the link between type system and logic and allows us to exploit soundness of the logic (Theorem 3.1) to show soundness of the type system (Theorem 2.1). Indeed, if a program e is well-typed at an empty typing context (as assumed by Theorem 2.1), that is, if $\vdash e : \rho : \tau$, then it follows by Theorem 4.2 that the specification $\vDash e : \rho : \tau$ is derivable. Then, by Theorem 3.1, it follows, in particular, that e is safe, and that, if $\rho = \langle\rangle$, then e is exception safe.

The proof of Theorem 4.2 follows, as standard in the logical approach, by induction over the derivation of the (syntactic) typing judgment $\Gamma \vdash e : \rho : \tau$. This amounts to proving that, for each typing rule in Figure 2, the *semantic version* of the rule obtained by replacing syntactic judgments with semantic judgments should hold in the logic. Most cases are straightforward, following directly from the reasoning rules of the logic (Figure 3). The key cases are those related to exceptions. They depend on the properties of the exception-signature assertion $\ltimes$ described next.

4.3 A Theory of Exception-Signature Assertions

What properties should exception-signature assertions enjoy so that the semantic versions of the typing rules in Figure 2 hold? The answer is in Figure 7.

Rule Exn-Map-Alloc is used in the proof that the semantic version of Let-Exn-Typed holds to discard the *exnLbl* assertions introduced through Rules Let-Exn-L and Let-Exn-R of the logic

$$\textsc{Exn-Map-Alloc}\quad \frac{\mathit{exnLbl}_{\mathsf{i}}\ l_1 \qquad \mathit{exnLbl}_{\mathsf{s}}\ l_2}{\Rrightarrow (l_1, l_2) \ltimes \Phi}*$$

$$\textsc{Exn-Map-Agree}\quad \frac{(l_1, l_2) \ltimes \Phi \qquad (l_1, l'_2) \ltimes \Psi}{l_2 = l'_2 \ * \ \triangleright \forall v_1, v_2.\ \Phi\, v_1\, v_2 \mathrel{*\!\!*} \Psi\, v_1\, v_2}*$$

$$\textsc{Exn-Map-Disagree}\quad \frac{(l_1, l_2) \ltimes \Phi \qquad (l'_1, l'_2) \ltimes \Psi}{l_1 \neq l'_1 \Rightarrow l_2 \neq l'_2}*$$

$$\textsc{Exn-Map-Proj}\quad (l_1, l_2) \ltimes \Phi \vdash \mathit{exnLbl}^{\square}_{\mathsf{i}}\ l_1 \wedge \mathit{exnLbl}^{\square}_{\mathsf{s}}\ l_2$$

Fig. 7. Properties of exception-signature assertions.

in exchange for an exception-signature assertion with $[\![\sigma]\!]$ as the value relation (where σ is the exception type of the declared exception E). This exception-signature assertion, on its turn, is used to justify the introduction of E at type σ in the typing context.

Rules Exn-Map-Agree and Exn-Map-Disagree are used in the proof that the semantic version of Match-Exn-Typed holds. During this proof, exception-signature assertions come from two places: from the interpretation of the typing context, which must contain the exception name E with type σ, and from the interpretation of **exn**, the type of the scrutinee. In short, Rules Exn-Map-Agree and Exn-Map-Disagree are used to justify that the exception matching in both sides of the refinement in a semantic judgment either both succeed or both fail.

Rule Exn-Map-Proj is used to justify the freshness of newly allocated exception labels during proofs of contextual refinement (§ 5). Via this rule, because one can deduce persistent $\mathit{exnLbl}^{\square}$ assertions hold for l_1 and l_2, it is possible to conclude l_1 and l_2 are distinct from newly allocated ones for which *exnLbl* holds with full ownership.

The exception-signature assertion is defined using *monotone partial bijections* [57], *authoritative ghost state* [26, §6.3.3], and *saved predicates* [14, §5.1]. We omit the details, but emphasize that these Iris constructions act as building blocks for its model and to prove its properties.

5 Examples of Representation Independence

Abstract exceptions. The following two equations characterize the property that, if a program uses an exception abstractly, then locally declared exceptions cannot be used to catch this exception:

$$(h : \tau) \vDash \lambda f.\ \mathbf{let\ exn}\ E\ \mathbf{in\ try}\ f\ ()\ \mathbf{with}\ E\ \_ \Rightarrow h =_{\mathrm{ctx}} \lambda f.\ f\ () : \langle\rangle : \forall \theta.\ (\mathbf{unit} \to_{\theta} \tau) \to_{\theta} \tau$$

$$(h_1 : \tau), (h_2 : \tau) \vDash \lambda x.\ \mathbf{let\ exn}\ E\ \mathbf{in\ match}\ x\ \mathbf{with}\ E\ \_ \Rightarrow h_1 \mid \_ \Rightarrow h_2 =_{\mathrm{ctx}} \lambda\_.\ h_2 : \langle\rangle : \forall \theta.\ \mathbf{exn}_{\theta} \to_{\langle\rangle} \tau$$

Intuitively, because, in both statements, the row θ is polymorphic, both programs should treat the set of exceptions denoted by θ abstractly. In particular, the locally declared exception E cannot accidentally reveal the exceptions denoted by θ. The proof of both cases starts with Theorem 4.1, allowing us to switch to a proof of refinement in the logic. Then, using the exclusive ownership of the label l to which E is dynamically bound and using Exn-Map-Proj, we conclude that any exception raised by $f\ ()$ differs from l, so h and h_1 are never called.

***Equivalent implementations of* mapM.** We show $\mathbf{mapM}_{\mathsf{exn}}$ is contextually equivalent to $\mathbf{mapM}_{\mathsf{opt}}$:

$$\vDash \mathbf{mapM}_{\mathsf{exn}} =_{\mathrm{ctx}} \mathbf{mapM}_{\mathsf{opt}} : \langle\rangle : \forall \theta, \alpha, \beta.\ (\alpha \to_{\theta} \mathbf{option}\ \beta) \to_{\langle\rangle} (\mathbf{list}\ \alpha \to_{\theta} \mathbf{option}\ (\mathbf{list}\ \beta))$$

By definition of $=_{\mathrm{ctx}}$, it suffices to prove contextual refinements on both directions. They are analogous, so let us consider the left-to-right direction. By Theorem 4.1, we exchange $\leq_{\mathrm{ctx}}$ for $\leq_{\mathrm{log}}$. The crux of the proof is then an application of With-Return-L, which gives $\mathbf{mapM}_{\mathsf{exn}}$ exclusive ownership of the exception label l used by **with_return**. By Exn-Map-Proj, any exception raised by the iterated function must differ from l. We modularize the proof by stating a generic refinement between **List.map** and $\mathbf{mapM}_{\mathsf{opt}}$. The statement of this refinement relies on the ability to relate code that raises an exception to code that terminates normally. We therefore use the $e_1 \precsim e_2 \{r_1, r_2. R\}$ form instead of the $e_1 \precsim e_2 \left\{\Phi_E \,\middle|\, \Phi_V\right\}$ form of our refinement judgment.

`Lazy` *and* `Domain`. We prove that `Lazy.from_fun` contextually refines `Domain.spawn`:

$$\vDash \texttt{Lazy.from\_fun} \leq_{\text{ctx}} \texttt{Domain.spawn} : \langle\rangle : \forall\theta.\, \forall\alpha.\, (\texttt{unit} \to_\theta \alpha) \to_{\langle\rangle} (\texttt{unit} \to_\theta \alpha)$$

By Theorem 4.1, we exchange $\leq_{\text{ctx}}$ for $\leq_{\text{log}}$. The proof then follows directly from the refinement in §3.3 by (roughly) instantiating Φ_E with $[\![\theta]\!]$ and Φ_V with $[\![\alpha]\!]$.

***Equivalent implementations of* `new_dyn`.** We show that the three implementations of dynamically extensible variant types (§2.5) are contextually equivalent, by proving a number of logical refinements between them. The proofs rely on picking the right Iris-style invariant and the right semantic interpretation for the existentially quantified type α. We delegate the interested reader to the Rocq mechanization for details and only highlight the key aspects here. First, we generalize the theory of exception-signatures assertions (§4.3). Instead of considering x_1 and x_2 in $(x_1, x_2) \ltimes \Phi$ to be exception labels, we allow these to be of arbitrary types A_1 and A_2. To prove the desired refinements, we let A_1 and A_2 be combinations of exception labels (for $\texttt{new\_dyn}_{\texttt{exn}}$), memory locations (for $\texttt{new\_dyn}_{\texttt{ref}}$), and integers (for $\texttt{new\_dyn}_{\texttt{cntr}}$). Second, to prove that one implementation returns `Some` if and only if the other does so too, we generalize the rules Exn-Map-Agree and Exn-Map-Disagree. Finally, we emphasize that, while the implementation of $\texttt{new\_dyn}_{\texttt{cntr}}$ is untyped, nothing prevents us from proving an equivalence involving an untyped implementation. The definition of the logical relation is stated purely in terms of the language semantics (encapsulated by our separation logic, which operates on untyped programs), not the type system.

6 Related Work

Type systems. The type systems of Standard ML and OCaml do not track exceptions; they guarantee a relatively weak notion of safety, whereby raising an uncaught exception is considered safe. A number of papers describe analyses or type systems to prevent uncaught exceptions [10, 39, 64]. A number of papers also describe type systems to prevent unhandled effects addressing similar concerns of backwards-compatibility [47] or addressing a similar language [12].

Leroy and Pessaux [39] present a program analysis that can estimate uncaught exceptions in ML programs. They combine a type system with exception row polymorphism with control-flow analysis. They use control-flow analysis since in languages like OCaml, a handler may choose to handle an exception based on the value of its arguments. The type system thus needs to keep track of which values can possibly be thrown. They prove type and exception safety of a core version of their system using progress and preservation, but do not study representation independence.

New et al. [47] apply *gradual typing* [54] to address the related challenge of designing a system where *checked* and *unchecked* effects can co-exist and where the migration to *checked* effects is, in our terms, graceful. Their approach differs in scope, in flexibility, and in guarantees. Our scope is a statically typed language where exceptions are locally declared with a fixed signature. New et al. [47] do not consider locally declared effects but design their system to support even dynamically typed languages where effects may abide by zero, one, or more type signatures. Their approach is also more flexible than ours in its support for migrating unchecked to checked effects: modules can be migrated at any order and with support for both up and down casts (via dynamic checks). In our system, our subtyping rules allow only up casts: information can only be lost. Moreover, the order in which function signatures are generalized must respect the order in which these functions are defined. Finally, they prove interesting *static* and *dynamic gradual guarantees* telling that the addition of annotations can only make programs harder to type check or to fail because of dynamic checks. Because our system does not rely on dynamic checks, such a guarantee does not reveal interesting properties of our system, so we focus on representation independence.

de Vilhena and Pottier [12] develop a type system for an ML-style language with dynamically allocated labels for local effects similar to OCaml's **`let exception`** construct we consider. Like our work, they use row-annotated types to ensure (effect) safety. Unlike our work, (1) they do not consider extensible variant types, (2) they do not consider backwards-compatibility issues, and (3) their interpretation of rows requires the sets of labels denoted by two rows ρ_1 and ρ_2 in a row union $\rho_1 \cdot \rho_2$ to be disjoint. This disjointness requirement is necessary in their system for soundness, because their system allows the effect signature of an effect name to be changed whenever a handler is installed. In contrast, following OCaml's approach to fix the signature of an exception when it is declared, we can forgo this requirement. They prove type and effect safety of their system using a unary logical relations model defined in Iris, but do not study representation independence.

Program logics and logical relations. There has been much work on program logics and logical relations for language with non-local control flow, most of which in the unary case [1, 6, 11, 31, 40]. A notable exception is Maillard et al. [41], who develop the first relational program logic for exceptions as part of a generic framework for relational program logics involving arbitrary monadic effects. Similar to our logic, they support postconditions on results that are either values or exceptions (but not expressions in general). However, they do not consider local exceptions, the extensible variant type **`exn`**, concurrency, nor separation-logic assertions and reasoning principles.

The literature also counts with various binary logical relation models for languages with (delimited) continuations (and, in particular, effect handlers) [8, 9, 16, 42, 55, 63]. Since continuations are more expressive than exceptions, they all take a different approach than us. They use *biorthogonality* [49] to define the logical relation in two steps: first relating evaluation contexts, and then relating expressions based on their behavior in related contexts. This approach works well in the context of continuations, but our work shows that the easier approach of generalizing the postcondition is sufficient for exceptions. Up to our knowledge, none of these works consider extensible variant types, nor factor the definition of their logical relation through a relational program logic.

Recent work by de Vilhena et al. [13] develops an Iris-based relational logic for effect handlers. They support local effects in the style of de Vilhena and Pottier [12], but do not consider OCaml's extensible variant type **`Effect.t`**. More broadly, they provide no connection to a type system and do not establish contextual refinement. Their logic enables compositional verification of programs that use different effects via *relational theories*, which is similar to relating code that raises exceptions to code that terminates normally. Despite the similar aim, their approach is different from ours. Their model is a complex recursive definition based on biorthogonality, whereas we use a simpler model inspired by ReLoC [18, 19], which is in turn inspired by Turon et al. [58].

Abstraction guarantees for exceptions. Zhang and Myers [63] prove the *absence of accidental handling* in a type system with *tunneled effects*, which in turn are inspired by *tunneled exceptions* [64]. Tunneled exceptions can be seen as the combination of **`let exn`** with an exception handler. The absence of a *catch-all* construct prevents them from supporting functions like **`Domain.spawn`**, **`Lazy.from_fun`**, and **`Fun.protect`**, but allows them to prove exceptions cannot accidentally be caught by a row-polymorphic function, strengthening their abstraction guarantees. Their language and type system are also different from ours: the core language is sequential and pure (that is, no mutable references) and the type system has no support for extensible variant types.

Comparison to ReLoC and Simuliris. Our work is inspired by ReLoC [18, 19], an Iris-based relational separation logic for contextual refinements, but without support for exceptions. Aside from exceptions, we perform two improvements that are of independent interest: (1) the refinement judgment in ReLoC limits postconditions to persistent predicates on values, whereas our refinement judgment allows arbitrary postconditions on expressions, and (2) our logic enjoys a novel general invariant-opening rule (Inv-Open), whereas ReLoC includes one such rule per atomic instruction.

Simuliris [20] proves termination-preserving refinements in a step-indexing-free setting that uses parameterized coinduction [22] to reason about loops. Their approach requires the generalization to postcondition on expressions, an idea we adopt. They use monotone partial bijections [57] to reason about freshness of memory locations. Allain et al. [3] extend Simuliris to reason handle different calling conventions by generalizing the protocols of de Vilhena and Pottier [11].

7 Future Work

While we focused specifically on exceptions in the style of ML, it would be interesting to investigate whether our techniques can be applied to other languages, such as Java. Exceptions in Java are classes, and therefore have a hierarchical structure, whereas exceptions in ML are flat. It would be interesting to extend our type system with support for orthogonal features for exceptions such as value tracking [39] or down casts [47]. Finally, also in the light of Leroy and Pessaux [39], it would be interesting to investigate a type inference algorithm for our system.

It would be interesting to scale our work to a larger fragment of OCaml, for example, by using the recently developed semantics for OCaml by Seassau et al. [53] or Allain and Scherer [4]. We expect that extending our separation logic and logical relation requires one to investigate several orthogonal features, such as supporting OCaml's non-deterministic expression evaluation. Like most developments based on Iris, we consider sequentially consistent concurrency, while it would be interesting to provide support for OCaml's weak memory model [15, 43].

Acknowledgments

We would like to thank François Pottier for suggesting many of the examples in this paper.

References

[1] Danel Ahman, Catalin Hritcu, Kenji Maillard, Guido Martínez, Gordon D. Plotkin, Jonathan Protzenko, Aseem Rastogi, and Nikhil Swamy. 2017. Dijkstra monads for free. In *POPL*. 515–529. https://doi.org/10.1145/3009837.3009878

[2] Amal Ahmed, Derek Dreyer, and Andreas Rossberg. 2009. State-dependent representation independence. In *POPL*. 340–353. https://doi.org/10.1145/1480881.1480925

[3] Clément Allain, Frédéric Bour, Basile Clément, François Pottier, and Gabriel Scherer. 2025. Tail Modulo Cons, OCaml, and Relational Separation Logic. *PACMPL* 9, POPL (2025), 2337–2363. https://doi.org/10.1145/3704915

[4] Clément Allain and Gabriel Scherer. 2026. Zoo: A framework for the verification of concurrent OCaml 5 programs using separation logic. *PACMPL* (2026). https://clef-men.github.io/publications/allain-scherer-26.pdf

[5] Anonymous Authors. 2025. Rocq mechanization of "Backwards-Compatible Row-Based Exceptions in ML". Submitted as anonymous supplementary material in HotCRP.

[6] Andrew W. Appel and Sandrine Blazy. 2007. Separation Logic for Small-Step Cminor. In *TPHOLs (LNCS, Vol. 4732)*. 5–21. https://doi.org/10.1007/978-3-540-74591-4_3

[7] Andrew W. Appel, Paul-André Melliès, Christopher D. Richards, and Jérôme Vouillon. 2007. A very modal model of a modern, major, general type system. In *POPL*. 109–122. https://doi.org/10.1145/1190216.1190235

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

[9] Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. 2020. Binders by day, labels by night: Effect instances via lexically scoped handlers. *PACMPL* 4, POPL (2020), 48:1–48:29. https://doi.org/10.1145/3371116

[10] Matthias Blume, Umut A. Acar, and Wonseok Chae. 2008. Exception Handlers as Extensible Cases. In *APLAS (LNCS, Vol. 5356)*. 273–289. https://doi.org/10.1007/978-3-540-89330-1_20

[11] Paulo Emílio de Vilhena and François Pottier. 2021. A separation logic for effect handlers. *PACMPL* 5, POPL (2021), 1–28. https://doi.org/10.1145/3434314

[12] Paulo Emílio de Vilhena and François Pottier. 2023. A Type System for Effect Handlers and Dynamic Labels. In *ESOP (LNCS, Vol. 13990)*. 225–252. https://doi.org/10.1007/978-3-031-30044-8_9

[13] Paulo Emílio de Vilhena, Simcha van Collem, Ines Wright, and Robbert Krebbers. 2026. A Relational Separation Logic for Effect Handlers. *PACMPL* POPL (2026). https://devilhena-paulo.github.io/files/blaze.pdf

[14] Mike Dodds, Suresh Jagannathan, Matthew J. Parkinson, Kasper Svendsen, and Lars Birkedal. 2016. Verifying Custom Synchronization Constructs Using Higher-Order Separation Logic. *TOPLAS* 38, 2 (2016), 4:1–4:72. https://doi.org/10.1145/2818638

[15] Stephen Dolan, K. C. Sivaramakrishnan, and Anil Madhavapeddy. 2018. Bounding Data Races in Space and Time. In *PLDI*. 242–255. https://doi.org/10.1145/3192366.3192421

[16] Derek Dreyer, Amal Ahmed, and Lars Birkedal. 2009. Logical Step-Indexed Logical Relations. In *LICS*. 71–80. https://doi.org/10.1109/LICS.2009.34

[17] Andrzej Filinski. 1996. *Controlling Effects*. Ph. D. Dissertation. Carnegie Mellon University.

[18] Dan Frumin, Robbert Krebbers, and Lars Birkedal. 2018. ReLoC: A Mechanised Relational Logic for Fine-Grained Concurrency. In *LICS*. 442–451. https://doi.org/10.1145/3209108.3209174

[19] Dan Frumin, Robbert Krebbers, and Lars Birkedal. 2021. ReLoC Reloaded: A Mechanized Relational Logic for Fine-Grained Concurrency and Logical Atomicity. *LMCS* 17, 3 (2021). https://doi.org/10.46298/LMCS-17(3:9)2021

[20] Lennard Gäher, Michael Sammler, Simon Spies, Ralf Jung, Hoang-Hai Dang, Robbert Krebbers, Jeehoon Kang, and Derek Dreyer. 2022. Simuliris: A separation logic framework for verifying concurrent program optimizations. *PACMPL* 6, POPL (2022), 1–31. https://doi.org/10.1145/3498689

[21] Daniel Hillerström and Sam Lindley. 2016. Liberating effects with rows and handlers. In *TyDe@ICFP*. 15–27. https://doi.org/10.1145/2976022.2976033

[22] Chung-Kil Hur, Georg Neis, Derek Dreyer, and Viktor Vafeiadis. 2013. The power of parameterization in coinductive proof. In *POPL*. 193–206. https://doi.org/10.1145/2429069.2429093

[23] Jane Street. 2025. Module Base.With_return. https://ocaml.janestreet.com/ocaml-core/v0.14/doc/base/Base/With_return/index.html

[24] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2018. RustBelt: securing the foundations of the rust programming language. *PACMPL* 2, POPL (2018), 66:1–66:34. https://doi.org/10.1145/3158154

[25] Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. 2016. Higher-order ghost state. In *ICFP*. 256–269. https://doi.org/10.1145/2951913.2951943

[26] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *JFP* 28 (2018), e20. https://doi.org/10.1017/S0956796818000151

[27] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In *POPL*. 637–650. https://doi.org/10.1145/2676726.2676980

[28] Robbert Krebbers, Jacques-Henri Jourdan, Ralf Jung, Joseph Tassarotti, Jan-Oliver Kaiser, Amin Timany, Arthur Charguéraud, and Derek Dreyer. 2018. MoSeL: A general, extensible modal framework for interactive proofs in separation logic. *PACMPL* 2, ICFP (2018), 77:1–77:30. https://doi.org/10.1145/3236772

[29] Robbert Krebbers, Ralf Jung, Ales Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. 2017. The Essence of Higher-Order Concurrent Separation Logic. In *ESOP (LNCS, Vol. 10201)*. 696–723. https://doi.org/10.1007/978-3-662-54434-1_26

[30] Robbert Krebbers, Amin Timany, and Lars Birkedal. 2017. Interactive proofs in higher-order concurrent separation logic. In *POPL*. 205–217. https://doi.org/10.1145/3009837.3009855

[31] Robbert Krebbers and Freek Wiedijk. 2013. Separation Logic for Non-local Control Flow and Block Scope Variables. In *FoSSaCS (LNCS, Vol. 7794)*. 257–272. https://doi.org/10.1007/978-3-642-37075-5_17

[32] Daan Leijen. 2014. Koka: Programming with Row Polymorphic Effect Types. In *MSFP@ETAPS (EPTCS, Vol. 153)*. 100–126. https://doi.org/10.4204/EPTCS.153.8

[33] Daan Leijen. 2017. Type directed compilation of row-typed algebraic effects. In *POPL*. 486–499. https://doi.org/10.1145/3009837.3009872

[34] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. 2025. OCaml manual (Version 5.4): Extensible variant types. https://ocaml.org/manual/5.4/extensiblevariants.html

[35] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. 2025. OCaml manual (Version 5.4): Module Domain. https://ocaml.org/manual/5.4/api/Domain.html

[36] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. 2025. OCaml manual (Version 5.4): Module Fun. https://ocaml.org/manual/5.4/api/Fun.html

[37] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. 2025. OCaml manual (Version 5.4): Module Lazy. https://ocaml.org/manual/5.4/api/Lazy.html

[38] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. 2025. OCaml manual (Version 5.4): Module Obj. https://ocaml.org/manual/5.4/api/Obj.html

[39] Xavier Leroy and François Pessaux. 2000. Type-based analysis of uncaught exceptions. *TOPLAS* 22, 2 (2000), 340–377. https://doi.org/10.1145/349214.349230

[40] Kenji Maillard, Danel Ahman, Robert Atkey, Guido Martínez, Catalin Hritcu, Exequiel Rivas, and Éric Tanter. 2019. Dijkstra monads for all. *PACMPL* 3, ICFP (2019), 104:1–104:29. https://doi.org/10.1145/3341708

[41] Kenji Maillard, Catalin Hritcu, Exequiel Rivas, and Antoine Van Muylder. 2020. The next 700 relational program logics. *PACMPL* 4, POPL (2020), 4:1–4:33. https://doi.org/10.1145/3371072

[42] Craig McLaughlin. 2020. *Relational reasoning for effects and handlers.* Ph. D. Dissertation. University of Edinburgh, UK. https://doi.org/10.7488/ERA/537

[43] Glen Mével, Jacques-Henri Jourdan, and François Pottier. 2020. Cosmo: A concurrent separation logic for multicore OCaml. *PACMPL* 4, ICFP (2020), 96:1–96:29. https://doi.org/10.1145/3408978

[44] Robin Milner and Mads Tofte. 1991. *Commentary on standard ML.* MIT Press.

[45] John C. Mitchell. 1986. Representation Independence and Data Abstraction. In *POPL*. 263–276. https://doi.org/10.1145/512644.512669

[46] Hiroshi Nakano. 2000. A Modality for Recursion. In *LICS*. 255–266. https://doi.org/10.1109/LICS.2000.855774

[47] Max S. New, Eric Giovannini, and Daniel R. Licata. 2023. Gradual Typing for Effect Handlers. 7, OOPSLA (oct 2023). https://doi.org/10.1145/3622860

[48] Benjamin C. Pierce. 2002. *Types and programming languages.* MIT Press.

[49] Andrew M. Pitts. 2005. Typed Operational Reasoning. In *Advanced Topics in Types and Programming Languages*, Benjamin C. Pierce (Ed.). MIT Press, Chapter 7, 245–289.

[50] Andrew M. Pitts and Ian Stark. 1998. Operational reasoning for functions with local state. In *HOOTS*.

[51] Didier Rémy. 1989. Typechecking Records and Variants in a Natural Extension of ML. In *POPL*. 77–88. https://doi.org/10.1145/75277.75284

[52] John C. Reynolds. 1974. Towards a theory of type structure. In *Programming Symposium, Proceedings Colloque sur la Programmation (LNCS, Vol. 19)*. 408–423. https://doi.org/10.1007/3-540-06859-7_148

[53] Remy Seassau, Irene Yoon, Jean-Marie Madiot, and François Pottier. 2025. Formal semantics and program logics for a fragment of OCaml. To appear at ICFP 2025.

[54] Jeremy Siek and Walid Taha. 2006. Gradual typing for functional languages. *Scheme and Functional Programming*.

[55] Amin Timany and Lars Birkedal. 2019. Mechanized relational verification of concurrent programs with continuations. *PACMPL* 3, ICFP (2019), 105:1–105:28. https://doi.org/10.1145/3341709

[56] Amin Timany, Robbert Krebbers, Derek Dreyer, and Lars Birkedal. 2024. A Logical Approach to Type Soundness. *JACM* 71, 6 (2024), 40:1–40:75. https://doi.org/10.1145/3676954

[57] Amin Timany, Léo Stefanesco, Morten Krogh-Jespersen, and Lars Birkedal. 2017. A logical relation for monadic encapsulation of state: proving contextual equivalences in the presence of runST. *PACMPL* 2, POPL (2017). https://doi.org/10.1145/3158152

[58] Aaron Turon, Derek Dreyer, and Lars Birkedal. 2013. Unifying refinement and Hoare-style reasoning in a logic for higher-order concurrency. In *ICFP*. 377–390. https://doi.org/10.1145/2500365.2500600

[59] Aaron Joseph Turon, Jacob Thamsborg, Amal Ahmed, Lars Birkedal, and Derek Dreyer. 2013. Logical relations for fine-grained concurrency. In *POPL*. 343–356. https://doi.org/10.1145/2429069.2429111

[60] Simon Friis Vindum and Lars Birkedal. 2021. Contextual refinement of the Michael-Scott queue (proof pearl). In *CPP*. 76–90. https://doi.org/10.1145/3437992.3439930

[61] Mitchell Wand. 1991. Type Inference for Record Concatenation and Multiple Inheritance. *I&C* 93, 1 (1991), 1–15. https://doi.org/10.1016/0890-5401(91)90050-C

[62] Andrew K. Wright. 1995. Simple Imperative Polymorphism. *LISP Symb. Comput.* 8, 4 (1995), 343–355.

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

[64] Yizhou Zhang, Guido Salvaneschi, Quinn Beightol, Barbara Liskov, and Andrew C. Myers. 2016. Accepting blame for safe tunneled exceptions. In *PLDI*. 281–295. https://doi.org/10.1145/2908080.2908086