Backwards-Compatible Row-Based Exceptions in ML

Simcha van Collem[a], Paulo Emílio de Vilhena[b], Robbert Krebbers[a]

17 June 2026

Radboud University Nijmegen[a], Imperial College London → University of Surrey[b]

Type Safety and Exceptions

Type safety: typed programs cannot go wrong

Type Safety and Exceptions

Type safety: typed programs ~~cannot go wrong~~ can still raise exceptions

`List.map (fun _ -> raise Err) [1; 2; 3]` has type `int list`

Type Safety and Exceptions

Type safety: typed programs ~~cannot go wrong~~ can still raise exceptions

`List.map (fun _ -> raise Err) [1; 2; 3]` has type `int list`

Exception safety: typed programs cannot go wrong and do not raise exceptions

Type Safety and Exceptions

Type safety: typed programs ~~cannot go wrong~~ can still raise exceptions

`List.map (fun _ -> raise Err) [1; 2; 3]` has type `int list` and row $\langle$`Err`$\rangle$

Exception safety: typed programs with row $\langle\rangle$ cannot go wrong and do not raise exceptions

Type Safety and Exceptions

Type safety: typed programs ~~cannot go wrong~~ can still raise exceptions

`List.map (fun _ -> raise Err) [1; 2; 3]` has type `int list` and row $\langle$`Err`$\rangle$

Exception safety: typed programs with row $\langle\rangle$ cannot go wrong and do not raise exceptions

Goal: design a backwards-compatible type system with exception tracking

Backwards Compatibility

- Programmer should be able to opt out from exception tracking
- Function signatures can be migrated to exception tracking one at a time

Backwards Compatibility

- Programmer should be able to opt out from exception tracking
  - Programs should remain to type check
- Function signatures can be migrated to exception tracking one at a time

Backwards Compatibility

- Programmer should be able to opt out from exception tracking
  - Programs should remain to type check
- Function signatures can be migrated to exception tracking one at a time

```
let prog1 : τ = ... in
let prog2 : σ = ... prog1 ... in
...
```

Backwards Compatibility

- Programmer should be able to opt out from exception tracking
  - Programs should remain to type check
- Function signatures can be migrated to exception tracking one at a time

```
let prog1 : τ raises ρ = ... in
let prog2 : σ = ... prog1 ... in
...
```

Contributions

- ML-style type system with support for:
  - Higher-order references
  - Polymorphism
  - Concurrency
  - Local exceptions
  - Exception tracking
- Backwards compatibility
- Safety and abstraction guarantees
- Binary logical relations model via a relational separation logic
- Mechanized in Rocq using Iris

Exceptions can be declared locally in ML, *e.g.*, in a function scope

Backwards-Compatible Row-Based Exceptions in ML

The type of exceptions in ML, making them first-class data

- Rows track exception names: $\rho ::= \langle\rangle \mid \theta \mid \langle E \rangle \mid \rho \cdot \rho$
- We annotate $\Gamma \vdash e : \rho : \tau$, and types $\tau \to_\rho \sigma$ and $\texttt{exn}_\rho$ with rows
- Row polymorphism gives us strong types:

$$\texttt{List.map}\colon \forall \theta, \alpha, \beta.\, (\alpha \to_\theta \beta) \to_{\langle\rangle} \alpha\, \texttt{list} \to_\theta \beta\, \texttt{list}$$

Backwards-Compatible Row-Based Exceptions in ML

The type of exceptions in ML, making them first-class data

- Rows track exception names: $\rho ::= \langle\rangle \mid \theta \mid \langle E \rangle \mid \rho \cdot \rho \mid \top$
- We annotate $\Gamma \vdash e : \rho : \tau$, and types $\tau \to_\rho \sigma$ and $\texttt{exn}_\rho$ with rows
- Row polymorphism gives us strong types:

$$\texttt{List.map}\colon \forall \theta, \alpha, \beta.\, (\alpha \to_\theta \beta) \to_{\langle\rangle} \alpha\, \texttt{list} \to_\theta \beta\, \texttt{list}$$

- $\top$ indicates that any exception in scope may be raised

Backwards-Compatible Row-Based Exceptions in ML

Recall: programmers should be able to opt out, and migrating should not break code

Backwards-Compatible Row-Based Exceptions in ML

Recall: programmers should be able to opt out, and migrating should not break code

Theorem: Our type system is an extension: $\Gamma \vdash e : \tau$ implies $\Gamma \vdash e : \top : \tau$

Implicit translation: $\tau \to \sigma$ maps to $\tau \to_\top \sigma$, and `exn` maps to $\texttt{exn}_\top$

Backwards-Compatible Row-Based Exceptions in ML

Recall: programmers should be able to opt out, and migrating should not break code

Theorem: Our type system is an extension: $\Gamma \vdash e : \tau$ implies $\Gamma \vdash e : \top : \tau$

Implicit translation: $\tau \to \sigma$ maps to $\tau \to_\top \sigma$, and `exn` maps to $\texttt{exn}_\top$

Graceful generalization: We can subtype interesting examples one at a time

$$\tau_{\texttt{with\_exception\_tracking}} <: \tau_{\texttt{without\_exception\_tracking}}$$

Recall: programmers should be able to opt out, and migrating should not break code

Theorem: Our type system is an extension: $\Gamma \vdash e : \tau$ implies $\Gamma \vdash e : \top : \tau$

Implicit translation: $\tau \to \sigma$ maps to $\tau \to_\top \sigma$, and `exn` maps to $\texttt{exn}_\top$

Graceful generalization: We can subtype interesting examples one at a time

$$\tau_{\texttt{with\_exception\_tracking}} <: \tau_{\texttt{without\_exception\_tracking}}$$

```
let prog1 : τ raises ρ = ... in
let prog2 : σ = ... prog1 ... in
...
```

Backwards-Compatible Row-Based Exceptions in ML

Recall: programmers should be able to opt out, and migrating should not break code

Theorem: Our type system is an extension: $\Gamma \vdash e : \tau$ implies $\Gamma \vdash e : \top : \tau$

Implicit translation: $\tau \to \sigma$ maps to $\tau \to_\top \sigma$, and `exn` maps to $\texttt{exn}_\top$

Graceful generalization: We can subtype interesting examples one at a time

$$\tau_{\texttt{with\_exception\_tracking}} <: \tau_{\texttt{without\_exception\_tracking}}$$

- Not a formal property, but we show this for various example from OCaml's standard libraries
- Precise exception tracking via rank-N row polymorphism

`with_return` (adopted from the Jane Street OCaml standard library)

```
(** with_return : ∀α. ((α → ⊥) → α) → α *)
let with_return (f : (α → ⊥) → α) =
  let exception Return of α in

  let return = fun x -> raise (Return x) in
  try
    f return
  with Return x -> x
```

`with_return` (adopted from the Jane Street OCaml standard library)

```
(** with_return : ∀α. ((α → ⊥) → α) → α *)
let with_return (f : (α → ⊥) → α) =
  let exception Return of α in

  let return = fun x -> raise (Return x) in
  try
    f return
  with Return x -> x
```

`with_return` (adopted from the Jane Street OCaml standard library)

```
(** with_return : ∀α. ((α → ⊥) → α) → α *)
let with_return (f : (α → ⊥) → α) =
  let exception Return of α in

  let return = fun x -> raise (Return x) in
  try
    f return
  with Return x -> x
```

`with_return` (adopted from the Jane Street OCaml standard library)

```
(** with_return : ∀α. ((α → ⊥) → α) → α *)
let with_return (f : (α → ⊥) → α) =
  let exception Return of α in

  let return = fun x -> raise (Return x) in
  try
    f return
  with Return x -> x
```

`with_return` (adopted from the Jane Street OCaml standard library)

```
(** with_return : ∀α. ((α → ⊥) → α) → α *)
let with_return (f : (α → ⊥) → α) =
  let exception Return of α in

  let return = fun x -> raise (Return x) in
  try
    f return
  with Return x -> x
```

`with_return` (adopted from the Jane Street OCaml standard library)

```
(** with_return : ∀α. ((α →⊤ ⊥) →⊤ α) →⊤ α *)
let with_return (f : (α →⊤ ⊥) →⊤ α) =
  let exception Return of α in

  let return = fun x -> raise (Return x) in
  try
    f return
  with Return x -> x
```

`with_return` (adopted from the Jane Street OCaml standard library)

```
(** with_return : ∀θ_user, α. ((α →_⊤ ⊥) →_θ_user α) →_θ_user α *)
let with_return (f : (α →_⊤ ⊥) →_θ_user α) =
  let exception Return of α in

  let return = fun x -> raise (Return x) in
  try
    f return
  with Return x -> x
```

`with_return` (adopted from the Jane Street OCaml standard library)

```
(** with_return : ∀θ_user, α. (∀θ_local. (α →θ_local ⊥) →θ_user·θ_local α) →θ_user α *)
let with_return (f : ∀θ_local. (α →θ_local ⊥) →θ_user·θ_local α) =
  let exception Return of α in

  let return = fun x -> raise (Return x) in
  try
    f return
  with Return x -> x
```

`with_return` (adopted from the Jane Street OCaml standard library)

```
(** with_return : ∀θ_user, α. (∀θ_local. (α →_θ_local ⊥) →_θ_user·θ_local α) →_θ_user α *)
let with_return (f : ∀θ_local. (α →_θ_local ⊥) →_θ_user·θ_local α) =
  let exception Return of α in
  (* Instantiate θ_local with ⟨Return⟩ *)
  let return = fun x -> raise (Return x) in
  try
    f return
  with Return x -> x
```

`with_return` (adopted from the Jane Street OCaml standard library)

```
(** with_return : ∀θ_user, α. (∀θ_local. (α →_θ_local ⊥) →_θ_user·θ_local α) →_θ_user α *)
let with_return (f : ∀θ_local. (α →_θ_local ⊥) →_θ_user·θ_local α) =
  let exception Return of α in
  (* Instantiate θ_local with ⟨Return⟩ *)
  let return = fun x -> raise (Return x) in
  try
    f return
  with Return x -> x
  (* Exceptions from θ_local are handled, those in θ_user are bubbled up *)
```

`with_return` (adopted from the Jane Street OCaml standard library)

```
(** with_return : ∀θ_user, α. (∀θ_local. (α →_θ_local ⊥) →_θ_user·θ_local α) →_θ_user α *)
let with_return (f : ∀θ_local. (α →_θ_local ⊥) →_θ_user·θ_local α) =
  let exception Return of α in
  (* Instantiate θ_local with ⟨Return⟩ *)
  let return = fun x -> raise (Return x) in
  try
    f return
  with Return x -> x
  (* Exceptions from θ_local are handled, those in θ_user are bubbled up *)
```

`with_return`'s original type can be recovered as a supertype

$$\forall\theta_{\text{user}}, \alpha. (\forall\theta_{\text{local}}. (\alpha \to_{\theta_{\text{local}}} \bot) \to_{\theta_{\text{user}}\cdot\theta_{\text{local}}} \alpha) \to_{\theta_{\text{user}}} \alpha \;\; <: \;\; \forall\alpha. ((\alpha \to_{\top} \bot) \to_{\top} \alpha) \to_{\top} \alpha$$

- Introduce $\forall\,\theta$ on the right, and later subtype $\theta <: \top$
- Eliminate $\forall\,\theta$ on the left with $\top$

Representation Independence

- Just like local references, local exceptions should be local

$$\texttt{let exception E in try f () with E x -> h} =_{\mathrm{ctx}} \texttt{f ()}$$

Representation Independence

- Just like local references, local exceptions should be local

$$\texttt{let exception E in try f () with E x -> h} =_{\text{ctx}} \texttt{f ()}$$

$$\texttt{let r = ref 10 in f (); ! r} =_{\text{ctx}} \texttt{f (); 10}$$

Representation Independence

- Just like local references, local exceptions should be local

$$\texttt{let exception E in try f () with E x -> h} =_{\text{ctx}} \texttt{f ()}$$

$$\texttt{let r = ref 10 in f (); ! r} =_{\text{ctx}} \texttt{f (); 10}$$

- We present a methodology to prove these contextual equivalences
- **Novelty:** semantic interpretation of rows
  - $\top$ row is not any exception, but precisely those in scope
- Implementations using `with_return` and option monad are contextually equivalent
  - More examples involving concurrency, storing exceptions, ...
- Backwards compatibility gives us strong results

Conclusions

- Type system for exception tracking in ML-style languages
- Row polymorphism, $\top$ row, and subtyping ensure backwards compatibility
- Safety and abstraction guarantees
- Proven correct in Rocq using Iris