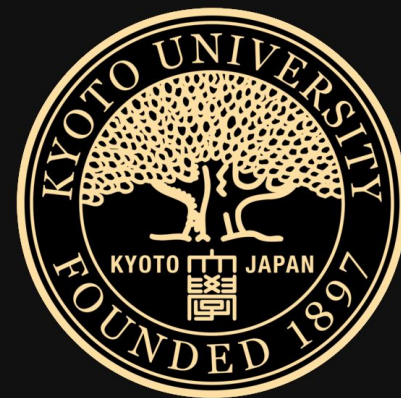


Refined² Environment Classifiers



Yuito Murase and Atsushi Igarashi from Kyoto University

2026-Oct-7 OOPSLA @ Oakland



Static Safety for MetaML-style Staging with Effects

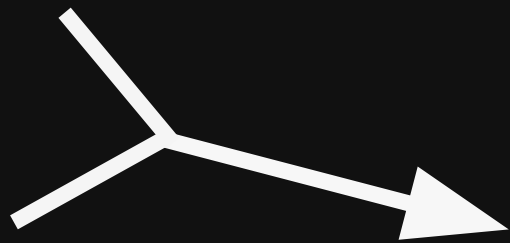
Static Safety

for

MetaML-style Staging

with

Effects



Scope Extrusion

MetaML-style Staging [Taha+ 2000]

```
let x = < 1 + 2 > in  
run < 10 * $x >
```

MetaML-style Staging [Taha+ 2000]

Quote



```
let x = < 1 + 2 > in  
run < 10 * $x >
```

MetaML-style Staging [Taha+ 2000]

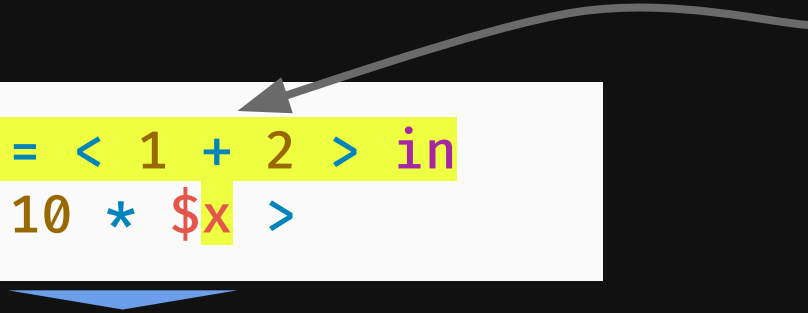
Quote



```
let x = < 1 + 2 > in  
run < 10 * $x >
```

MetaML-style Staging [Taha+ 2000]

Quote



```
let x = < 1 + 2 > in  
run < 10 * $x >
```

```
run < 10 * $< 1 + 2 > >
```

MetaML-style Staging [Taha+ 2000]

```
let x = < 1 + 2 > in  
run < 10 * $x >
```

```
run < 10 * $< 1 + 2 > >
```

Quote

Splice

MetaML-style Staging [Taha+ 2000]

```
let x = < 1 + 2 > in  
run < 10 * $x >
```

Quote

```
run < 10 * $< 1 + 2 > >
```

Splice

```
run < 10 * (1 + 2) >
```

MetaML-style Staging [Taha+ 2000]

```
let x = < 1 + 2 > in  
run < 10 * $x >
```

```
run < 10 * $< 1 + 2 > >
```

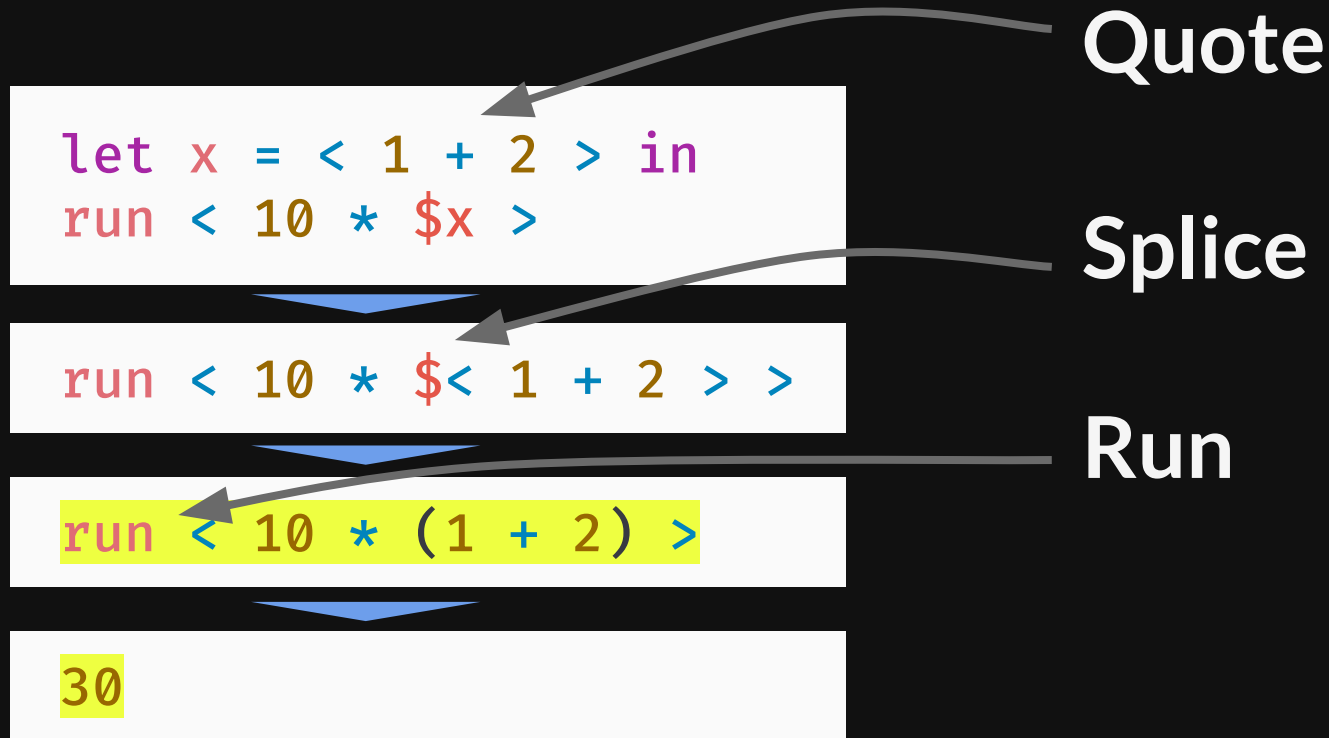
```
run < 10 * (1 + 2) >
```

Quote

Splice

Run

MetaML-style Staging [Taha+ 2000]



Staging + Effects $\Rightarrow$ Scope Extrusion [Taha+ 2000]

env



heap



```
let r = ref < 1 > in
let _ = < fun x  $\rightarrow$ 
    $(r := < x >; < x >)
    > in
deref r
```

Staging + Effects $\Rightarrow$ Scope Extrusion [Taha+ 2000]

env



heap



```
let r = ref < 1 > in
let _ = < fun x →
    $(r := < x >; < x >)
    > in
deref r
```

Staging + Effects $\Rightarrow$ Scope Extrusion [Taha+ 2000]

env

 $r \mapsto l$

heap

 $l \mapsto \langle 1 \rangle$

```
let r = ref < 1 > in
let _ = < fun x  $\rightarrow$ 
    $(r := < x >; < x >)
    > in
deref r
```

Staging + Effects $\Rightarrow$ Scope Extrusion [Taha+ 2000]

env

 $r \mapsto l$

heap

 $l \mapsto \langle 1 \rangle$

```
let r = ref < 1 > in
let _ = < fun x  $\rightarrow$ 
    $(r := < x >; < x >)
    > in
deref r
```

Staging + Effects $\Rightarrow$ Scope Extrusion [Taha+ 2000]

env

 $r \mapsto l$

heap

 $l \mapsto \langle x \rangle$

```
let r = ref < 1 > in
let _ = < fun x  $\rightarrow$ 
    $(r := < x >; < x >)
    > in
deref r
```

Staging + Effects $\Rightarrow$ Scope Extrusion [Taha+ 2000]

env

 $r \mapsto l$

heap

 $l \mapsto \langle x \rangle$

```
let r = ref < 1 > in
```

```
let _ = < fun x  $\rightarrow$ 
```

```
    $(r := < x >; < x >)
```

```
    > in
```

```
deref r
```

Staging + Effects $\Rightarrow$ Scope Extrusion [Taha+ 2000]

env

 $r \mapsto l$

heap

 $l \mapsto \langle x \rangle$

```
let r = ref < 1 > in
let _ = < fun x  $\rightarrow$ 
    $(r := < x >; < x >)
    > in
deref r
```

Staging + Effects $\Rightarrow$ Scope Extrusion [Taha+ 2000]

env

 $r \mapsto l$

heap

 $l \mapsto \langle x \rangle$

```
let r = ref < 1 > in
let _ = < fun x  $\rightarrow$ 
    $(r := < x >; < x >)
    > in
deref r
```

Staging + Effects $\Rightarrow$ Scope Extrusion [Taha+ 2000]

env

 $r \mapsto l$

heap

 $l \mapsto \langle x \rangle$

```

let r = ref < 1 > in
let _ = < fun x  $\rightarrow$ 
            $(r := < x >; < x >)
            > in
< x >

```

Undefined variable

Staging + Effects $\Rightarrow$ Scope Extrusion [Taha+ 2000]

env

 $r \mapsto l$

heap

 $l \mapsto \langle x \rangle$

```
let r = ref < 1 > in
```

```
let _ = < fun x  $\rightarrow$ 
```

scope of x

```
$(r := < x >; < x >)
```

```
> in
```

```
< x >
```

Escape from
the defining scope

Type Systems for MetaML-style staging

MetaML[Taha+ 2000], λ^α [Taha+ 2003]

- 🤪 Cannot detect scope extrusion

Closed Types [Calcagno+ 2000]

- Avoid scope extrusion by *allowing only closed code to be stored*

Refined Environment Classifiers [Kiselyov+ 2014, Lee+ 2026]

- Avoid scope extrusion while *allowing open code to be stored*
- But supports only *limited subset of MetaML-style staging*

Q: Can RECs be generalized to support full MetaML-style staging?

Type Systems for MetaML-style staging

MetaML[Taha+ 2000], λ^α [Taha+ 2003]

- 🤔 Cannot detect scope extrusion

Closed Types [Calcagno+ 2000]

- 🤔 Avoid scope extrusion by *allowing only closed code to be stored*

Refined Environment Classifiers [Kiselyov+ 2014, Lee+ 2026]

- Avoid scope extrusion while *allowing open code to be stored*
- But supports only *limited subset of MetaML-style staging*

**Q: Can RECs be generalized to support
full MetaML-style staging?**

Type Systems for MetaML-style staging

MetaML[Taha+ 2000], λ^α [Taha+ 2003]

- 🤔 Cannot detect scope extrusion

Closed Types [Calcagno+ 2000]

- 🤔 Avoid scope extrusion by *allowing only closed code to be stored*

Refined Environment Classifiers [Kiselyov+ 2014, Lee+ 2026]

- 😊 Avoid scope extrusion while *allowing open code to be stored*
- 😞 But supports only *limited subset of MetaML-style staging*

Q: Can RECs be generalized to support
full MetaML-style staging?

Type Systems for MetaML-style staging

MetaML[Taha+ 2000], λ^α [Taha+ 2003]

- 🤔 Cannot detect scope extrusion

Closed Types [Calcagno+ 2000]

- 🤔 Avoid scope extrusion by *allowing only closed code to be stored*

Refined Environment Classifiers [Kiselyov+ 2014, Lee+ 2026]

- 😊 Avoid scope extrusion while *allowing open code to be stored*
- 😞 But supports only *limited subset of MetaML-style staging*

**Q: Can RECs be generalized to support
full MetaML-style staging?**

Refined Environment Classifiers

[Kiselyov+ 2014, Lee+ 2026]

- Two-level staging
- Does NOT support run with cross-stage persistence
- Code generators are not reusable

Reconstruct



Refined² Environment Classifiers

Our Proposal

- Multi-level staging
- Support run with cross-stage persistence
- Reusable code generators

Refined Environment Classifiers

[Kiselyov+ 2014, Lee+ 2026]

Reconstruct

Refined² Environment Classifiers

Our Proposal

- Two-level staging
- Does NOT support run with cross-stage persistence
- Code generators are not reusable

- Multi-level staging
- Support run with cross-stage persistence
- Reusable code generators

Explicit Scope Dependency via RECs [Kiselyov+ 2014, Lee+ 2026]

Classifiers (e.g. `!`, `γ1`, `γ2`) represent sets of available variables

- A binding form declares a classifier for the new scope
- A quote carries an existing classifier to specify the scope it depends on
- A code type `<T@γ>` carries a classifier to specify the scope it depends on

```

<!let a@γ1 = 2 in
  let b@γ2 = 3 in
    $ { <γ2 a + b > } >
  
```

(`!` is reserved for the global scope)

`!` $\Rightarrow$ (no vars)

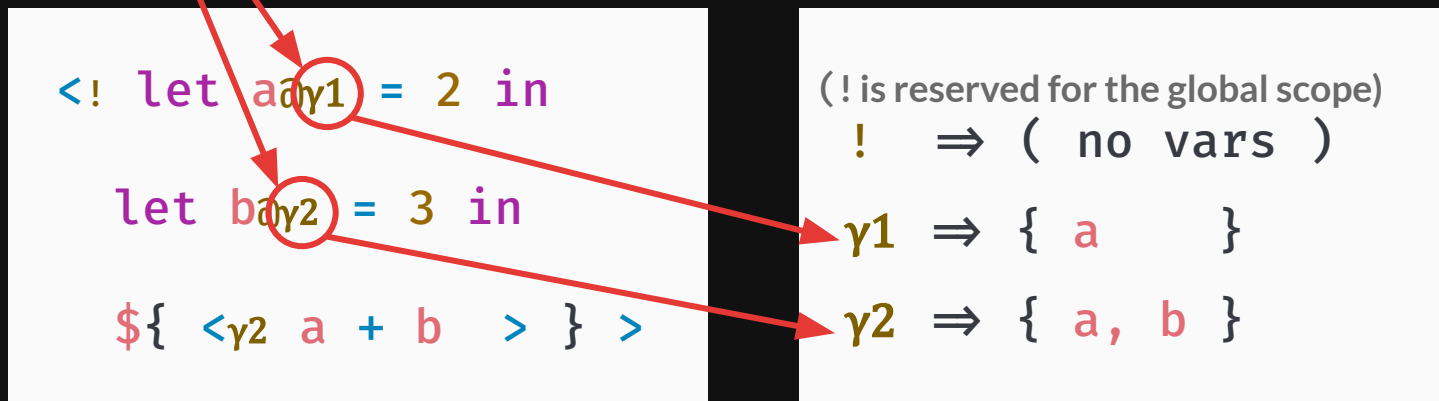
`γ1` $\Rightarrow$ { a }

`γ2` $\Rightarrow$ { a, b }

Explicit Scope Dependency via RECs [Kiselyov+ 2014, Lee+ 2026]

Classifiers (e.g. `!`, γ_1 , γ_2) represent sets of available variables

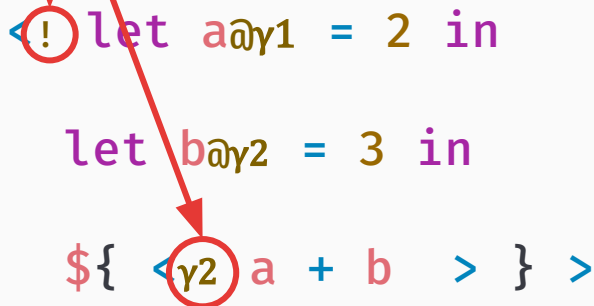
- A binding form declares a classifier for the new scope
- A quote carries an existing classifier to specify the scope it depends on
- A code type `<T@ $\gamma$ >` carries a classifier to specify the scope it depends on



Explicit Scope Dependency via RECs [Kiselyov+ 2014, Lee+ 2026]

Classifiers (e.g. `!`, `γ1`, `γ2`) represent sets of available variables

- A binding form declares a classifier for the new scope
- A quote carries an existing classifier to specify the scope it depends on
- A code type `<T@γ>` carries a classifier to specify the scope it depends on



```
<!  
let a@γ1 = 2 in  
  let b@γ2 = 3 in  
    ${ <γ2 a + b > } >
```

(`!` is reserved for the global scope)

`!` $\Rightarrow$ (no vars)

`γ1` $\Rightarrow$ { a }

`γ2` $\Rightarrow$ { a, b }

Explicit Scope Dependency via RECs [Kiselyov+ 2014, Lee+ 2026]

Classifiers (e.g. `!`, `γ1`, `γ2`) represent sets of available variables

- A binding form declares a classifier for the new scope
- A quote carries an existing classifier to specify the scope it depends on
- A code type `<T@γ>` carries a classifier to specify the scope it depends on

```
<int@!> | <! let a@γ1 = 2 in
          let b@γ2 = 3 in
          ${ <γ2 a + b > } >
          <int@γ2>
```

(`!` is reserved for the global scope)

`!` $\Rightarrow$ (no vars)

`γ1` $\Rightarrow$ { `a` }

`γ2` $\Rightarrow$ { `a`, `b` }

Detecting Scope Extrusion by RECs [Kiselyov+ 2014, Lee+ 2026]

This mutable state stores
quoted expression under the global scope

This part has type $\langle \text{int}@_\gamma \rangle$
because x is valid under γ

```
let r: <int@!> ref = ref <! 1 > in
let _ = <! fun x@γ ->
    $(r := <γ x >; <γ x >)
    > in
deref r (* => < x > *)
```

Type-Error: Inconsistent classifier info

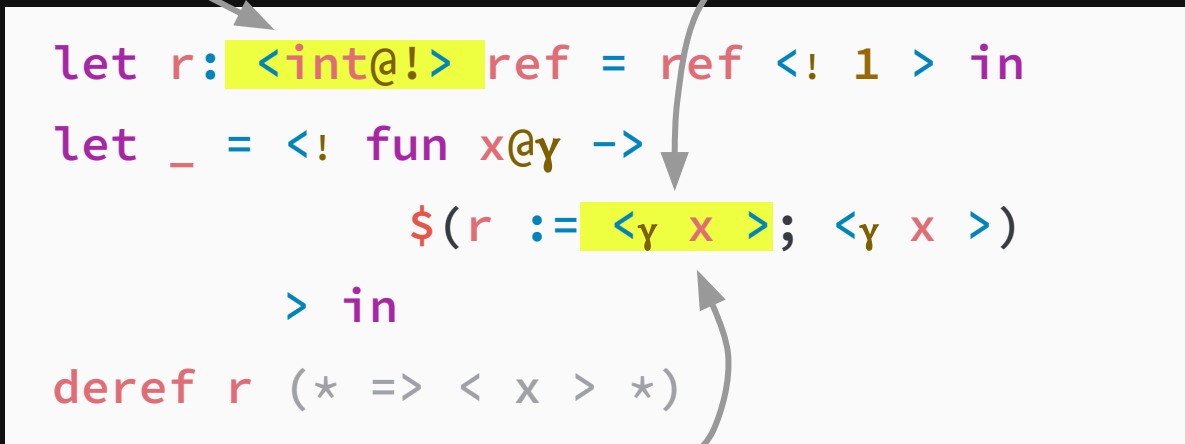
```

r      : < int@! > ref
< x >  : < int@γ >
```

Detecting Scope Extrusion by RECs [Kiselyov+ 2014, Lee+ 2026]

This mutable state stores
quoted expression under the global scope

This part has type $\langle \text{int}@_\gamma \rangle$
because x is valid under γ



```

let r: <int@!> ref = ref <! 1 > in
let _ = <! fun x@_gamma ->
    $(r := <_gamma x >; <_gamma x >)
    > in
deref r (* => < x > *)
  
```

Type-Error: Inconsistent classifier info

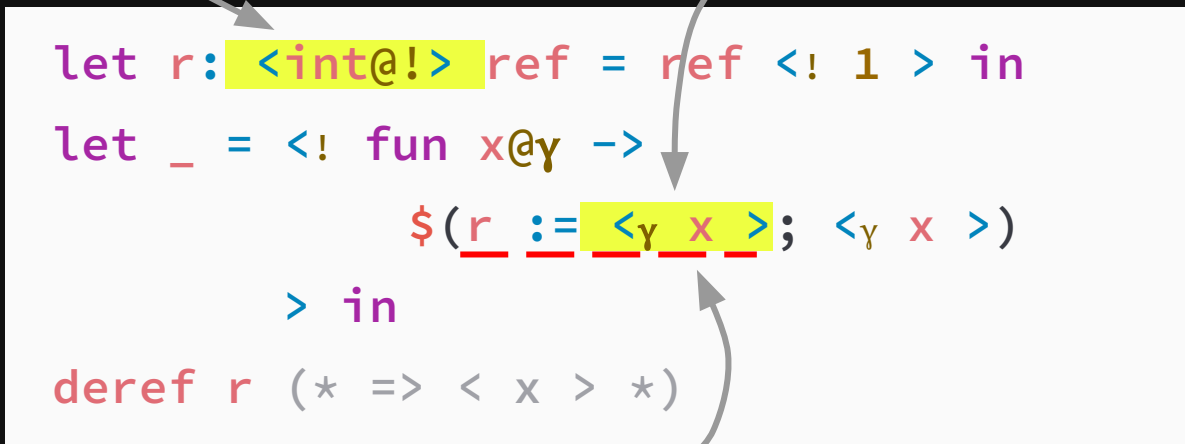
```

r      : < int@! > ref
< x >  : < int@_gamma >
  
```

Detecting Scope Extrusion by RECs [Kiselyov+ 2014, Lee+ 2026]

This mutable state stores
quoted expression under the global scope

This part has type $\langle \text{int}@_\gamma \rangle$
because x is valid under γ



```

let r: <int@!> ref = ref <! 1 > in
let _ = <! fun x@_gamma ->
    $(r := <_gamma x >; <_gamma x >)
    > in
deref r (* => < x > *)
  
```

Type-Error: Inconsistent classifier info

```

r      : < int@! > ref
< x >  : < int@_gamma >
  
```

Refined Environment Classifiers

[Kiselyov+ 2014, Lee+ 2026]

Reconstruct



- Two-level staging
- Does NOT support run with cross-stage persistence
- Code generators are not reusable

Refined² Environment Classifiers

Our Proposal

- Multi-level staging
- Support run with cross-stage persistence
- Reusable code generators

Cross-Stage Persistence [Taha+ 2000]

sqr is defined at
runtime stage

..... but used at
future stage inside quote

```
let sqr a = a * a in
let code = < sqr 10 > in
run code
```

run lifts expression level,
and levels of both sqr align

Point: Earlier formalizations
did not support run nor
cross-stage persistence

Cross-Stage Persistence [Taha+ 2000]

sqr is defined at
runtime stage

..... but used at
future stage inside quote

```
let sqr a = a * a in  
let code = < sqr 10 > in  
run code
```

run lifts expression level,
and levels of both **sqr** align

Point: Earlier formalizations
did not support **run** nor
cross-stage persistence

Cross-Stage Persistence [Taha+ 2000]

sqr is defined at
runtime stage

..... but used at
future stage inside quote

```
let sqr a = a * a in  
let code = < sqr 10 > in  
run code
```

run lifts expression level,
and levels of both **sqr** align

Point: Earlier formalizations
did not support **run** nor
cross-stage persistence

Cross-Stage Persistence [Taha+ 2000]

sqr is defined at
runtime stage

..... but used at
future stage inside quote

```
let sqr a = a * a in  
let code = < sqr 10 > in  
run code
```

run lifts expression level,
and levels of both **sqr** align

Point: Earlier formalizations
did not support **run** nor
cross-stage persistence

Typing Cross-Stage Persistence with Classifiers

γ is the *runtime-level* classifier for `sqr`

This *future-level* quoted expression depends on γ

```
let sqr@ $\gamma$  a = a * a in  
let code:<int@ $\gamma$ > = < $\gamma$  sqr 10> in  
run code
```

This `run` types because classifier info aligns

- It is located inside the scope of γ
- The type of `code` is <int@ γ >

Novelty: R2ECs allow classifiers to be shared across different stages

Typing Cross-Stage Persistence with Classifiers

γ is the *runtime-level* classifier for `sqr`

This *future-level* quoted expression depends on γ

```
let sqr@ $\gamma$  a = a * a in  
let code:<int@ $\gamma$ > = < $\gamma$  sqr 10> in  
run code
```

This run types because classifier info aligns

- It is located inside the scope of γ
- The type of code is <int@ γ >

Novelty: R2ECs allow classifiers to be shared across different stages

Typing Cross-Stage Persistence with Classifiers

γ is the *runtime-level* classifier for `sqr`

This *future-level* quoted expression depends on γ

```
let sqr@ $\gamma$  a = a * a in  
let code:<int@ $\gamma$ > = < $\gamma$  sqr 10 > in  
run code
```

This run types because classifier info aligns

- It is located inside the scope of γ
- The type of code is <int@ γ >

Novelty: R2ECs allow classifiers to be shared across different stages

Typing Cross-Stage Persistence with Classifiers

γ is the *runtime-level* classifier for `sqr`

This *future-level* quoted expression depends on γ

```
let sqr@ $\gamma$  a = a * a in  
let code:<int@ $\gamma$ > = < $\gamma$  sqr 10 > in  
run code
```

This run types because classifier info aligns

- It is located inside the scope of γ
- The type of code is <int@ γ >

Novelty: R2ECs allow classifiers to be **shared across different stages**

Issue with Reusable Code Generators

The original RECs do not type this program 😱

```
(* defined as a library func *)  
let gen_add1 a = < 1 + ~a > in  
... < fun b → ~( gen_add1 < b > ) > ...
```

Issue with Reusable Code Generators

The original RECs do not type this program 😱

This arg of `gen_add1`
has type `<int@γ>`

```
(* defined as a library func *)

let gen_add1 (a:<int@?>): <int@?> = ... in
... < fun b@γ → ~( gen_add1 <γ b >) > ...
```

Then, we expect `gen_add1` to have
the type `<int@γ>→<int@γ>`
...However, `γ` is not declared at this point

Issue with Reusable Code Generators

The original RECs do not type this program 😱

This arg of `gen_add1`
has type `<int@γ>`

```
(* defined as a library func *)

let gen_add1 (a:<int@?>): <int@?> = ... in
... < fun b@γ → ~( gen_add1 <γ b > ) > ...
```

Then, we expect `gen_add1` to have
the type `<int@γ>→<int@γ>`
...However, `γ` is not declared at this point

Issue with Reusable Code Generators

The original RECs do not type this program 😱

This arg of `gen_add1`
has type `<int@γ>`

```
(* defined as a library func *)

let gen_add1 (a:<int@?>): <int@?> = ... in
... < fun b@γ → ~( gen_add1 <γ b >) > ...
```

Then, we expect `gen_add1` to have
the type `<int@γ>→<int@γ>`
...However, `γ` is not declared at this point

Our Proposal: Polymorphic Classifiers

We avoid this issue by **bounded polymorphism over classifiers**.

Declares a *polymorphic classifier* δ with the bound $!$,
and `gen_add1` has type $\forall \delta : >!. <int @ \delta > \rightarrow <int @ \delta >$



```
(* defined as a library func *)
let gen_add1 [ $\delta : >!$ ] ( $a : <int @ \delta >$ ):  $<int @ \delta >$ 
    =  $< \delta \ 1 + \sim a >$  in
... < fun  $b @ \gamma \rightarrow \sim( \text{gen\_add1 } [\gamma] <_{\gamma} b > ) >$  ...
```

Then we instantiate δ with γ ,
and safely pass a argument of $<int @ \gamma >$



Novelty: First formalization of
polymorphic classifiers!

Our Proposal: Polymorphic Classifiers

We avoid this issue by **bounded polymorphism over classifiers**.

Declares a *polymorphic classifier* δ with the bound $!$,
and `gen_add1` has type $\forall \delta :>!. <int @ \delta> \rightarrow <int @ \delta>$



```
(* defined as a library func *)
let gen_add1 [delta :>!] (a :<int @ delta>) :<int @ delta>
  = <delta 1 + ~a > in
... < fun b @ gamma -> ~( gen_add1 [gamma] <gamma b >) > ...
```

Then we instantiate δ with γ ,
and safely pass a argument of $<int @ \gamma>$



Novelty: First formalization of
polymorphic classifiers!

Our Proposal: Polymorphic Classifiers

We avoid this issue by **bounded polymorphism over classifiers**.

Declares a *polymorphic classifier* δ with the bound $!$,
and `gen_add1` has type $\forall \delta :>!. <int @ \delta> \rightarrow <int @ \delta>$



```
(* defined as a library func *)
let gen_add1 [delta: > !] (a: <int @ delta>): <int @ delta>
    = <delta 1 + ~a > in
... < fun b @ gamma -> ~( gen_add1 [gamma] <gamma b >) > ...
```

Then we instantiate δ with γ ,
and safely pass a argument of $<int @ \gamma>$

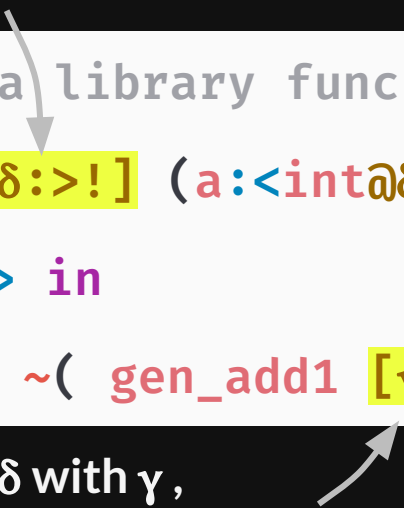


Novelty: First formalization of
polymorphic classifiers!

Our Proposal: Polymorphic Classifiers

We avoid this issue by **bounded polymorphism over classifiers**.

Declares a *polymorphic classifier* δ with the bound $!$,
and `gen_add1` has type $\forall \delta :>!. <\text{int}@_\delta> \rightarrow <\text{int}@_\delta>$



```

(* defined as a library func *)
let gen_add1 [δ: > !] (a: <int@δ>): <int@δ>
    = <δ 1 + ~a > in
... < fun b@γ → ~( gen_add1 [γ] <γ b >) > ...

```

Then we instantiate δ with γ ,
and safely pass a argument of $<\text{int}@_\gamma>$

Novelty: First formalization of
polymorphic classifiers!

Refined Environment Classifiers

[Kiselyov+ 2014, Lee+ 2026]

- Two-level staging
- Does NOT support cross-stage persistence
- Code generators are not reusable

Reconstruct



Refined² Environment Classifiers

Our Proposal

- Multi-level staging
- Support (scoped-variant of) cross-stage persistence
- Reusable code generators

By tracking classifier scopes
(omitted in this talk)

By sharing classifiers
across different stages

By introducing
polymorphism over classifiers

These refinements are achieved by reconstructing the REC-based typing discipline from ground up.

Check out our paper for ...

- Illustrative examples
- Type system
- Definitional Interpreter
- Soundness proofs
- Rocq formalization
- **Interactive web demo** with various examples



<https://bit.ly/4hwhgtJ>

(Yuito Murase is on job market)

Appendix

RECs are NOT Sufficient for Nested Quotes

```

let r: < int@! > ref = ref <! 1 > in
let _ = <! <! fun x@y →
    $$ (r := <! let y = <y x >; 1 >;
        <! <y x > >)
    > > in
deref r (* ⇒ < let y = < x >; 1 > *)

```

RECs are NOT Sufficient for Nested Quotes

```
let r: < int@! > ref = ref <! 1 > in
```

```
let _ = <! <! fun x@y →
```

```
    $$ (r := <! let y = <y x >; 1 >;
```

```
        <! <y x > >)
```

<int@!>, but has free variables

```
    > > in
```

```
deref r (* ⇒ < let y = < x >; 1 > *)
```

RECs are NOT Sufficient for Nested Quotes

```
let r: < int@! > ref = ref <! 1 > in
```

```
let _ = <! <! fun x@y →
```

```
    $$ (r := <! let y = <y x >; 1 >;
```

```
        <! <y x > >)
```

<int@!>, but has free variables

```
    > > in
```

Undefined variable 😞

```
deref r (* ⇒ < let y = < x >; 1 > *)
```

Typed program leads to
scope extrusion

Diagnosis of the Issue

```
let r: < int@! > ref = ref <! 1 > in
```

```
let _ = <! <! fun x@y →
```

No variables/classifiers are
defined at !

```
  $$ (r := <! let y = <y x >; 1 >;
```

```
        <! <y x > >)
```

```
  > > in
```

```
deref r (* ⇒ < let y = < x >; 1 > *)
```

Diagnosis of the Issue

```
let r: < int@! > ref = ref <! 1 > in
```

```
let _ = <! <! fun x@y →
```

```
  $$ (r := <! let y = <y x >; 1 >;
```

```
    <! <y x > >)
```

```
  > > in
```

```
deref r (* ⇒ < let y = < x >; 1 > *)
```

No variables/classifiers are defined at !

Then it seems illegal to use y here?

Refined Environment Classifiers

[Kiselyov+ 2014, Lee+ 2026, etc]

Track
variable scopes



Refined² Environment Classifiers

Our Proposal

Track
variable scopes
and classifier scopes

Tracking Variable Scopes and Classifier Scopes



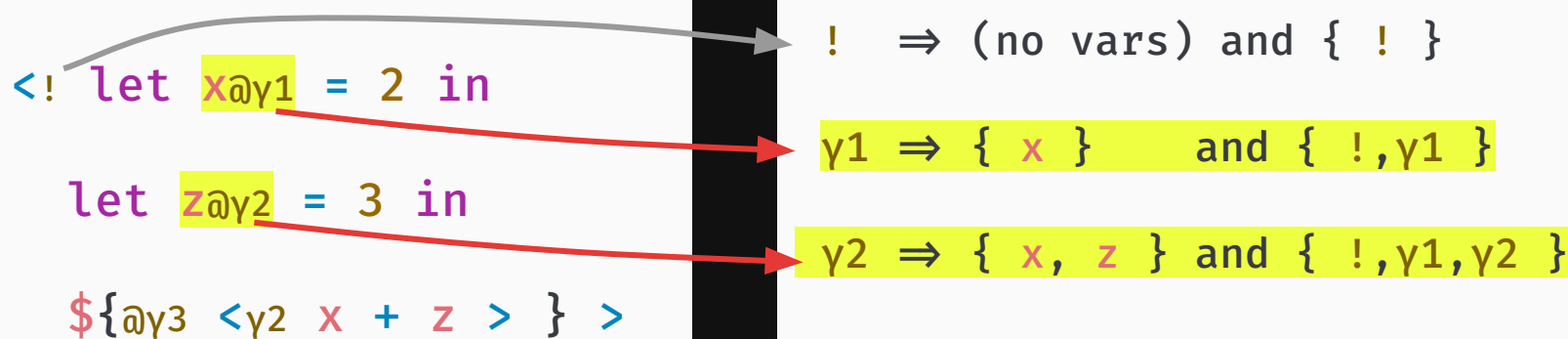
```
<! let x@γ1 = 2 in
```

```
  let z@γ2 = 3 in
```

```
  ${@γ3 <γ2 y + z > } >
```

! $\Rightarrow$ (no vars) and { ! }

Tracking Variable Scopes and Classifier Scopes



```
<! let x@γ1 = 2 in
```

```
let z@γ2 = 3 in
```

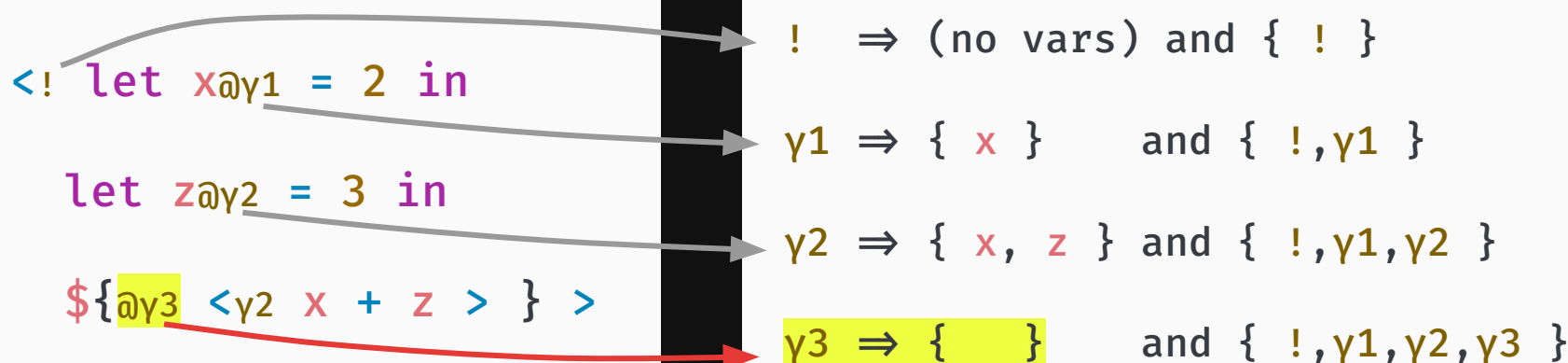
```
$ { @γ3 <γ2 x + z > } >
```

! $\Rightarrow$ (no vars) and { ! }

$\gamma_1 \Rightarrow \{ x \}$ and { !, γ_1 }

$\gamma_2 \Rightarrow \{ x, z \}$ and { !, γ_1, γ_2 }

Tracking Variable Scopes and Classifier Scopes



```
<! let x@y1 = 2 in
```

```
let z@y2 = 3 in
```

```
${@y3 <y2 x + z > } >
```

! $\Rightarrow$ (no vars) and { ! }

y1 $\Rightarrow$ { x } and { !, y1 }

y2 $\Rightarrow$ { x, z } and { !, y1, y2 }

y3 $\Rightarrow$ { } and { !, y1, y2, y3 }

Tracking Variable Scopes and Classifier Scopes



```
<! let x@γ1 = 2 in
```

```
let z@γ2 = 3 in
```

```
${@γ3 <γ2 x + z > } >
```

! $\Rightarrow$ (no vars) and { ! }

$\gamma_1 \Rightarrow$ { x } and { !, γ_1 }

$\gamma_2 \Rightarrow$ { x, z } and { !, γ_1 , γ_2 }

$\gamma_3 \Rightarrow$ { } and { !, γ_1 , γ_2 , γ_3 }

Tracking Variable Scopes and Classifier Scopes



```
<! let x@γ1 = 2 in
```

```
let z@γ2 = 3 in
```

```
${@γ3 <γ2 x + z > } >
```

! $\Rightarrow$ (no vars) and { ! }

$\gamma_1 \Rightarrow$ { x } and { !, γ_1 }

$\gamma_2 \Rightarrow$ { x, z } and { !, γ_1 , γ_2 }

$\gamma_3 \Rightarrow$ { } and { !, γ_1 , γ_2 , γ_3 }

R2ECs detects Scope Extrusion of Nested Quotes

```
let r: < int@! > ref = ref <! 1 > in
```

```
let _ = <! <! fun x@y →
```

```
    $$ (r := <! let y = <y x >; 1 >;
```

```
        <! <y x > >)
```

```
    > > in
```

```
deref r (* ⇒ < < let y = < x >; 1 > > *)
```

Type error: y is ill-scoped under !