Refined^2 Environment Classifiers

Elaborating Lexical Scoping in Multi-Stage Programming

Yuito Murase
Supervised by Atsushi Igarashi
Kyoto University

Motivation

Lexical scoping in multi-stage programming is NOT trivial

when **evaluation stages can change dynamically** or **program has side-effect**

A ***quote*** generates a code fragment (of a future stage)

An **unquote** embeds a code fragment to another code fragment

Run-time evaluation executes a future-stage code fragment under the current environment

```
`{ let x = 1 in
   ~{ let y = `{ x } in
      `{ ~y + ~y } } }
```
→ `{ let x = 1 in x + x }` — well-scoped

```
let f x =
  1 + x in
`{ f 1 }
```
(cross-stage use of vars.)
→ `{ f 1 }` — ill-scoped

```
let f x =
  1 + x in
run `{ f 1 }
```
→ 2 — well-scoped (↳ CROSS-STAGE PERSISTENCE)

```
let r = ref `{ 1 } in
`{ let y = 2 in
   ~{ r := `{ y };
      `{ 1 } } };
r!
```
→ `{ y }` — ill-scoped (↳ SCOPE EXTRUSION)

Existing MSP type systems are NOT precise enough!

[Davies '17]	accept	reject	**reject**	**accept**
[Hanada+ '14]	accept	reject	accept	**accept**
[Rhiger '12]	accept	reject	**reject**	reject
OURS	accept	reject	accept	reject

Approach

We develop a novel type system that annotates programs and types with **explicit scope information** (a.k.a. **Refined Environment Classifiers(REC)**) [Kiselyov+ '16]

similar to lifecime in C Rust?

We elaborated theory for scope-safety using REC

1 Each binding introduce a new scope, represented by a classifeir

```
let f@g1:int→int =
  fn(z@g2:int)→ z in
let x@g3:int = 1 in
f x
```
(g2 ⊑ g1, g3 ⊑ g1, ! ⊑ g1, ! ⊑ g2, ! ⊑ g3)

g1 <: g2

"g2 is inside of g1"

```
!  ↦ (global scope)
g1 ↦ !  + f
g2 ↦ g1 + z
g3 ↦ g1 + x
```
regarded as set of (valid) vars.

2 A quote switches current scope, and capture the scope in its type

```
let y@h = 2 in
...
`{@h y + y}
```
(y : int@h) → type is <int@h>

3 A variable can be used when it is valid under current scope

```
let x@g = 1 in
let y@h = x in
run `{@g x + y }
```
y : int@h, h ⊑ g, x : int@g, g ⊑ g → invalid

4 run/unquote can be used if scopes are consistent

```
let x@g = 1 in
let y@h = 2 in
run `{@g x }
```
(h ⊑ g) → 👍 types because g<:h

5 A function cannot return code with its own scope

```
fn(z@g)→`{@g z+1 }
```
<int@g> 🚫

```
fn(z@g)→run `{@g z+1 }
```
int 👍

6 Bounded polymorphism over classifiers is allowed

```
let dbl:[g:>!]<int@g>→<int@g>=
  fn [g:>!](x:<int@g>) →
    `{@g ~x + ~x } in
`{@! let y@h:int = 1 in
     ~{ dbl[h] `{@h y } } }
```

Comparison	Original Proposal [Kiselyov+ '16]	Our Proposal
Syntax for Code Gen.	Code Combinator	Quasi Quote
Levels	Two-level	Multi-level
Safety w/ ref	✅	✅*
Safety w/ CSP	❌	✅*
Polymorphism	❌	✅

* to be proved

Formalization

Type System is inspired by *Kriple/Fitch-style modal calculi* and *labelled deductive systems* [Reed+ '09] [Marin+ '21]

[Davies+ '01] [Clouston '18]

$$\textbf{Context}\quad \Gamma, \Delta, \ldots \quad ::= \quad \emptyset \mid \Gamma, (x : A)^{\gamma} \mid \Gamma, \text{🔓}_{\gamma} \mid \Gamma, \text{🔒}_{k} \mid \Gamma, \gamma_1 \succeq \gamma_2$$

$$\text{Var}\ \frac{\vdash \Gamma : \mathbf{ctx} \qquad (x : A)^{\gamma} \in \Gamma \qquad \Gamma \vdash \gamma \preceq \mathbf{cur}(\Gamma)}{\Gamma \vdash x : A}$$

$$\text{Quo}\ \frac{\Gamma, \text{🔓}_{\gamma} \vdash M : A}{\Gamma \vdash \mathbf{quo}\{^{\gamma} M\} : [A]^{\gamma}} \qquad \text{Unq}\ \frac{\Gamma, \text{🔒}_{k} \vdash M : [A]^{\gamma} \qquad \Gamma \vdash \gamma \preceq \mathbf{cur}(\Gamma)}{\Gamma \vdash \mathbf{unq}_{k}\{M\} : A}$$

$$\text{Abs}\ \frac{\Gamma, (x : A)^{\gamma} \vdash M : B \qquad \gamma \notin \mathbf{FC}(B)}{\Gamma \vdash \lambda(x : A)^{\gamma}.M : A \to B} \qquad \text{CAbs}\ \frac{\Gamma, \gamma_1 \succeq \gamma_2 \vdash M : A}{\Gamma \vdash \Lambda(\gamma_1 \succeq \gamma_2).M : \forall(\gamma_1 \succeq \gamma_2).A}$$

We elaborated semantics with *explicit substitution* so that substitutions take place only at runtime stage

[Nakata+ '09] [Accattoli+ '14]

```
 let x@g = 1 in run `{@g x+x }
→  (run `{@ x+x })[x@g := 1]
→  (x+x)[x@g := 1]
→* (1+1)[x@g := 1]
→  2[x@g := 1]
→  2
```

```
let x@g = 1 in `{@ x+x }
→ `{@g x+x }[x@g := 1]
```
(evaluation is stuck)

Related Work

Existing MSP calculi

Scopes are strictly separated by stages (hence, prone to dynamic statges)

Our calculus

REC allow different stages to share scopes

→ Capability to type CSP

Contextual Modal Types [Nanevski+ '08, Murase+ '23]

```
`{ x,y. x+y }
 : [int, int |- int]
```
Code types carry typing ctx. (sound with **run** and **ref**)

```
`{ let x = 1 in
  ~{ f `{x. x } }[x] }
```
Quotes must close all free vars, which blocks binding beyond quotes

Current Progress

- ✅ Definition of type system
- ✅ Definition of operational semantics
- WIP Soundness Proofs (Preservation + Progress)
- WIP Extension with mutable state
- WIP Translation from/to other calculi

Future Direction

- **Classifier inference for annotation-free staged programs** (like MetaOCaml / Scala 3)
- Analytic metaprogramming (pattern match on code)
- Curry-Howard isomorphism (Extension of S4 modality?)