# Staging calculi

Oleg Kiselyov and Chung-chieh Shan

Our motivation: program generation

- flexible: scope mismatch, let insertion
- convenient: first-class delimited continuations
- safe: prevent scope extrusion

Our need: a typed staging calculi

- call by value, with small-step semantics
- splice open code and run closed code
- cross-stage persistence

Embarrassment of riches: $\lambda^{U}$, $\lambda^{\Box}$, $\lambda^{\mathsf{S4}}$, $\lambda^{\bigcirc}$, $\lambda^{\bigcirc\Box}$, $\lambda^{a}$, $\lambda^{i}_{\mathsf{let}}$

- Relationship unclear
- None satisfactory
- No effects

# Goals

With hindsight:

1. Compile a definite reference to staged calculi, to reduce rummaging through the literature.
   - Which calculus supersedes which?
   - How do they compare?
2. Design a common calculus or calculi substrate.
   - At least share notation.
   - Add or remove features such as ‘run’ and polymorphism.
3. Model real implementations such as MetaOCaml.
4. Compile a database of mechanized calculi.

# Motivation

We want to prevent

```
let code = let x = ref .<1>. in
           let _ = .<fun v -> .~(x := .<v>.; .<()>.)>. in
           !x
```

- `val code: ('a, int) code = .<v_1>.`

# Motivation

We want to prevent

```
let code = let x = ref .<1>. in
           let _ = .<fun v -> .~(x := .<v>.; .<()>.)>. in
           !x
```

- `val code: ('a, int) code = .<v_1>.`

```
.!code
```

- `Unbound value v_1`

  `Exception: Trx.TypeCheckingError.`

This example is contrived; we can show a more realistic example.

These examples occur in MetaOCaml, which has mutable state, exceptions, and first-class delimited continuations.

No real model.

# Comparison

| | Typed | Splice | Run | Persist | Call by | Steps |
|---|---|---|---|---|---|---|
| $\lambda^{U}$ | no | yes | yes | values | name | big |
| $\lambda^{\Box}$, $\lambda^{S4}$ | yes | no | yes | no | ? | ? |
| $\lambda^{\bigcirc}$ | yes | yes | no | no | ? | ? |
| $\lambda^{\bigcirc\Box}$ | yes | yes | some | variables | value | small |
| $\lambda^{a}$ | yes | yes | yes | yes | name | big |
| $\lambda^{i}_{\mathsf{let}}$ | yes | yes | yes | yes | name | big |

# Comparison

| | Typed | Splice | Run | Persist | Call by | Steps |
|---|---|---|---|---|---|---|
| $\lambda^{U}$ | no | yes | yes | values | name | big |
| $\lambda^{\Box}$, $\lambda^{S4}$ | yes | no | yes | no | ? | ? |
| $\lambda^{\bigcirc}$ | yes | yes | no | no | ? | ? |
| $\lambda^{\bigcirc\Box}$ | yes | yes | some | variables | value | small |
| $\lambda^{a}$ | yes | yes | yes | yes | name | big |
| $\lambda^{i}_{\text{let}}$ | yes | yes | yes | yes | name | big |

Cross-stage persistence:

- only variables? only values?
- evaluated when code is built? is run?
- open code? `.<fun x -> .%(.<x>.)>.`
- restricted by type?

Mechanization: how to type the context $(a)$`.<fun x -> .~[]>.`$^{a}$

# Goals

With hindsight:

1. Compile a definite reference to staged calculi, to reduce rummaging through the literature.
   - Which calculus supersedes which?
   - How do they compare?
2. Design a common calculus or calculi substrate.
   - At least share notation.
   - Add or remove features such as ‘run’ and polymorphism.
3. Model real implementations such as MetaOCaml.
4. Compile a database of mechanized calculi.