Safe Imperative Metaprogramming with Contextual Linear References

Maite Kramarz
University of Toronto

Scan for references!

1. MetaML & Scope Extrusion

⟨e⟩	$e	run e
Quoting: Creates a code fragment to be evaluated later	**Splicing**: Assembles code fragments into larger ones	**Running**: Executes a code fragment in current stage

```
let l = mut <1>
(* val l = ... : Ref Code Int *)
let f = <fun x => $(l := <x+1>; <2>)>
(* val f = <fun x => 2> : Code (Int -> Int) *)
let c = !l
(* val c = <x+1> : Code Int *)
run c
(* error: x is a free variable *)
```

Naive implementation of references is unsound
(variables can escape their scopes!)

2. How can this be prevented?

BER MetaOCaml:
"Thou shalt be very careful about putting code fragments into references, as this may cause errors."

Calcagno et al. (2003):
"Thou shalt not put non-dead code fragments containing dynamic variables into a reference."

This Abstract:
"Thou shalt not access a reference containing a code fragment until its surrounding scope returns it to you."

3. Linearity

Why

- Linearity helps us model a kind of ownership.
- Once a reference is altered, the linear type system only allows us to interact it with it again when that scope is evaluated (no extrusion!).
- Note we still need a way to track open code references.

How

- We believe we are the first to combine a MetaML-style system with linear types!
- We adapt our system from Taha et al. and Walker.
- Rather than separate read and write operations, we follow Walker and Morissett et al. in using a 'swap.'

4. Contextualizing

Our semantics tracks context, stage, and variable store:

$$(S; \Gamma; e) \xrightarrow{n} (S'; \Gamma; e')$$

While S represents concrete values at stage 0, Γ tracks the higher-staged typing data for binders we pass under.

$$S ::= \emptyset \mid S, x_\sigma \mapsto v[\Gamma]$$

1. References carry a context to type higher-staged bindings.
2. Substitutions on a reference are deferred in σ to avoid mutation inside the heap.

5. Putting it Together

1. We propose a novel combination of MetaML style MSP with linear types.
2. This can statically forbid scope extrusion, but requires additional theoretical machinery.
3. MSP + linearity can help programmers write code that bridges the gap between abstract and performant.