

When do staging annotations preserve semantics?

IFIP WG 2.11 Twenty-sixth Meeting

Niagara-on-the-Lake, Ontario, Canada, Oct 2026

Guannan Wei

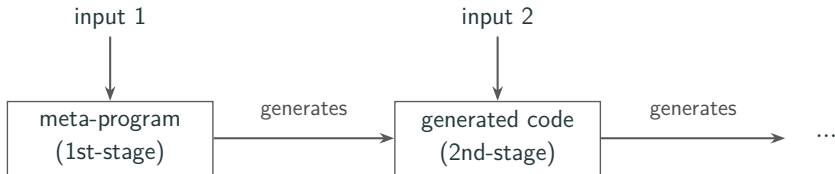
with Jun Tan, Dinghong Zhong, and Robi Jenik



Motivation

Multi-stage programming (MSP) is a meta-programming paradigm:

- divides code that executes in multiple stages
- generated code is specialized to the inputs available at earlier stages



Stage Annotations

Programmers can add annotations to mark whether the code is generated vs executed, e.g. in MetaML:

- Quotation `<e>`: convert an expression into code value
- Splice `~e`: insert the code value into the surrounding code template

Stage Annotations

Programmers can add annotations to mark whether the code is generated vs executed, e.g. in MetaML:

- Quotation `<e>`: convert an expression into code value
- Splice `~e`: insert the code value into the surrounding code template

Similar variants adopted by many popular languages:

- Lisp family: ``(...)` and `,e`
- BER MetaOCaml: `.< ... >.` and `~e`
- Scala 3: `'{ ... }` and `$e`
- (Typed) Template Haskell: `[| ... |]` and `$e`; `[|| ... ||]` and `$$e`
- F#, Julia, Rust, etc.

Why You May Want to Stage a Program?

Mostly, performance: the generated program could be used as a *faster* replacement for the original unstaged program.

Program specialization & partial evaluation:

Efficiency. Specializing a function on a fixed argument can lead to dramatic efficiency gains. Program generators can provide the same efficiency gains that partial evaluation does.

Multi-Stage Programming with Explicit Annotations.

Walid Taha & Tim Sheard. PEPM '97

Stage Annotations Can Break Semantics

Performance is a good thing only when the generated program behaves the same as the original unstaged one.

But, this is not guaranteed in quotation-based staging!

Stage Annotations Can Break Semantics

Consider the unstaged **imperative** power function:

```
let rec power (n: int) (x: int) (res: int ref): int =  
  if n = 0 then !res  
  else (res := x * !res; power (n - 1) x res)
```

```
fun n -> fun x -> power n x (ref 1)
```

- When calling the power function, provide a mutable reference cell (ref 1)
- Each iteration multiplies x into res; after n iterations, !res stores x^n

Let's Add Staging Annotations!

Decide types based on binding-time information:

```
let rec power (n: int) (x: int code) (res: (int ref) code): int code =  
  if n = 0 then ...  
  else ...
```

- Exponent n is known in the first stage, so keep its type `int`.
- x , res , and the return value are known later, so their types should be `code`.

Let's Add Staging Annotations!

Quote the computation that happens later and splice the code-valued components into each branch:

```
let rec power (n: int) (x: int code) (res: (int ref) code): int code =  
  if n = 0 then < !(~res) >  
  else < ~res := ~x * !(~res);  
        ~(power (n - 1) x res) >
```

- Splice code fragments: ~res inserts the mutable reference, and ~x inserts base.
- ~(power (n - 1) x res) is unrolled during code generation.

Specializing the exponent n to 3.

Then let's specialize the exponent n to 3!

```
let rec power (n: int) (x: int code) (res: (int ref) code): int code =  
  if n = 0 then < !(~res) >  
  else < ~res := ~x * !(~res);  
        ~(power (n - 1) x res) >
```

```
< fun x -> ~(power 3 <x> <(ref 1)>) >
```

- The generated function still takes x at run time.
- Quote $\langle x \rangle$ and $\langle (\text{ref } 1) \rangle$ as code, as they are residual code.

Let's Add Staging Annotations!

If we run the code generator, things will go wrong...

```
< fun x ->  
  (ref 1) := x * !(ref 1);  
  (ref 1) := x * !(ref 1);  
  (ref 1) := x * !(ref 1);  
  !(ref 1) >
```

Let's Add Staging Annotations!

If we run the code generator, things will go wrong...

```
< fun x ->  
  (ref 1) := x * !(ref 1);  
  (ref 1) := x * !(ref 1);  
  (ref 1) := x * !(ref 1);  
  !(ref 1) >
```

- Every occurrence of `res` duplicates a fresh allocation.
- The final value reads from a fresh cell too: the function returns `1`, instead of x^3 .

This is clearly not the power function we intended.

What Should We Expect from Staging?

The programmer can manually insert a let-binding or using MetaOCaml's `genlet` to avoid duplication. But it is cumbersome, because if we remove the staging annotation, we get exactly the original program!

```
let rec power (n: int)
  (x: int code)
  (res: (int ref) code): int code =
  if n = 0 then < !(~res) >
  else < ~res := ~x * !(~res);
      ~(power (n - 1) x res) >
  < fun x -> ~(power 3 <x> <(ref 1)>) >
```

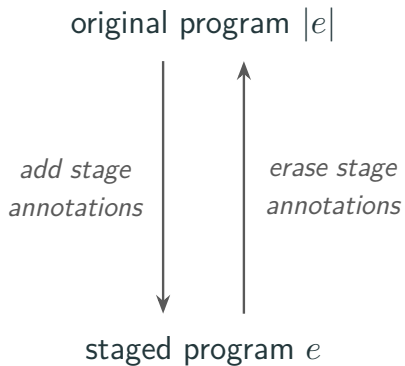
erase stage
annotations



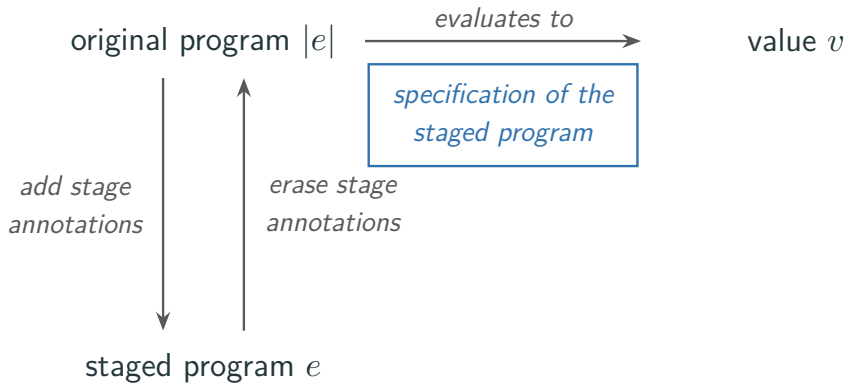
```
let rec power (n: int)
  (x: int)
  (res: int ref): int =
  if n = 0 then !res
  else (res := x * !res;
      power (n - 1) x res)
  fun x -> power 3 x (ref 1)
```

What goes wrong and what should we expect from staging?

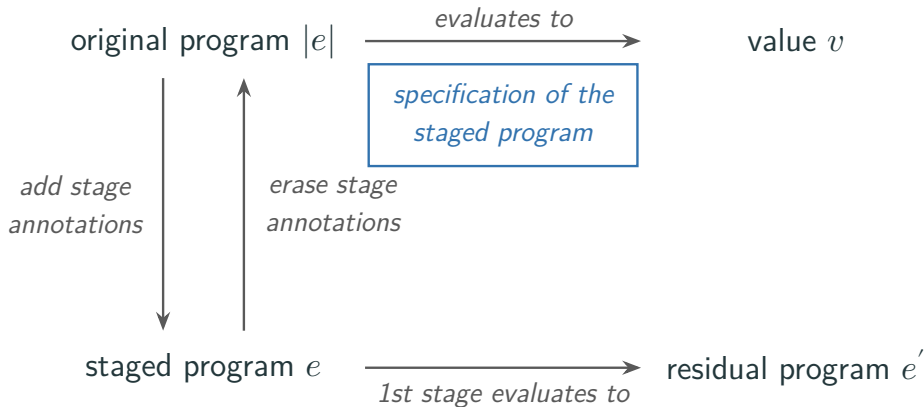
Staging and Evaluation



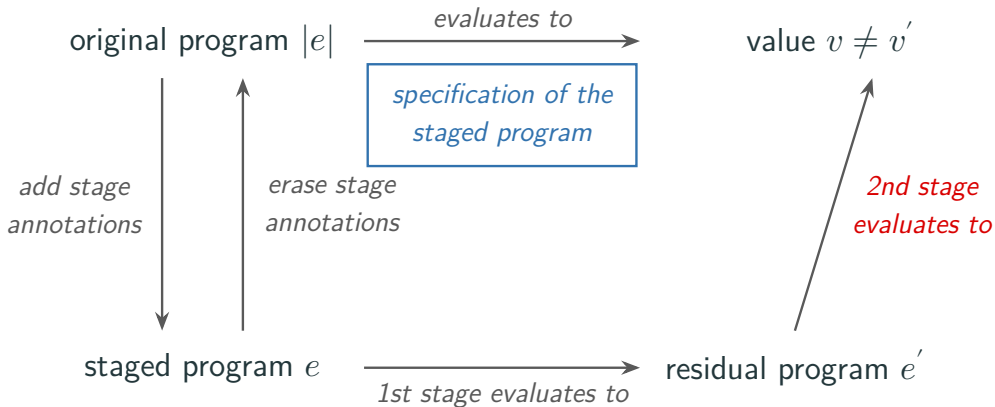
Staging and Evaluation



Staging and Evaluation

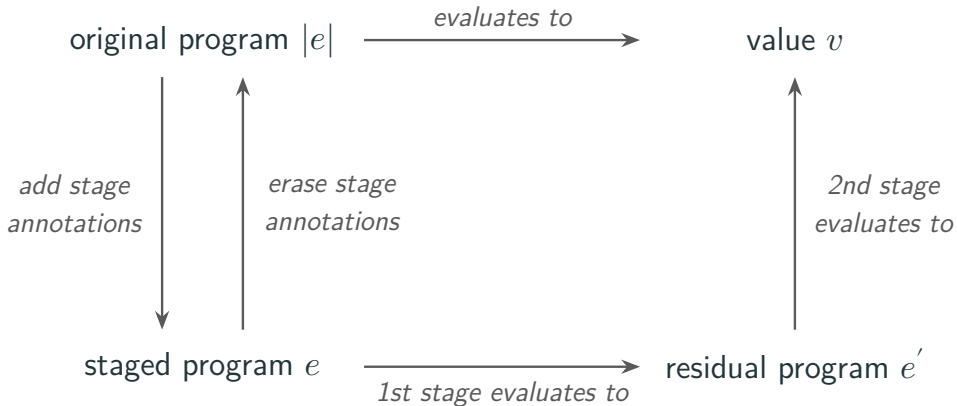


Staging and Evaluation



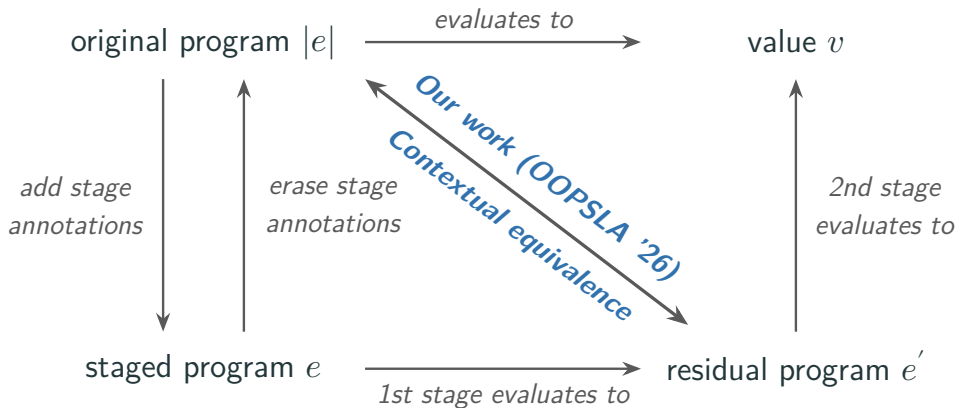
MetaML-style quotation/splicing manipulates syntax, but can also change program behavior and effects in call-by-value.

Semantics-Preserving Staging



In Inoue and Taha (ESOP '12, JFP '16), they asked the question
“*when do we know that erasing annotations preserves semantics?*”

Our Work: Designing Semantics-Preserving Staged Languages (OOPSLA '26)



Typed Semantics Preservation: If $\vdash e : \text{code } \tau$ and $e \rightarrow^* \text{code } e'$, then $|e| \simeq^{\text{ctx}} e' : \tau$.

Historical Context

Multi-stage programming has its roots in partial evaluation,
but MSP has not yet realized what partial evaluation promised:

	Binding-time information	Semantics preservation
Partial evaluation	automatic analysis, but conservative	✓

Historical Context

Multi-stage programming has its roots in partial evaluation,
but MSP has not yet realized what partial evaluation promised:

	Binding-time information	Semantics preservation
Partial evaluation	automatic analysis, but conservative	✓
MetaML-style multi-stage programming	manual annotations, but can be precise	✗

Historical Context

Multi-stage programming has its roots in partial evaluation,
but MSP has not yet realized what partial evaluation promised:

	Binding-time information	Semantics preservation
Partial evaluation	automatic analysis, but conservative	✓
MetaML-style multi-stage programming	manual annotations, but can be precise	✗
<i>This work</i>	manual annotations, but can be precise	✓

Two typed call-by-value two-stage calculi with semantics preservation:

- $\lambda_{|2|}$: a core staging calculus + run-time code generation + divergence
- $\lambda_{|2|}^{\text{ref}}$: $\lambda_{|2|}$ + 2nd-stage only mutable references

Two typed call-by-value two-stage calculi with semantics preservation:

- $\lambda_{|2|}$: a core staging calculus + run-time code generation + divergence
- $\lambda_{|2|}^{\text{ref}}$: $\lambda_{|2|}$ + 2nd-stage only mutable references

Three key ingredients:

- Dynamic semantics
 - A reduction semantics with automatic let-insertion to preserve semantics

Two typed call-by-value two-stage calculi with semantics preservation:

- $\lambda_{|2|}$: a core staging calculus + run-time code generation + divergence
- $\lambda_{|2|}^{\text{ref}}$: $\lambda_{|2|}$ + 2nd-stage only mutable references

Three key ingredients:

- Dynamic semantics
 - A reduction semantics with automatic let-insertion to preserve semantics
- Static semantics
 - Code generation is treated as an effect and tracked by a type-and-effect system
 - Types distinguish complete code values from intermediate fragments.

Two typed call-by-value two-stage calculi with semantics preservation:

- $\lambda_{|2|}$: a core staging calculus + run-time code generation + divergence
- $\lambda_{|2|}^{\text{ref}}$: $\lambda_{|2|}$ + 2nd-stage only mutable references

Three key ingredients:

- Dynamic semantics
 - A reduction semantics with automatic let-insertion to preserve semantics
- Static semantics
 - Code generation is treated as an effect and tracked by a type-and-effect system
 - Types distinguish complete code values from intermediate fragments.
- Semantics-preservation via binary Kripke-style logical relations

Instead of using syntactic quotation/splicing, $\lambda_{|2|}/\lambda_{|2|}^{\text{ref}}$ is a core **combinator-style staging calculus**.

$$\begin{array}{llll} n \in \mathbb{N} & x \in \text{Vars} & \text{Stages } s \in \{1, 2\} \\ \text{Terms } t ::= & () \mid n \mid x \mid \lambda x. t \mid \text{let } x = t_1 \text{ in } t_2 \mid \text{lift } t \mid \text{run } t \\ & \mid \text{app}^s t_1 t_2 \mid \text{fix}^s t \mid \text{ifz}^s t_1 t_2 t_3 \mid t_1 \oplus^s t_2 \mid \dots \end{array}$$

- Terms carry a marker 1 or 2 indicating the stage of evaluation.
- `lift` converts a first-stage value into a code value.
- Lifting a function will partially evaluate the function body (as in TDPE/NbE), roughly $(\text{code } A \rightarrow \text{code } B) \rightarrow \text{code}(A \rightarrow B)$.

The staged imperative power function in $\lambda_{|2|}^{\text{ref}}$:

```
let rec power (n: int) (x: int code) (res: (int ref) code) : int code =  
  if (n = 0) then !2res  
  else (res :=2 x *2 !2res;  
        power (n - 1) x res)  
  
lift (fun x -> power 3 x (ref2 (lift 1)))
```

- During the evaluation, second-stage constructs are automatically let-inserted into the residual code to force their evaluation and preserves their relative order.

```
<fun x ->  
  let r = ref 1 in  
  let _ = (r := x * !r) in  
  let _ = (r := x * !r) in  
  let _ = (r := x * !r) in  
  let v = !r in  
  v>
```

Dynamic Semantics and Let-insertion

We borrow the reduction semantics from Amin & Rompf (POPL '18), which introduces administrative terms (reflect) as markers for let-insertion:

Terms	t	$::=$	$\dots \mid g$
Admin. Terms	g	$::=$	$\text{code } t \mid \text{reflect } t \mid \text{let}_c x = t_1 \text{ in } t_2 \mid \lambda_c x. t$
Values	v	$::=$	$n \mid () \mid \lambda x. t \mid \text{code } t$

Selective Rules of the Reduction Semantics

stage 1 — ordinary CBV

$$\begin{array}{ll} (\text{ST-APP}^1) & \text{app}^1 (\lambda x.t) v \rightsquigarrow t[v/x] \\ (\text{ST-OP}^1) & n_1 \oplus^1 n_2 \rightsquigarrow \delta(\oplus, n_1, n_2) \end{array}$$

stage 2 — assemble code by reflect

$$\begin{array}{ll} (\text{ST-APP}^2) & \text{app}^2 (\text{code } t_1) (\text{code } t_2) \rightsquigarrow \text{reflect} (\text{app}^1 t_1 t_2) \\ (\text{ST-OP}^2) & (\text{code } t_1) \oplus^2 (\text{code } t_2) \rightsquigarrow \text{reflect} (t_1 \oplus^1 t_2) \end{array}$$

cross-stage — lift & run

$$\begin{array}{ll} (\text{ST-LAM}^\uparrow) & \text{lift } (\lambda x.t) \rightsquigarrow \lambda_c x.t[\text{code } x/x] \\ (\text{ST-RUN}) & \text{run } (\text{code } t) \rightsquigarrow t \end{array}$$

Selective Rules of the Reduction Semantics

stage 1 — ordinary CBV

$$\begin{array}{ll} (\text{ST-APP}^1) & \text{app}^1 (\lambda x.t) v \rightsquigarrow t[v/x] \\ (\text{ST-OP}^1) & n_1 \oplus^1 n_2 \rightsquigarrow \delta(\oplus, n_1, n_2) \end{array}$$

stage 2 — assemble code by reflect

$$\begin{array}{ll} (\text{ST-APP}^2) & \text{app}^2 (\text{code } t_1) (\text{code } t_2) \rightsquigarrow \text{reflect} (\text{app}^1 t_1 t_2) \\ (\text{ST-OP}^2) & (\text{code } t_1) \oplus^2 (\text{code } t_2) \rightsquigarrow \text{reflect} (t_1 \oplus^1 t_2) \end{array}$$

cross-stage — lift & run

$$\begin{array}{ll} (\text{ST-LAM}^\uparrow) & \text{lift } (\lambda x.t) \rightsquigarrow \lambda_c x.t[\text{code } x/x] \\ (\text{ST-RUN}) & \text{run } (\text{code } t) \rightsquigarrow t \end{array}$$

How do we know where to insert the let-binding?

We need to look at the context of reflect t :

$$\frac{x \text{ is fresh}}{P[E[\text{reflect } t]] \rightarrow P[\text{let}_c x = t \text{ in } E[\text{code } x]]}$$

- P reification context: the nearest boundary at which the let-binding can be safely inserted
- E pure context inside: call-by-value frames surrounding the reflect t , which t cannot depends on E
- Additionally, $M[t] \rightarrow M[t']$ if $t \rightsquigarrow t'$, where M is the top-level evaluation context where ordinary evaluation happens.

Dynamic Semantics and Let-insertion

`lift ( $\lambda x.x +^2$  (lift 1))`

P : reification context

E : pure context

M : evaluation context

t : redex / inserted code

Dynamic Semantics and Let-insertion

$\text{lift } (\lambda x. x +^2 (\text{lift } 1))$
 $\rightarrow \lambda_c x. ((\text{code } x) +^2 (\text{lift } 1))$

P : reification context

E : pure context

M : evaluation context

t : redex / inserted code

Dynamic Semantics and Let-insertion

$\text{lift } (\lambda x. x +^2 (\text{lift } 1))$
 $\rightarrow \lambda_c x. ((\text{code } x) +^2 (\text{lift } 1))$
 $\rightarrow \lambda_c x. ((\text{code } x) +^2 (\text{reflect } 1))$

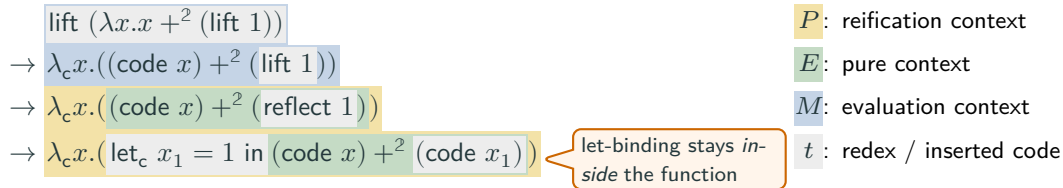
P : reification context

E : pure context

M : evaluation context

t : redex / inserted code

Dynamic Semantics and Let-insertion



Dynamic Semantics and Let-insertion

$\text{lift } (\lambda x. x +^2 (\text{lift } 1))$
 $\rightarrow \lambda_c x. ((\text{code } x) +^2 (\text{lift } 1))$
 $\rightarrow \lambda_c x. ((\text{code } x) +^2 (\text{reflect } 1))$
 $\rightarrow \lambda_c x. (\text{let}_c x_1 = 1 \text{ in } ((\text{code } x) +^2 (\text{code } x_1)))$

P : reification context

E : pure context

M : evaluation context

t : redex / inserted code

Dynamic Semantics and Let-insertion

$\text{lift } (\lambda x. x +^2 (\text{lift } 1))$
 $\rightarrow \lambda_c x. ((\text{code } x) +^2 (\text{lift } 1))$
 $\rightarrow \lambda_c x. ((\text{code } x) +^2 (\text{reflect } 1))$
 $\rightarrow \lambda_c x. (\text{let}_c x_1 = 1 \text{ in } (\text{code } x) +^2 (\text{code } x_1))$
 $\rightarrow \lambda_c x. (\text{let}_c x_1 = 1 \text{ in } \text{reflect } (x + x_1))$

P : reification context

E : pure context

M : evaluation context

t : redex / inserted code

Dynamic Semantics and Let-insertion

$\text{lift } (\lambda x.x +^2 (\text{lift } 1))$
 $\rightarrow \lambda_c x. ((\text{code } x) +^2 (\text{lift } 1))$
 $\rightarrow \lambda_c x. ((\text{code } x) +^2 (\text{reflect } 1))$
 $\rightarrow \lambda_c x. (\text{let}_c x_1 = 1 \text{ in } ((\text{code } x) +^2 (\text{code } x_1)))$
 $\rightarrow \lambda_c x. (\text{let}_c x_1 = 1 \text{ in } \text{reflect } (x + x_1))$
 $\rightarrow \lambda_c x. (\text{let}_c x_1 = 1 \text{ in } \text{let}_c x_2 = x + x_1 \text{ in } (\text{code } x_2))$

P : reification context

E : pure context

M : evaluation context

t : redex / inserted code

new let-binding cannot
cross the previous one

Dynamic Semantics and Let-insertion

$\text{lift } (\lambda x.x +^2 (\text{lift } 1))$
 $\rightarrow \lambda_c x. ((\text{code } x) +^2 (\text{lift } 1))$
 $\rightarrow \lambda_c x. ((\text{code } x) +^2 (\text{reflect } 1))$
 $\rightarrow \lambda_c x. (\text{let}_c x_1 = 1 \text{ in } ((\text{code } x) +^2 (\text{code } x_1)))$
 $\rightarrow \lambda_c x. (\text{let}_c x_1 = 1 \text{ in } \text{reflect } (x + x_1))$
 $\rightarrow \lambda_c x. (\text{let}_c x_1 = 1 \text{ in } \text{let}_c x_2 = x + x_1 \text{ in } (\text{code } x_2))$

P : reification context

E : pure context

M : evaluation context

t : redex / inserted code

Dynamic Semantics and Let-insertion

$\text{lift } (\lambda x.x +^2 (\text{lift } 1))$
 $\rightarrow \lambda_c x.((\text{code } x) +^2 (\text{lift } 1))$
 $\rightarrow \lambda_c x.((\text{code } x) +^2 (\text{reflect } 1))$
 $\rightarrow \lambda_c x.(\text{let}_c x_1 = 1 \text{ in } (\text{code } x) +^2 (\text{code } x_1))$
 $\rightarrow \lambda_c x.(\text{let}_c x_1 = 1 \text{ in } \text{reflect } (x + x_1))$
 $\rightarrow \lambda_c x.(\text{let}_c x_1 = 1 \text{ in } \text{let}_c x_2 = x + x_1 \text{ in } (\text{code } x_2))$
 $\rightarrow \lambda_c x.(\text{let}_c x_1 = 1 \text{ in } \text{code } (\text{let } x_2 = x + x_1 \text{ in } x_2))$

P : reification context

E : pure context

M : evaluation context

t : redex / inserted code

Dynamic Semantics and Let-insertion

$\text{lift } (\lambda x. x +^2 (\text{lift } 1))$
 $\rightarrow \lambda_c x. ((\text{code } x) +^2 (\text{lift } 1))$
 $\rightarrow \lambda_c x. ((\text{code } x) +^2 (\text{reflect } 1))$
 $\rightarrow \lambda_c x. (\text{let}_c x_1 = 1 \text{ in } ((\text{code } x) +^2 (\text{code } x_1)))$
 $\rightarrow \lambda_c x. (\text{let}_c x_1 = 1 \text{ in } \text{reflect } (x + x_1))$
 $\rightarrow \lambda_c x. (\text{let}_c x_1 = 1 \text{ in } \text{let}_c x_2 = x + x_1 \text{ in } (\text{code } x_2))$
 $\rightarrow \lambda_c x. (\text{let}_c x_1 = 1 \text{ in } \text{code } (\text{let } x_2 = x + x_1 \text{ in } x_2))$
 $\rightarrow \lambda_c x. (\text{code } (\text{let } x_1 = 1 \text{ in } \text{let } x_2 = x + x_1 \text{ in } x_2))$

P : reification context

E : pure context

M : evaluation context

t : redex / inserted code

Dynamic Semantics and Let-insertion

$\text{lift } (\lambda x. x +^2 (\text{lift } 1))$
→ $\lambda_c x. ((\text{code } x) +^2 (\text{lift } 1))$
→ $\lambda_c x. ((\text{code } x) +^2 (\text{reflect } 1))$
→ $\lambda_c x. (\text{let}_c x_1 = 1 \text{ in } ((\text{code } x) +^2 (\text{code } x_1)))$
→ $\lambda_c x. (\text{let}_c x_1 = 1 \text{ in } \text{reflect } (x + x_1))$
→ $\lambda_c x. (\text{let}_c x_1 = 1 \text{ in } \text{let}_c x_2 = x + x_1 \text{ in } (\text{code } x_2))$
→ $\lambda_c x. (\text{let}_c x_1 = 1 \text{ in } \text{code } (\text{let } x_2 = x + x_1 \text{ in } x_2))$
→ $\lambda_c x. (\text{code } (\text{let } x_1 = 1 \text{ in } \text{let } x_2 = x + x_1 \text{ in } x_2))$
→ $\text{reflect } (\lambda x. \text{let } x_1 = 1 \text{ in } \text{let } x_2 = x + x_1 \text{ in } x_2)$

P : reification context

E : pure context

M : evaluation context

t : redex / inserted code

Dynamic Semantics and Let-insertion

lift ($\lambda x.x +^2$ (lift 1))
→ $\lambda_c x.((\text{code } x) +^2$ (lift 1))
→ $\lambda_c x.((\text{code } x) +^2$ (reflect 1))
→ $\lambda_c x.(\text{let}_c x_1 = 1$ in $(\text{code } x) +^2$ (code x_1))
→ $\lambda_c x.(\text{let}_c x_1 = 1$ in reflect $(x + x_1)$)
→ $\lambda_c x.(\text{let}_c x_1 = 1$ in $\text{let}_c x_2 = x + x_1$ in (code x_2))
→ $\lambda_c x.(\text{let}_c x_1 = 1$ in code (let $x_2 = x + x_1$ in x_2))
→ $\lambda_c x.(\text{code } (\text{let } x_1 = 1$ in let $x_2 = x + x_1$ in x_2))
→ reflect ($\lambda x.\text{let } x_1 = 1$ in let $x_2 = x + x_1$ in x_2)
→ $\text{let}_c x_3 = (\lambda x.\text{let } x_1 = 1$ in let $x_2 = x + x_1$ in x_2) in (code x_3)

P : reification context

E : pure context

M : evaluation context

t : redex / inserted code

Dynamic Semantics and Let-insertion

lift ($\lambda x.x +^2$ (lift 1))
→ $\lambda_c x.((\text{code } x) +^2 (\text{lift } 1))$
→ $\lambda_c x.((\text{code } x) +^2 (\text{reflect } 1))$
→ $\lambda_c x.(\text{let}_c x_1 = 1 \text{ in } (\text{code } x) +^2 (\text{code } x_1))$
→ $\lambda_c x.(\text{let}_c x_1 = 1 \text{ in } \text{reflect } (x + x_1))$
→ $\lambda_c x.(\text{let}_c x_1 = 1 \text{ in } \text{let}_c x_2 = x + x_1 \text{ in } (\text{code } x_2))$
→ $\lambda_c x.(\text{let}_c x_1 = 1 \text{ in } \text{code } (\text{let } x_2 = x + x_1 \text{ in } x_2))$
→ $\lambda_c x.(\text{code } (\text{let } x_1 = 1 \text{ in } \text{let } x_2 = x + x_1 \text{ in } x_2))$
→ $\text{reflect } (\lambda x.\text{let } x_1 = 1 \text{ in } \text{let } x_2 = x + x_1 \text{ in } x_2)$
→ $\text{let}_c x_3 = (\lambda x.\text{let } x_1 = 1 \text{ in } \text{let } x_2 = x + x_1 \text{ in } x_2) \text{ in } (\text{code } x_3)$

P : reification context

E : pure context

M : evaluation context

t : redex / inserted code

Dynamic Semantics and Let-insertion

lift ($\lambda x.x +^2 (\text{lift } 1)$)

→ $\lambda_c x.((\text{code } x) +^2 (\text{lift } 1))$

→ $\lambda_c x.((\text{code } x) +^2 (\text{reflect } 1))$

→ $\lambda_c x.(\text{let}_c x_1 = 1 \text{ in } (\text{code } x) +^2 (\text{code } x_1))$

→ $\lambda_c x.(\text{let}_c x_1 = 1 \text{ in } \text{reflect } (x + x_1))$

→ $\lambda_c x.(\text{let}_c x_1 = 1 \text{ in } \text{let}_c x_2 = x + x_1 \text{ in } (\text{code } x_2))$

→ $\lambda_c x.(\text{let}_c x_1 = 1 \text{ in } \text{code } (\text{let } x_2 = x + x_1 \text{ in } x_2))$

→ $\lambda_c x.(\text{code } (\text{let } x_1 = 1 \text{ in } \text{let } x_2 = x + x_1 \text{ in } x_2))$

→ $\text{reflect } (\lambda x.\text{let } x_1 = 1 \text{ in } \text{let } x_2 = x + x_1 \text{ in } x_2)$

→ $\text{let}_c x_3 = (\lambda x.\text{let } x_1 = 1 \text{ in } \text{let } x_2 = x + x_1 \text{ in } x_2) \text{ in } (\text{code } x_3)$

→ $\text{code } (\text{let } x_3 = (\lambda x.\text{let } x_1 = 1 \text{ in } \text{let } x_2 = x + x_1 \text{ in } x_2) \text{ in } x_3)$

Legend:

- P : reification context
- E : pure context
- M : evaluation context
- t : redex / inserted code

ANF code

Static Semantics: Type and Effect System

$$\begin{array}{lcl} \text{Effects } \epsilon & ::= & \perp \mid \top \\ \text{Types } \tau & ::= & b \mid \text{ref } b \mid \tau_1 \rightarrow^\epsilon \tau_2 \\ & & \mid \text{rep } \tau \mid \text{frag } \tau \end{array}$$

- $\Gamma \vdash^S t : \tau \mid \epsilon$, where effect ϵ tracks whether let-insertion may happen.
- Distinguish two kinds of code types: complete code values $\text{rep } \tau$ and fragment code $\text{frag } \tau$.

rep τ **code blocks**

- Can be effectful in the second stage
- Cannot be bound to a variable in the meta-program
- E.g., a sequence of residual let-bindings

$$\frac{\Gamma \vdash^1 t_1 : \text{frag } \mathbb{N} \mid \epsilon_1 \quad \Gamma \vdash t_2 : \text{rep } \tau \mid \epsilon_2 \quad \Gamma \vdash t_3 : \text{rep } \tau \mid \epsilon_3}{\Gamma \vdash^1 \text{ifz}^2 t_1 t_2 t_3 : \text{frag } \tau \mid \top}$$

Static Semantics: Type and Effect System

rep τ **code blocks**

- Can be effectful in the second stage
- Cannot be bound to a variable in the meta-program, thus cannot be duplicated/discarded
- E.g., a sequence of residual let-bindings

$$\frac{\Gamma \vdash^1 t_1 : \text{frag } \mathbb{N} \mid \epsilon_1 \quad \Gamma \vdash t_2 : \text{rep } \tau \mid \epsilon_2 \quad \Gamma \vdash t_3 : \text{rep } \tau \mid \epsilon_3}{\Gamma \vdash^1 \text{ifz}^2 t_1 t_2 t_3 : \text{frag } \tau \mid \top}$$

frag τ **code fragments**

- To be let-inserted or a variable referring an let-inserted code
- Can be used unrestrictedly
- E.g., *reflect* an application, which will be let-bound during reduction

$$\frac{\Gamma \vdash^2 t : \tau \mid \perp}{\Gamma \vdash^1 \text{reflect } t : \text{frag } \tau \mid \top}$$

Erasure

Erasure $|\cdot|$ removes staging annotations for types and terms:

$$|\tau_1 \rightarrow^\epsilon \tau_2| = |\tau_1| \rightarrow^\perp |\tau_2|$$

$$|\text{rep } \tau| = |\tau|$$

$$|\text{frag } \tau| = |\tau|$$

$$|\Gamma, x^s : \tau| = |\Gamma|, x^2 : |\tau|$$

$$|\text{lift } t| = |t|$$

$$|\text{app}^s t_1 t_2| = \text{app}^\mathbb{1} |t_1| |t_2|$$

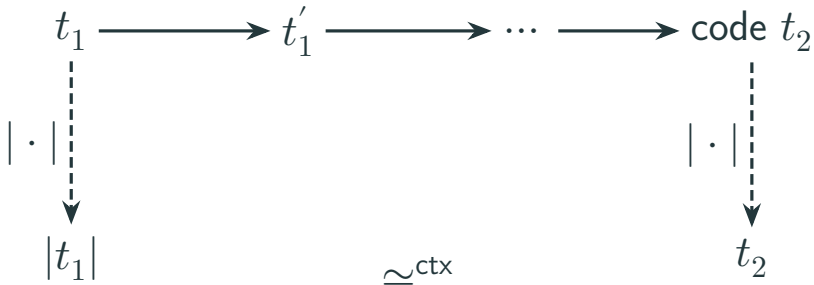
$$|\text{let}_c x = t_1 \text{ in } t_2| = \text{let } x = |t_1| \text{ in } |t_2|$$

$$|\lambda_c x. t| = \lambda x. |t|$$

$$|\text{reflect } t| = |t|$$

$$\vdots$$

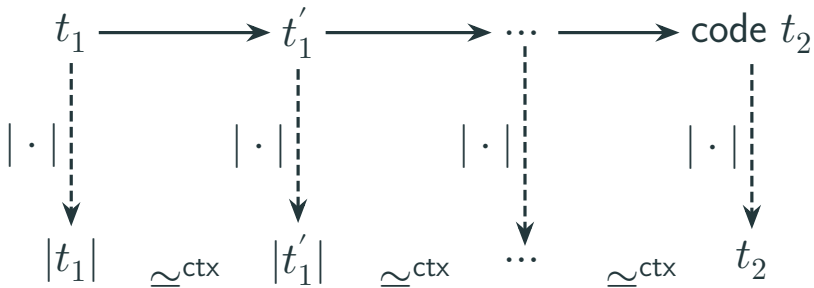
Metatheory: Semantics Preservation



Semantics Preservation Theorem ($\lambda_{|2|}^{\text{ref}}$)

If $\emptyset \vdash t_1 : \text{rep } \tau \mid \epsilon$, and $\langle \emptyset, t_1 \rangle \rightarrow^* \langle \emptyset, \text{code } t_2 \rangle$, then $\emptyset \models |t_1| \simeq^{\text{ctx}} t_2 : \tau$.

Metatheory: Semantics Preservation



- Proof technique: Kripke-style binary logical relations
- In the two-stage setting, we only need to relate staging-free code, thus standard logical relation techniques suffice.

- Definitional interpreters that implement the let-inserting staging calculus (ICFP '26)
- Alternative quotation semantics from the use-mention perspective (work in progress)

Implementing Let-Insertion

Let-insertion is fundamentally about control, and there are many different ways to implement it:

- **Mutable state** [Sumii and Kobayashi, '01; Rompf, '16; Amin and Rompf, '18]
- 1-CPS [Flanagan, '93]
- **2-CPS** [Danvy, '90]
- control operators (shift/reset) [Danvy '96; Kameyama, '07; genlet in BER MetaOCaml]
- abstract machine [Biernacka '05]

Let it be optimized: building multi-stage evaluators with let-insertion and optimizations in small pieces (ICFP 26 Functional Pearl, Wei et al.)

Correspondence between the 2-CPS definitional interpreter and small-step reduction semantics:

```
def reflect(e, k, γ) =  
  val x = fresh()  
  k(x, body =>  
    γ(VCode(ELet(x, e, body)))
```

$$\frac{x \text{ is fresh}}{P[E[\text{reflect } t]] \rightarrow P[\text{let}_c x = t \text{ in } E[\text{code } x]]}$$

Intermediate artifact: CEK-like abstract machine semantics (ICFP '26).

Let-insertion with Mutable State

An environment-passing evaluator for two-stage let-inserting evaluation:

```
def eval(e: Expr, ρ: Env): Value = e match
  case EApp2(e1, e2) =>
    val VCode(x1) = eval(e1, ρ)
    val VCode(x2) = eval(e2, ρ)
    reflect(EApp1(x1, x2))
  case ECond2(e1, e2, e3) =>
    val VCode(x1) = eval(e1, ρ)
    val x2 = reify { eval(e2, ρ) }
    val x3 = reify { eval(e3, ρ) }
    reflect(ECond1(x1, x2, x3))
  ...
```

Let-insertion with Mutable State

The interpreter operates with a mutable list to construct let-bindings for the next stage (Rompf '16):

```
var stBlock: List[(String, Expr)]  
def reflect(e) =  
    stBlock += (x, e) // record x -> e in stBlock for some fresh x  
    VCode(x) // return object variable x  
def reify(f) = ... // delimit a scope and assemble its bindings
```

Example:

```
(fun x => x +2 x) (f@2())
```

Trace of the internal state:

```
reflect(f()) // stBlock = [x1 -> f()]  
reflect(x1 + x1) // stBlock = [x1 -> f(), x2 -> x1 + x1]  
reify(...) // let x1 = f() in let x2 = x1 + x1 in x2
```

Use-Mention Distinction and Quotation

A perspective from the **use-mention** distinction:

Snow is white.

We **use** the word “snow” to talk about snow.

“Snow” has four letters.

We **mention** the word “snow” to talk about the word itself.

Use-Mention Distinction and Quotation

A perspective from the **use-mention** distinction:

Snow is white.

We **use** the word “snow” to talk about snow.

“Snow” has four letters.

We **mention** the word “snow” to talk about the word itself.

More often, we mix use and mention when using quotation:

Quine says quotation “has a certain anomalous feature”.

Use-Mention Distinction and Quotation

A perspective from the **use-mention** distinction:

Snow is white.

We **use** the word “snow” to talk about snow.

“Snow” has four letters.

We **mention** the word “snow” to talk about the word itself.

More often, we mix use and mention when using quotation:

Quine says quotation “has a certain anomalous feature”.

**In programming languages, quotation solely serves the mentioning purpose.
What would be the analogue of use-quotation in staging?**

Two Different Quotations

We implemented a Scala 3 compiler plugin supporting:

- `'{ ... }` for ordinary mention-quotation
- `!'{ ... }` for use-quotation, which ensures the code generated preserves the semantics of the annotation-erased program

Example:

```
val c = !'{ println("hello"); } // use-quotation  
'42                             // mention-quotation
```

Generates code:

```
val x1 = println("hello")  
42
```

More Examples

// mixing use- and mention-quotation

```
val c = !'{ println("hello"); 42 }  
'{ $c + $c }
```

Generates code:

```
val x1 = println("hello")  
val x2 = 42  
x2 + x2
```

// both are use-quotation

```
val c = !'{ println("hello"); 42 }  
!'{ $c + ($c * $c) }
```

Generates code:

```
val x1 = println("hello")  
val x2 = 42  
val x3 = x2 * x2  
val x4 = x2 + x3  
x4
```

How

- Translating terms in `!'{ ... }` into combinators that will be executed at runtime
- These combinators are essentially the tagless-final style interpreter (Carette et al., 07) that operates with a mutable state to record the let-bindings for the next stage

```
def mkAdd(lhs: Expr[Any], rhs: Expr[Any]): Expr[Any] =  
  reflect('{ ${lhs.asExprOf[Int]} + ${rhs.asExprOf[Int]} }')
```

```
def reflect[T](rhs: Expr[T]): Expr[T] =  
  val term = rhs.asTerm  
  val symbol = Symbol.newVal(...)   
  bindings += ValDef(symbol, term)  
  Ref(symbol).asExpr.asInstanceOf[Expr[T]]
```

- Work-in-progress Scala 3 compiler plugin:
<https://github.com/instar-lang/use-quotation/>

Semantics-Preserving Quotation

Now, erasing staging annotations also preserves the semantics:

```
def impPower(x: Expr[Double], n: Int,  
            res: Expr[RefCell[Double]]): Expr[Double] =  
  if n == 0 then '{ ${res}.x }  
  else '{  
    ${res}.x = $x * ${res}.x;  
    ${impPower(x, n - 1, res)}  
  }
```

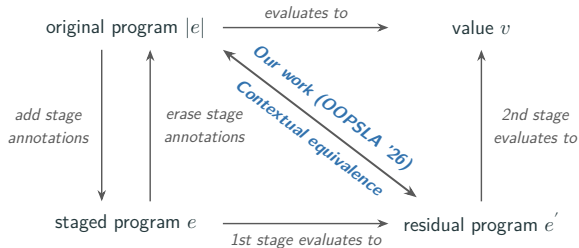
```
def powerCode(n: Int): Expr[Double => Double] =  
  val res = !'{ RefCell[Double](1.0) }  
  '{ (x: Double) => ${impPower('x, n, res)} }
```

Towards Expressive Semantics-Preserving Multi-Stage Programming

“When do we know that erasing annotations preserves semantics?”

(Inoue & Taha, 2012, 2016)

In $\lambda_{|2|}/\lambda_{|2|}^{\text{ref}}$, erasing annotations **always** preserves semantics!



- Mechanization & WIP implementations:
<https://github.com/instar-lang>

Thank you!

- When Do Staging Annotations Preserve Semantics? (OOPSLA '26, Tan & Wei)
- Let It Be Optimized (ICFP 26, Wei et al.)