

When do staging annotations preserve semantics?

*Mechanizing typed semantics-preserving
multi-stage programming with let-insertion*

Jun Tan, **Guannan Wei**

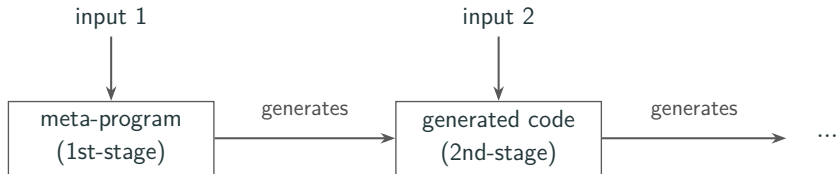
OOPSLA 2026



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?

But 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)>) >
```

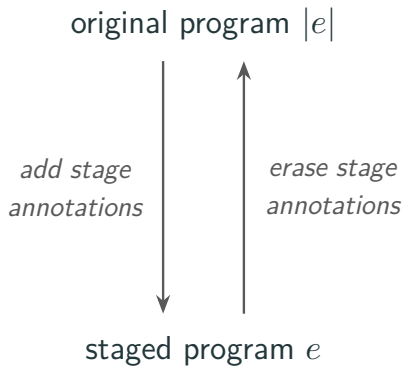
erase stage
annotations



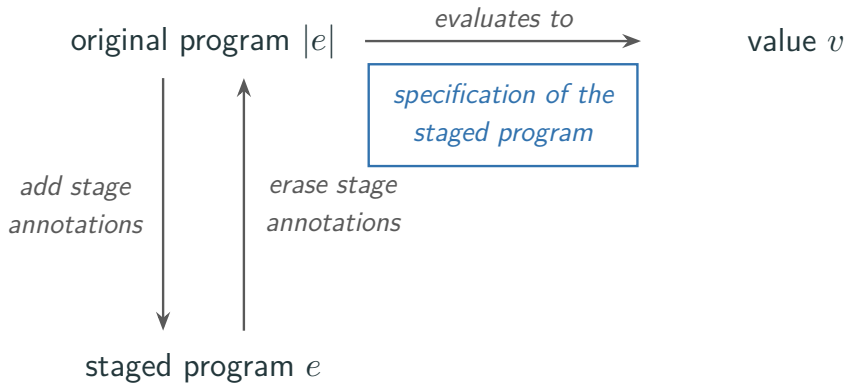
```
let rec power (n: int)
  (x: int)
  (res: int ref): int =
  if n = 0 then !res
  else (res := x * !res;
        power (n - 1) x res)
  fun x -> power 3 x (ref 1)
```

What goes wrong and what should we expect from staging?

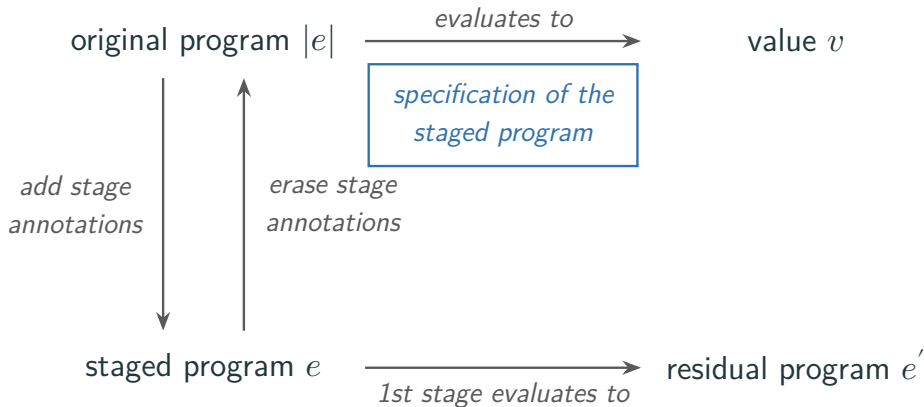
Staging and Evaluation



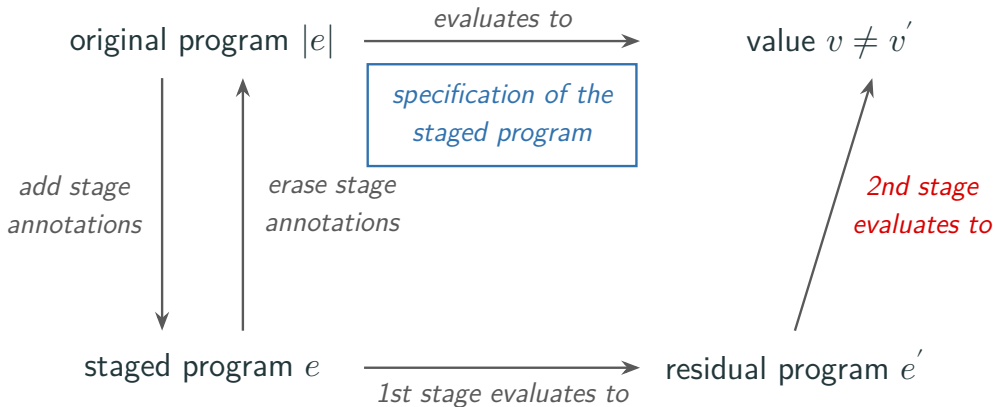
Staging and Evaluation



Staging and Evaluation

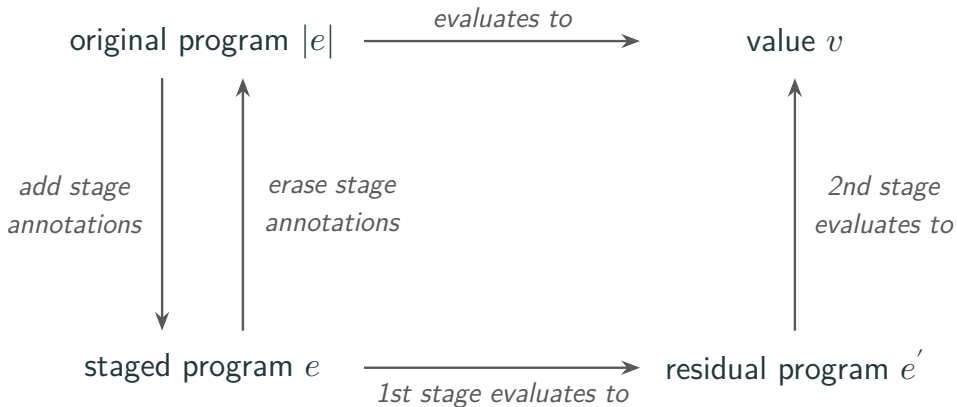


Staging and Evaluation



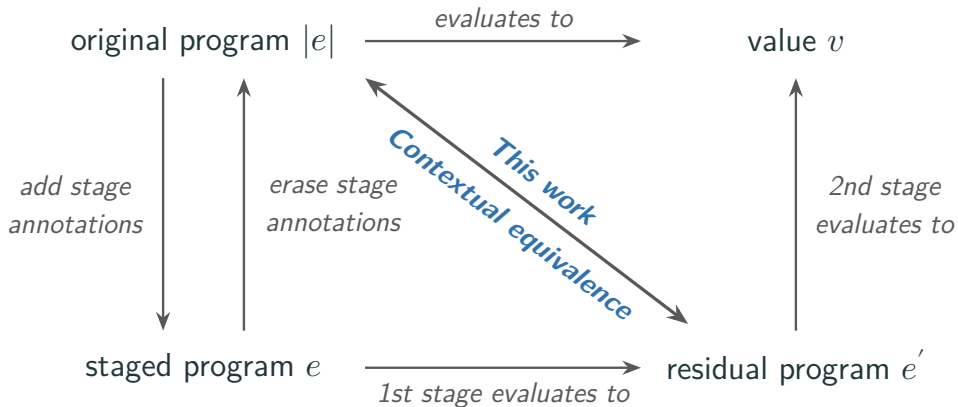
MetaML-style quotation/splicing manipulates syntax, but can also change program behavior and effects in call-by-value.

Semantics-Preserving Staging



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

This Work: Designing Semantics-Preserving Staged Languages



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**.

Terms carry a marker $\mathbb{1}$ or $\mathbb{2}$ indicating the stage of evaluation. For example, 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)))
```

Dynamic Semantics and Let-Insertion

- 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  
    r := x * !r;  
    r := x * !r;  
    r := x * !r;  
  !r>
```

Dynamic Semantics and Let-Insertion

- 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  
    r := x * !r;  
    r := x * !r;  
    r := x * !r;  
  !r>
```

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

$$(\text{code } f)@^2(\text{code } 2) \rightsquigarrow \text{reflect } f(2)$$

Dynamic Semantics and Let-Insertion

- 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  
    r := x * !r;  
    r := x * !r;  
    r := x * !r;  
  !r>
```

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

$$(\text{code } f)@^2(\text{code } 2) \rightsquigarrow \text{reflect } f(2)$$

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

We need to look at the context of reflect t :

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

- 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

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
- E.g., a sequence of residual let-bindings

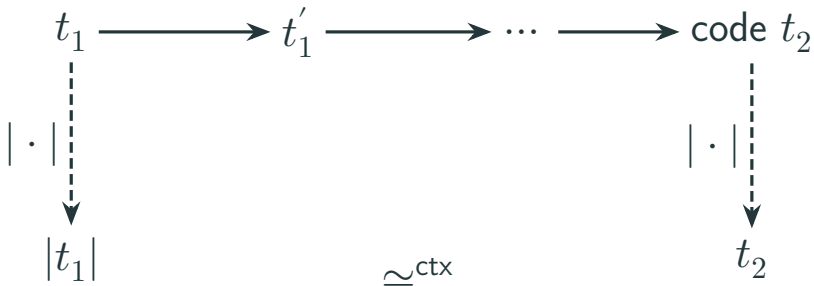
$$\frac{\Gamma \vdash^1 t_1 : \text{frag } \mathbb{N} \mid \epsilon_1 \quad \Gamma \vdash t_2 : \text{rep } \tau \mid \epsilon_2 \quad \Gamma \vdash t_3 : \text{rep } \tau \mid \epsilon_3}{\Gamma \vdash^1 \text{ifz}^2 t_1 t_2 t_3 : \text{frag } \tau \mid \top}$$

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

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

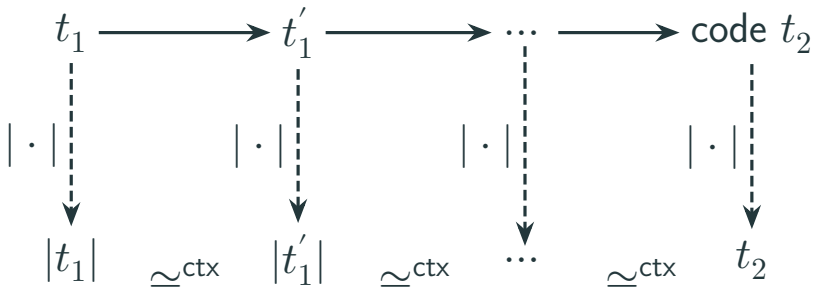
Metatheory: Semantics Preservation



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



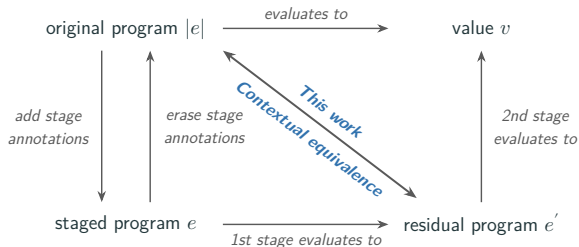
- Proof technique: Kripke-style binary logical relations
- In the two-stage setting, we only need to relate staging-free code, thus standard logical relation techniques suffice.

Conclusion: Semantics-Preserving Multi-Stage Programming

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

(Inoue & Taha, 2012, 2016)

We answer this question from the language design perspective: in $\lambda_{|2|}/\lambda_{|2|}^{\text{ref}}$, erasing annotations **always** preserves semantics!



- Mechanization & WIP prototype: <https://github.com/instar-lang>
- Future work: first-stage store effects, higher-order stores, and arbitrary number of stages, ...

Thank you!