



When Do Staging Annotations Preserve Semantics? Mechanizing Typed Semantics-Preserving Multi-stage Programming with Let-Insertion

JUN TAN, Independent, China

GUANNAN WEI[✉], Tufts University, USA

Multi-stage programming with quotations has long provided a powerful way to generate and manipulate code. By treating code as data, programmers can write multi-stage programs in which earlier stages produce specialized code from inputs available at generation time. Modern typed multi-stage languages (e.g., MetaML, MetaOCaml, Template Haskell, and Scala 3) adopt quotation/splicing constructs while enforcing the well-typedness of generated code. However, manipulating code fragments syntactically can subtly change evaluation order, leading to semantic discrepancies between a staged program and its unstaged counterpart, which is intended to serve as a reference implementation in many cases. The inconsistency complicates reasoning about correctness, and prevents staged code from being a drop-in replacement for its unstaged counterpart.

In this paper, we study the design of multi-stage languages with semantics preservation guarantees. We develop two statically typed two-stage calculi, $\lambda_{[2]}$ and $\lambda_{[2]}^{\text{ref}}$, the latter supporting mutable references in the second stage. Their dynamic semantics model automatic let-insertion, tracked as a control effect in a light-weight type-and-effect system, enabling type-safe and semantics-preserving manipulation of effectful code fragments. We develop binary logical relations to prove strong semantics-preservation theorems: if a well-typed two-stage program t_1 evaluates to a value code t_2 , then t_2 is contextually equivalent to the stage-erasure of t_1 . Our calculi and their mechanized metatheory provide a simple and definitive answer to the question posed by Inoue and Taha of when staging annotations preserve semantics, and lay a foundation for future work on semantics-preserving multi-stage programming.

CCS Concepts: • **Software and its engineering** → **Source code generation**; **Semantics**; **Functional languages**; • **Theory of computation** → **Type theory**.

Additional Key Words and Phrases: staging, let-insertion, partial evaluation, type systems, mechanization

ACM Reference Format:

Jun Tan and Guannan Wei. 2026. When Do Staging Annotations Preserve Semantics? Mechanizing Typed Semantics-Preserving Multi-stage Programming with Let-Insertion. *Proc. ACM Program. Lang.* 10, OOPSLA2, Article 345 (October 2026), 28 pages. <https://doi.org/10.1145/3839477>

1 Introduction

Multi-stage programming (MSP) with quoting and splicing has long provided a powerful means for code generation and manipulation, from the early days of Lisp [7] to MetaML [52, 53] and its descendants. By constructing the meta-program with quotations and splices, MSP allows programs to execute in multiple stages, where earlier stages generate code specialized to the data available at those stages. Staging annotations provide a natural and programmer-controlled form of partial evaluation [29], as well as an effective mechanism for domain-specific program optimization.

Authors' Contact Information: Jun Tan, Independent, Shanghai, China, tjapzjj@gmail.com; Guannan Wei, Tufts University, Medford, USA, guannan.wei@tufts.edu.



This work is licensed under a Creative Commons Attribution 4.0 International License.

© 2026 Copyright held by the owner/author(s).

ACM 2475-1421/2026/10-ART345

<https://doi.org/10.1145/3839477>

With the syntactic form of annotations, MetaML [52, 53] further ensures hygiene and type safety of generated code by type systems, giving rise to what is now generally referred to as multi-stage programming. Since the introduction of MetaML, typed MSP with syntactic annotations has grown into a large family of research calculi (e.g., [64, 68, 70]) and has been adopted in various industrial languages, including MetaOCaml [33–35], F# [51], (Typed) Template Haskell [48, 67], and more recently Scala 3 [49, 50].

Semantics-Breaking Staging Annotations. Despite the expressive power and popularity of MetaML-style syntactic MSP, multi-stage programs are notoriously difficult to reason about [27, 28]. Unlike the name “annotations” might suggest, annotations are not just syntactic markers that can be freely added or removed without changing the semantics of the program. They can rearrange the evaluation order within the same stage, which can lead to inconsistent behaviors between the staged program and its original unstaged counterpart (obtained by erasing staging annotations) in the presence of computational effects, such as divergence and mutable states.

We use the syntax from MetaOCaml [35] to illustrate a few examples. Brackets `.<e>.` delay the evaluation of the enclosed expression e as a code value, and an escape `.~e` splices the value of e into a surrounding code fragment. The type τ code denotes code fragments that, when run, produces a value of type τ . Now let us consider the following *first-stage* function k (adapted from the SKI combinators), which takes two *second-stage* integers and returns the first one:

```
let k (x : int code) (y : int code) : int code = x in
let rec loop () = loop () in
k .<42> . .<loop ()> .
```

Here, when applying k , the first argument is a code fragment of integer `.<42>.`, and the second is a code fragment of a non-terminating computation `.<loop ()>.`. The first-stage evaluation terminates and produces a *code* `.<42>.`, which can be further evaluated to *integer* 42 when run. Since k simply returns the first argument, quoting not only delays the evaluation of `loop ()`, but completely prevents it from being evaluated *at all*.

However, if we compare the staged program with its unstaged counterpart, we can see that they behave differently. The unstaged version obtained by erasing staging annotations (brackets, escapes, and τ code types) is the following, which diverges under call-by-value (CBV):

```
let k (x : int) (y : int) : int = x in
let rec loop () = loop () in
k 42 (loop ())
```

Moreover, splicing code fragments at will can lead to code duplication, and with it, duplicated effects. Consider the following *twice* function that splices the argument x twice into the body, effectively adding the argument to itself:

```
(* erased: let twice x = x + x *)
let twice (x : int code) : int code = .<~x + ~x> . in ...
```

Now invoking *twice* with a code fragment inducing the second-stage stdout effect:

```
twice .<(print_string "Hello"; 42)> .
(* generates code by copying the argument twice: *)
.<(print_string "Hello"; 42) + (print_string "Hello"; 42)> .
```

The staged program generates code that prints twice. In contrast, the unstaged (or annotation-erased) program evaluates the argument only once *before* the function call and thus prints “Hello” only once, consistent with standard CBV evaluation.

Why it Matters? These examples show that, because syntactic staging annotations permit evaluation to be rearranged, they can arbitrarily discard, reorder, or duplicate effects. Although such a flexibility can be useful for program manipulation in some cases, we lose the ability to relate the

staged program with the unstaged program, thus weakening semantic predictability and making staged programs much harder to reason about.

This limitation particularly hinders performance-oriented uses of staging, such as Futamura projections [23] and domain-specific staging [4, 54, 59, 61, 62], whose *goal is precisely to produce faster program without changing the observable behavior of the original unstaged program*. If staging annotations do not preserve semantics, the unstaged program cannot serve as a reference specification from which one derives a faster, equivalent version by merely inserting staging annotations. Therefore, the generated program cannot be used as drop-in replacement for the original.

The key property connecting a staged program and its unstaged counterpart is known as *erasure soundness* [27, 28, 71] or semantics preservation. Although it was recognized as an important property [27, 28], MetaML-style CBV staging unfortunately does not enjoy this property in general. In fact, staging in CBN calculi is better behaved, since the semantics of quotation remains aligned with the underlying CBN semantics [27, 28, 53]. However, adding staging annotations to a CBV language breaks the host language’s equational theory [27, 28]. As a result, the standard reasoning principles for CBV programs can no longer be assumed to apply to staged programs.

Semantic preservation is also the cornerstone of automated partial evaluation [29]. As its successor, MSP gives programmers explicit control over binding-times but abandons guarantees that the generated code preserves the original program’s semantics.

“When Do We Know That Erasing Annotations Preserves Semantics?” Nevertheless, erasing staging annotations does preserve semantics for certain classes of staged programs. This raises a natural question by Inoue and Taha [27, 28]: “*when do we know that erasing annotations preserves semantics*”? They partially answered this question by developing applicative bisimulation techniques for proving the equivalence of *specific* staged programs and their unstaged counterparts. However, these techniques require substantial manual proof effort and remain program-specific: they do not yield a general erasure-soundness guarantee for the language as a whole. This leaves a fundamental open question: can we obtain stronger semantic guarantees other than well-typedness of generated code for a call-by-value staging language?

To shed light on this question, let us turn to practical staging languages and frameworks. They often adopt *let-insertion* [22] as a remedy, which has long been used in partial evaluation for effectful programs [15, 16, 20, 31, 37]. The idea is to introduce let-binding at the right places to force the evaluation and intended effects of a code fragment to happen, and then refer to the let-bound variable in the rest of the code. In this way, let-insertion preserves both effects *and* their relative evaluation order (see Section 2 for illustrative examples).

One may then ask whether an *automatic let-insertion* staging semantics makes staged programs easier to reason about. More specifically, does it guarantee semantics preservation of staging? After all, let-insertion is widely used in practical staging systems precisely because it is believed to help preserve the intended semantics. Answering it, however, requires a formal account relating three artifacts: the staged program, the code it generates, and its unstaged counterpart. Despite substantial research in MSP on typing, hygiene, and intensional analysis, this relational property has received far less attention. As a result, whether let-insertion provides stronger reasoning principles or semantics preservation remains largely folklore.

This Work. To gain a rigorous understanding of the semantic guarantees provided by let-insertion, this paper develops novel typed staging calculi featuring *automatic let-insertion* and investigates their metatheory. We summarize the main results below; Section 2.2 provides an overview for key ideas in our calculi, while the remainder of the paper develops the formal theory.

Following Inoue and Taha [27, 28], we begin with a pure setting without side effects, while still allowing divergence. In this setting, we introduce $\lambda_{|2|}$, a call-by-value, statically typed, two-stage

λ -calculus with general recursion and automatic let-insertion (Section 3). Its reduction semantics follows Amin and Rompf [3]’s untyped $\lambda_{\downarrow}$ -calculus, which already captures let-insertion operationally (Section 3.2). We develop a novel type-and-effect system for automatic let-insertion (Section 3.3), where the key idea is to treat code generation as a control effect in the first stage and to distinguish complete code values from intermediate code fragments.

We prove syntactic type soundness (Section 3.4), define the erasure operation, and show that erasure preserves syntactic typing (Section 4), which is a prerequisite for semantics preservation. We further prove the semantics-preservation theorem for staging and erasure (Section 5). The central technical tool is a binary Kripke logical relation for the two-stage calculus: we show that if a well-typed two-stage program t_1 evaluates to code t_2 , then t_2 is contextually equivalent to the erased term $|t_1|$, where $|\cdot|$ removes all staging annotations. Thus, in $\lambda_{|2|}$, erasure is sound for *all* well-typed programs. This directly answers the question posed by Inoue and Taha [27, 28]: *erasing annotations in $\lambda_{|2|}$ always preserves semantics*.

Next, we introduce ML-style mutable references to study the interaction between let-insertion and store effects (Section 6). Unrestricted references reveal an important boundary of erasure soundness: staging may evaluate under second-stage binders or conditionals, producing first-stage store effects that erasure does not preserve (Section 6.1). We then recover erasure soundness for a broad class of programs by restricting references to the second stage. We formalize this approach in $\lambda_{|2|}^{\text{ref}}$, a modest extension of $\lambda_{|2|}$ (Section 6.2), and prove both type soundness and erasure soundness (Section 6.3). This result is practically important because generating and manipulating effectful code is a central use case for staging.

At the same time, our study proposes a challenging open question: how to support more expressive first-stage store effects while retaining erasure soundness. We discuss the obstacles and possible directions (Section 7), and hope this work will serve as a foundation for future progress on semantics-preserving staging. Finally, all calculi and their metatheory results are mechanized (Section 8) in Lean 4 [19].

Contributions and Organization. The main contributions of this paper are threefold:

- To the best of our knowledge, no prior work has developed a typed, semantics-preserving staging calculus with a mechanized erasure soundness proof. We close this gap and answer the open question raised by Inoue and Taha [27, 28] by developing the first typed CBV multi-stage calculi $\lambda_{|2|}$ and $\lambda_{|2|}^{\text{ref}}$, with mechanized erasure soundness proofs. Our contributions allow programmers to introduce staging annotations for performance without worrying that they will silently alter the program’s meaning.
- We develop a novel type-and-effect system for automatic let-insertion. The key idea is to treat code generation as a control effect, while distinguishing complete code values from intermediate code fragments. The $\lambda_{|2|}^{\text{ref}}$ -calculus presents a novel design combining first-stage control effects and second-stage store effects, ensuring sound manipulation of effectful code.
- We apply binary Kripke logical relations to relate staged and unstaged programs and to prove semantics preservation of staging. To the best of our knowledge, this is one of the first uses of logical relations to reason about staging semantics and relate generated code to its erased counterpart, and we expect the technique to be of independent interest as a reusable foundation for future work.

The rest of the paper is organized as follows.

- In Section 2, we motivate the semantics preservation issue by examples that manipulate effectful code, discuss existing remedies, and introduce the key ideas of $\lambda_{|2|}/\lambda_{|2|}^{\text{ref}}$.

- In Section 3, we present the syntax, reduction semantics, and type-and-effect system of $\lambda_{|2|}$. In Section 3.4, we prove the syntactic type soundness of $\lambda_{|2|}$ via progress and preservation.
- In Section 4, we introduce the erasure operation applied to terms, types, and environments, and prove the syntactic erasure soundness theorem. In Section 5, we introduce the notion of contextual equivalence, develop a step-indexed Kripke logical relation for the erased subset language of $\lambda_{|2|}$, and prove the semantic soundness of erasure via the logical relation.
- In Section 6, we discuss the challenges in supporting general store effects while retaining erasure soundness. Based on the discussion, we extend $\lambda_{|2|}$ to $\lambda_{|2|}^{\text{ref}}$ with second-stage only mutable references, and prove syntactic type soundness and erasure soundness for it.
- In Section 7, we discuss possible solutions to support more expressive first-stage effects while retaining erasure soundness. We also discuss other possible extensions, such as supporting more stages and higher-order store.
- In Section 8, we discuss the mechanization in Lean 4. In Section 9, we discuss related work.

2 Motivation and Overview

In this section, we present a motivating example to illustrate the issue of semantics preservation in staging and existing solutions, and then explain the key ideas and main results of our work.

2.1 Motivating Example: Generative Imperative Power

The canonical motivating example for MSP is specializing the power function — *i.e.*, computing x^n when the exponent n is known at code generation time. The power function can be implemented in a purely functional style, but to discuss the subtle interaction between staging and effects, we consider an imperative version that uses a mutable reference `res` to accumulate the result:

```
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)
```

To call `power'`, the client code allocates a reference initialized to 1 and passes it as an argument:

```
let power n = fun x -> power' n x (ref 1)
```

Staged Imperative Power via Quasi-Quotation. Now, given a known value for n , a careless programmer may attempt to specialize `power'` by simply adding stage annotations. In this example, we use MetaML-style syntax for quotation and splicing. Since x and `res` are only known at runtime, they are marked as code types, while n is a plain `int`; the return type is also `int` code.

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

The body has been adapted using proper quotations and splices. The two branches of the `if` expression are both quoted code fragments. When referring to x and `res` immediately inside quotations, we splice (e.g., `<~x>`) them as they are already code values. To drive the specialization, we use `run` to execute the generated code at runtime. In the code argument for `run`, the function provides parameter x that can be used as a placeholder for the argument to be passed at runtime:

```
let power n = run <.<fun x -> <~(power' n <~x>). <~(ref 1)>.>.>.
```

We also add quotations around the allocation, since the result should be produced in the second stage, and we are in a splicing context to call `power'`.

However, despite the term “annotations” suggesting that they are orthogonal to program behavior, staging here unrolls the recursion and generates code that duplicates all allocations:

```
fun x -> (* example: specialized for n = 2 *)
  (ref 1) := x * !(ref 1);
```

```
(ref 1) := x * !(ref 1);
!(ref 1)
```

The generated code degenerates to a constant function that always returns 1, which is clearly not the intended power function! The problem here is that the quoted allocation `res` binds in the first stage, and each occurrence of `res` simply substitutes it into the larger template, causing duplication. When manipulating effectful code during staging, such duplication may violate the intended semantics of the generated code.

To generate the intended code, the programmer must manually insert a quoted let-binding to name the allocation, thus allocating only once:

```
let power n = run .<fun x -> let r = ref 1 in .~(power' n .<x>. .<r>.)>.
```

MetaOCaml [35] also provides an auxiliary primitive `genLet` to insert let-bindings for effectful code, which can be used as follows, avoiding an explicit `let`-expression:

```
let power n = run .<fun x -> .~(power' n .<x>. (genLet .<ref 1>.)>).
```

In either case, when refactoring an unstaged program into a staged one, the programmer must be peculiarly aware of whether the meta-program manipulates effectful code, and must manually indicate let-bindings to avoid duplication and unintended changes in behavior.

Staging with Automatic Let-Insertion. We have seen that simply adding staging annotations is not enough to correctly stage the *imperative* power function: without proper let-insertion, syntactic quotation does not in general preserve semantics. This is precisely why practical staging frameworks such as LMS, Squid, and BuildIt support *automatic let-insertion*, relieving programmers from having to insert let-bindings by hand.

To illustrate automatic let-insertion, we now switch from the quotation-based staging syntax to *combinator-based* syntax. Instead of building code with quotation and splicing, we use combinators that construct code values directly. This style goes back to two-level functional languages [39] and partial evaluation [29]. In practice, LMS elaborates surface syntax into such code-generating combinators via Scala's overloading mechanisms [43], and MetaOCaml translates quotations/splices into a combinator-based representation [35] similar to ours.

The following example shows a semantics-preserving staged version of the imperative power function in $\lambda_{|2|}^{\text{ref}}$ (Section 6). Rather than annotating with quotation and splicing, the programmer marks (infix) *operations* with the stage annotation 2. For instance, dereference, assignment, and multiplication on `res` are all marked with 2, indicating that they should occur at the second stage. Variables themselves require no splicing or annotation; they simply refer to code values.

```
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)
```

To drive specialization, we residualize the allocation, use `lift` to promote first-stage values to the second stage, and apply `run` to obtain the specialized function:

```
let power n = run (lift (fun x -> power' n x (ref2 (lift 1))))
```

Of course, the combinator-based syntax alone does not guarantee let-insertion. What matters is the dynamic semantics of $\lambda_{|2|}^{\text{ref}}$, which produces the desired residual program that runs faster:

```
fun x -> let res = ref 1 in (* example: specialized for n = 2 *)
  res := x * !res; res := x * !res; !res
```

Crucially, the generated code allocates `res` only once, and all subsequent dereferences and assignments refer to that same location. The programmer does not need to insert any let-bindings manually, yet the generated code can still serve as a drop-in replacement for computing x^2 .

2.2 Key Ideas and Main Results

In the example above, the let-inserted staged power clearly generates code semantically equivalent to the reference implementation, namely the unstaged power function. However, formally establishing this property is far from trivial, especially in the presence of the subtle interactions between staging annotations and effects. Our approach is to connect staged and unstaged programs via *erasure*, and connect generated code and unstaged programs via *contextual equivalence*. Figure 1 illustrates the high-level proof structure of our approach. Below, we present the key ideas behind our calculi and give an overview of our results.

Syntax and Dynamic Semantics. The $\lambda_{|2|}$ - and $\lambda_{|2|}^{\text{ref}}$ -calculi share the same foundation of syntax and semantics, with the latter extending the former with mutable references. We adopt the dynamic semantics of Amin and Rompf [3]’s untyped $\lambda_{\uparrow\downarrow}$ -calculus, which formulates let-insertion as a reduction semantics and captures operational aspects found in practical staging frameworks (such as LMS [43, 45, 46]) and partial evaluation for sound specialization with effects [15, 16, 20, 31, 37]. With let-insertion, generated code is organized as a sequence of let-bindings, similar to A-normal form (ANF) [22] and monadic normal form (MNF) [24].

Like MetaML, our calculi provide a run operation for executing generated code at runtime and a lift operation for promoting first-stage values to second-stage values. However, unlike MetaML’s lift, which is restricted to ground values, lift in our calculi also applies to λ -terms, where it acts as a two-level η -expansion and evaluates the function body [3, 18]. For example, the following code uses lift to promote a first-stage function of `int code -> int code` to the second stage:

```
let f1 (x: int code) : int code = x +2 (lift 1) +2 (lift 2) in
lift f1
```

The resulting generated code has type `(int -> int) code`. Its body sequentializes the evaluation order through a sequence of proper let-expressions:

```
fun x -> let x0 = x + 1 in let x1 = x0 + 2 in x1.
```

Type-and-Effect System. Although automatic let-insertion has been formalized in prior work [3], a type system that guarantees type soundness is new in this work. Keeping erasure soundness in mind, the challenge is that code cannot be treated as ordinary pure values: they cannot be duplicated, reordered, or discarded arbitrarily. A simply typed system cannot rule out programs that violate these constraints. This issue is particularly acute for effectful second-stage code, whose let-insertion points must precisely preserve effects and evaluation order. For example, consider the following program, which attempts to construct second-stage print code yet discards it:

```
let x = print2 "Hello" in 42
```

Since `x` is unused, one might expect the first stage to evaluate to 42, thereby discarding the generated print. But admitting this program as well-typed would violate semantics preservation, since the erased program should still print "Hello".

To rule out such programs, we develop lightweight type-and-effect systems for $\lambda_{|2|}$ and $\lambda_{|2|}^{\text{ref}}$ that treat let-inserting code generation uniformly as a control effect rather than as pure value construction. The control effect identifies a delimited reification boundary around the focused expression and lifts that expression to a let-binding within that boundary.

Accordingly, the type system distinguishes intermediate code fragments (`frag  $\tau$`) from complete code values (`rep  $\tau$`). Code fragments of type `frag  $\tau$` are introduced by let-insertion. Once named by code variables, they are pure and can be used freely. Complete code values `rep  $\tau$` , in contrast, represent fully generated code that may contain arbitrary second-stage effects such as divergence, and thus cannot be duplicated or discarded arbitrarily. The type system tracks pending reification

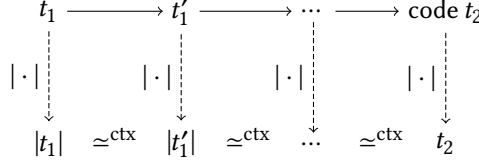


Fig. 1. Illustration of semantics preservation and our proof structure.

as a control effect through an effect flag, ensuring that fragments are properly let-inserted before being used.

Metatheory. By combining a first-stage effect system with a type-level distinction between complete code values and intermediate code fragments, we obtain a simple and robust foundation for proving syntactic type soundness, *i.e.*, progress and preservation, which further ensures that the generated program is also well-typed.

Our main technical contribution is the semantics-preservation theorem relating staging annotations to their inverse operation, erasure. To show that, instead of reasoning about the staged program itself, we reason about the result of such a staged program (*i.e.*, the generated code) and its equivalence with the erased program via binary step-indexed Kripke logical relations. Informally, the theorem (Section 5) states that if a well-typed two-stage program t_1 generates code t_2 , then t_2 is contextually equivalent to the erased program $|t_1|$, where $|\cdot|$ removes all staging annotations. As illustrated in Figure 1, we establish a stronger stepwise property: every reduction step preserves contextual equivalence between erased terms, *i.e.*, whenever $t \rightarrow t'$, we have $|t| \simeq^{\text{ctx}} |t'|$. It then follows that $|t_1| \simeq^{\text{ctx}} t_2$ where $t_1 \rightarrow^* \text{code } t_2$, since the erasure of code t_2 is just t_2 .

Extending with Store Effects. The base calculus $\lambda_{|2|}$ already admits divergence, and we show that let-insertion preserves semantics between staged programs and their erasures, yielding strong call-by-value reasoning principles for multi-stage programming. Our extension with mutable references further clarifies the scope of this guarantee: contrary to common intuition, let-insertion alone does not ensure semantic preservation in the presence of arbitrary effects during staging.

To study this interaction, we extend $\lambda_{|2|}$ with mutable references, yielding $\lambda_{|2|}^{\text{ref}}$. In Section 6, we characterize how first-stage store effects can violate semantic preservation and establish a useful sufficient condition: restricting store effects to the second stage. This restriction supports a common use case, *i.e.*, soundly generating and manipulating effectful code, and lays the foundation for future type-and-effect systems that distinguish unsafe first-stage effects from admissible ones. By adapting both the reduction semantics and logical relation, we prove semantic preservation for $\lambda_{|2|}^{\text{ref}}$. We further discuss possible paths toward a more expressive system (Section 7), drawing on recent type systems for tracking capture/reachability and establishing separation [6, 12, 38, 60, 69].

3 $\lambda_{|2|}$: A Semantics-Preserving Two-Stage Calculus with Recursion

We now formally present the base $\lambda_{|2|}$ -calculus that enjoys erasure soundness via automatic let-insertion. Following Inoue and Taha [27, 28], we consider a pure setting without mutable references but allow divergence. $\lambda_{|2|}$ is a statically typed, call-by-value, two-stage λ -calculus with general recursion and automatic let-insertion. It specializes the $\lambda_{\uparrow\downarrow}$ -calculus's reduction semantics [3] to two stages and equips it with a type-and-effect system. Building on the $\lambda_{\uparrow\downarrow}$'s account of let-insertion, our type system ensures both syntactic type soundness and semantic preservation under erasure.

We first present the surface syntax of $\lambda_{|2|}$, then its dynamic semantics with administrative and runtime constructs, and finally its type-and-effect system.

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

3.1 Surface Syntax

Figure 2 presents the surface syntax of $\lambda_{|2|}$. The syntax follows the two-level λ -calculus [39]; we adopt Curry-style syntax and defer the type system to Section 3.3. $\lambda_{|2|}$ is designed to manipulate and generate second-stage code. Stage superscripts $s \in \{1, 2\}$ annotate function application (app^s), arithmetic operations ($\oplus^s$), fixed-point (fix^s), and conditionals (ifz^s). Unlike the $\lambda_{\uparrow\downarrow}$ -calculus [3] that uses self-references of λ -terms for recursion, we use a separate fixed-point operator (the Z combinator) that is compatible with call-by-value evaluation.

Let-bindings in the surface language are stage-agnostic: they are evaluated in the first stage, yet the bound term and body can be first- or second-stage. Simple expressions (unit, numbers, variables, λ -abstractions) are annotation-free and can be lifted. Given a first-stage term t , $\text{lift } t$ evaluates t and produces a code fragment representing its value. Conversely, $\text{run } t$ evaluates a code value code t and produces a first-stage value (similar to eval in Lisp). Together with $\text{lift } t$ and $\text{run } t$, they support runtime code generation and evaluation.

3.2 Dynamic Semantics

The key aspect of $\lambda_{|2|}$'s dynamic semantics is let-insertion, formalized as reduction steps following Amin and Rompf [3]. Figure 3 defines the additional syntactic categories required by this semantics, including the administrative terms, values, and evaluation contexts. Administrative terms g occur only at runtime and comprise code fragments code t , reflections reflect t , code let-bindings $\text{let}_c x = t_1 \text{ in } t_2$, and code functions $\lambda_c x.t$. Values are those of the standard λ -calculus, extended with code values of the form code t .

Two-Stage Reduction Contexts. Evaluation in $\lambda_{|2|}$ proceeds in two stages under a small-step reduction relation. First, a term t reduces to a value v , written $t \rightarrow^* v$. When $v = \text{code } t'$, the second stage evaluates the generated term t' to produce the final value. As $\lambda_{|2|}$ is a two-stage language, t' contains no remaining staging constructs, i.e., no lift, run, or 2-annotations.

Figure 3 defines the pure context E for standard CBV evaluation and the reification context P for let-insertion, under which the reduction rules apply. Pure contexts E is either empty (i.e., a hole), or a sequence of pure frames B , which are standard evaluation frames (i.e., single terms with a hole) for left-to-right, call-by-value evaluation. We call them “pure” because they do not constrain the scope of let-insertion. Context composition $B \circ E$ denotes plugging E into the hole of B , i.e., $B[E]$, which is still a context.

Reification contexts P are either empty or Q -contexts. Context Q is built from reification frames R and pure frames B : the innermost frame in Q must be an R -frame, which identifies the let-insertion boundary when it contains a reflection reflect t . R -frames allow evaluation under second-stage binders (e.g., in a λ_c - or let_c -body), inside branches of second-stage conditionals, or under run. These P - and E -contexts mark the boundary where let-insertion can occur; the reduction rules below make this precise. Full evaluation contexts M allow any interleaving of B and R .

Pure Reduction. In Figure 3, $t \rightsquigarrow t'$ denotes the single-step reduction under an evaluation context M (ST-PURE). Rules annotated with $\mathbb{1}$ are standard call-by-value reduction; δ denotes primitive arithmetic in ST-OP $^{\mathbb{1}}$. The first-stage fixed-point operator (ST-FIX $^{\mathbb{1}}$) η -expands the function value v with the fixed function itself and the ordinary argument, following the Z combinator.

Terms	t	$::=$	$\dots \mid g$
Administrative 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	$::=$	$n \mid () \mid \lambda x. t \mid \text{code } t$
Pure Frames	B	$::=$	$\text{app}^s \square t \mid \text{app}^s v \square \mid \square \oplus^s t \mid v \oplus^s \square \mid \text{fix}^s \square$ $\mid \text{ifz}^s \square t_2 t_3 \mid \text{lift } \square \mid \text{let } x = \square \text{ in } t$
Pure Contexts	E	$::=$	$\square \mid B \circ E$
Reification Frames	R	$::=$	$\lambda_c x. \square \mid \text{let}_c x = t \text{ in } \square \mid \text{run } \square \mid \text{ifz}^2 v \square t \mid \text{ifz}^2 v_1 v_2 \square$
Reification Contexts	P	$::=$	$\square \mid Q$
	Q	$::=$	$R \mid B \circ Q \mid R \circ Q$
Evaluation Contexts	M	$::=$	$\square \mid B \circ M \mid R \circ M$

Reduction

$$\boxed{t \rightsquigarrow t'} \quad \boxed{t \rightarrow t'}$$

(ST-LET)	$\text{let } x = v \text{ in } t$	$\rightsquigarrow$	$t[v/x]$	
(ST-APP ¹)	$\text{app}^1(\lambda x. t) v$	$\rightsquigarrow$	$t[v/x]$	
(ST-APP ²)	$\text{app}^2(\text{code } t_1)(\text{code } t_2)$	$\rightsquigarrow$	$\text{reflect}(\text{app}^1 t_1 t_2)$	
(ST-OP ¹)	$n_1 \oplus^1 n_2$	$\rightsquigarrow$	n_3	where $n_3 = \delta(\oplus, n_1, n_2)$
(ST-OP ²)	$(\text{code } t_1) \oplus^2 (\text{code } t_2)$	$\rightsquigarrow$	$\text{reflect}(t_1 \oplus^1 t_2)$	
(ST-NAT [↑])	$\text{lift } n$	$\rightsquigarrow$	$\text{reflect } n$	
(ST-UNIT [↑])	$\text{lift } ()$	$\rightsquigarrow$	$\text{reflect } ()$	
(ST-LAM [↑])	$\text{lift } (\lambda x. t)$	$\rightsquigarrow$	$\lambda_c x. t[\text{code } x/x]$	
(ST-LAM _c)	$\lambda_c x. (\text{code } t)$	$\rightsquigarrow$	$\text{reflect}(\lambda x. t)$	
(ST-LET _c)	$\text{let}_c x = t_1 \text{ in } (\text{code } t_2)$	$\rightsquigarrow$	$\text{code}(\text{let } x = t_1 \text{ in } t_2)$	
(ST-RUN)	$\text{run}(\text{code } t)$	$\rightsquigarrow$	t	
(ST-FIX ¹)	$\text{fix}^1 v$	$\rightsquigarrow$	$\lambda x. \text{app}^1(\text{app}^1 v (\text{fix}^1 v)) x$	
(ST-FIX ²)	$\text{fix}^2(\text{code } t)$	$\rightsquigarrow$	$\text{reflect}(\text{fix}^1 t)$	
(ST-IFZ ¹ -0)	$\text{ifz}^1 0 t_1 t_2$	$\rightsquigarrow$	t_1	
(ST-IFZ ¹ -N)	$\text{ifz}^1 n t_1 t_2$	$\rightsquigarrow$	t_2	where $n \neq 0$
(ST-IFZ ²)	$\text{ifz}^2(\text{code } t_1)(\text{code } t_2)(\text{code } t_3)$	$\rightsquigarrow$	$\text{reflect}(\text{ifz}^1 t_1 t_2 t_3)$	
$\frac{t \rightsquigarrow t'}{M[t] \rightarrow M[t']} \text{ (ST-PURE)} \quad \frac{x \text{ is fresh}}{P[E[\text{reflect } t]] \rightarrow P[\text{let}_c x = t \text{ in } E[\text{code } x]]} \text{ (ST-REFLECT)}$				

Fig. 3. Additional syntax and reduction semantics of $\lambda_{|2|}$.

Reduction rules annotated with 2 “reflect” the corresponding first-stage term when all subterms have evaluated to code values. For example, ST-APP² steps to $\text{reflect}(\text{app}^1 t_1 t_2)$, where t_1 and t_2 are the function and argument code fragments, respectively.

Lifting a value likewise steps to reflecting. For a λ -value, lifting replaces the bound variable x with the neutral code variable $\text{code } x$ in the body t (ST-LAM[↑]), yielding $\lambda_c x. t[\text{code } x/x]$. Any latent first-stage computation in the resulting body may then reduce further. This rule follows the normalization-by-evaluation [10, 11] and type-directed partial-evaluation [15] technique of normalizing a function by applying it to a neutral variable. It also models the lifting mechanism used in staging frameworks such as LMS [44, 46].

Reduction with Let-Insertion. The let-insertion rule (ST-REFLECT) is central to $\lambda_{|2|}$ ’s dynamics. A term $\text{reflect } t$ marks a let-insertion site but is not itself a redex. When it appears within a reification

Reification Effects $\epsilon ::= \perp \mid \top$
 Types $\tau ::= \text{unit} \mid \text{nat} \mid \tau_1 \rightarrow^\epsilon \tau_2 \mid \text{rep } \tau \mid \text{frag } \tau$
 Typing Environments $\Gamma ::= \emptyset \mid \Gamma, x^s : \tau$

Fig. 4. Syntax for $\lambda_{|2|}$'s static semantics.

Type and Binding-time Well-formedness

 $\text{WF}^s \tau$

$$\begin{array}{c}
 \frac{\text{WF}^s \tau_1 \quad \text{WF}^s \tau_2 \quad s = 2 \implies \epsilon = \perp}{\text{WF}^s (\tau_1 \rightarrow^\epsilon \tau_2)} \quad \text{(WF-ARROW)} \qquad \frac{\tau = \text{unit} \vee \tau = \text{nat}}{\text{WF}^s \tau} \quad \text{(WF-BASE)} \qquad \frac{\text{WF}^2 \tau}{\text{WF}^1 (\text{frag } \tau)} \quad \text{(WF-FRAG)}
 \end{array}$$

Fig. 5. Type and binding-time well-formedness.

context P and a pure context E , the rule inserts a let_c -binding $x = t$ where x is fresh, and replaces $\text{reflect } t$ with code x in context E . Here, P is the largest context whose innermost frame is R and may contain E . Because t cannot depend on bindings introduced by the inner E , hoisting the binding into P is sound. The resulting let_c -binding represents an intermediate code-generation step: the right-hand side has been reflected/reified (thus represented as a first-stage term), while the body may still contain unevaluated code. Once the body becomes a code value, rule (ST-LET_C) reduces the expression to code containing an ordinary let-binding.

Determinism. A term decomposes uniquely as either $M[t_1]$ or $P[E[\text{reflect } t_2]]$, and the P/E -context decomposition is also unique, so exactly one of ST-PURE and ST-REFLECT applies. We later use determinism to prove transitivity of contextual equivalence (Theorem 5.3).

LEMMA 3.1 (DETERMINISTIC DECOMPOSITION). *If $P[E[\text{reflect } t]] = P'[E'[\text{reflect } t']]$, then $P = P'$, $E = E'$, and $t = t'$.*

THEOREM 3.2 (DETERMINISM). *If $t_1 \rightarrow t_2$ and $t_1 \rightarrow t_3$, then $t_2 = t_3$.*

3.3 Static Semantics

$\lambda_{|2|}$ features a lightweight type-and-effect system that ensures type-safe, semantics-preserving manipulation of code fragments. Figure 4 presents the syntax of types, effects, and typing contexts. Types τ include base types (unit and numbers), function types $\tau_1 \rightarrow^\epsilon \tau_2$ with latent effect ϵ , and code types $\text{rep } \tau$ and $\text{frag } \tau$. Typing contexts Γ map variables x to types τ and stage annotations s .

Types and Effects for Code Fragments. We distinguish two code types: $\text{rep } \tau$ for complete code values and $\text{frag } \tau$ for fragments not yet reified. In $\lambda_{|2|}$ (and $\lambda_{|2|}^{\text{ref}}$), general code values may represent effectful computations (e.g., divergence and states), therefore to preserve their semantics (Section 5), they must not be arbitrarily reordered, copied, or discarded. In contrast, a variable referring to a reified code introduced by let-insertion (i.e. code x) is pure, and can be used unrestrictedly. Exploiting this observation, our key insight is to assign the unrestricted type $\text{frag } \tau$ both to the pure code reference code x and to the terms that let-insertion will later be replaced with code x . Complete code values of type $\text{rep } \tau$ may contain arbitrary second-stage effects, and cannot be bound to variables. Moreover, code fragments of type $\text{frag } \tau$ can only appear in specific contexts that guarantee they will eventually be reified into code values before being run.

Code generation constructs (e.g., reflect and app^2) behave as a delimited control effect: they trigger reification, and the surrounding R frame delimits where that effect is handled. Outside of the R context, the reification effect within the R context is not observable. Therefore, in addition to

Typing Judgments for Surface Terms

		$\boxed{\Gamma \vdash^s t : \tau \mid \epsilon}$	$\boxed{\Gamma \vdash t : \tau \mid \epsilon}$
$\frac{x : \tau^s \in \Gamma \quad \text{WF}^s \tau}{\Gamma \vdash^s x : \tau \mid \perp} \text{ (T-VAR)}$	$\frac{}{\Gamma \vdash^s n : \text{nat} \mid \perp} \text{ (T-NAT)}$	$\frac{}{\Gamma \vdash^s () : \text{unit} \mid \perp} \text{ (T-UNIT)}$	
$\frac{\Gamma \vdash^{\mathbb{1}} t : (\text{frag } \tau_1 \rightarrow^{\epsilon_1} \text{frag } \tau_2) \mid \epsilon_2}{\Gamma \vdash^{\mathbb{1}} \text{lift } t : \text{frag } (\tau_1 \rightarrow^{\perp} \tau_2) \mid \top} \text{ (T-LAM}^{\uparrow})$	$\frac{\Gamma, x^s : \tau_1 \vdash^s t : \tau_2 \mid \epsilon \quad \text{WF}^s \tau_1}{\Gamma \vdash^s \lambda x. t : \tau_1 \rightarrow^{\epsilon} \tau_2 \mid \perp} \text{ (T-LAM)}$		
$\frac{\Gamma \vdash^{\mathbb{1}} t : \text{unit} \mid \epsilon}{\Gamma \vdash^{\mathbb{1}} \text{lift } t : \text{frag unit} \mid \top} \text{ (T-UNIT}^{\uparrow})$	$\frac{\Gamma \vdash^{\mathbb{1}} t : \text{nat} \mid \epsilon}{\Gamma \vdash^{\mathbb{1}} \text{lift } t : \text{frag nat} \mid \top} \text{ (T-NAT}^{\uparrow})$		
$\frac{\Gamma \vdash^s t_1 : (\tau_1 \rightarrow^{\epsilon} \tau_2) \mid \epsilon_1 \quad \Gamma \vdash^s t_2 : \tau_1 \mid \epsilon_2}{\Gamma \vdash^s \text{app}^{\mathbb{1}} t_1 t_2 : \tau_2 \mid (\epsilon \sqcup \epsilon_1 \sqcup \epsilon_2)} \text{ (T-APP}^{\mathbb{1}})$	$\frac{\Gamma \vdash^{\mathbb{1}} t_1 : \text{frag } (\tau_1 \rightarrow^{\perp} \tau_2) \mid \epsilon_1 \quad \Gamma \vdash^{\mathbb{1}} t_2 : \text{frag } \tau_1 \mid \epsilon_2}{\Gamma \vdash^{\mathbb{1}} \text{app}^2 t_1 t_2 : \text{frag } \tau_2 \mid \top} \text{ (T-APP}^2)$		
$\frac{\Gamma \vdash^s t_1 : \text{nat} \mid \epsilon_1 \quad \Gamma \vdash^s t_2 : \text{nat} \mid \epsilon_2}{\Gamma \vdash^s t_1 \oplus^{\mathbb{1}} t_2 : \text{nat} \mid (\epsilon_1 \sqcup \epsilon_2)} \text{ (T-OP}^{\mathbb{1}})$	$\frac{\Gamma \vdash^{\mathbb{1}} t_1 : \text{frag nat} \mid \epsilon_1 \quad \Gamma \vdash^{\mathbb{1}} t_2 : \text{frag nat} \mid \epsilon_2}{\Gamma \vdash^{\mathbb{1}} t_1 \oplus^2 t_2 : \text{frag nat} \mid \top} \text{ (T-OP}^2)$		
$\frac{\Gamma \vdash^s t_1 : \text{nat} \mid \epsilon_1 \quad \Gamma \vdash^s t_2 : \tau \mid \epsilon_2 \quad \Gamma \vdash^s t_3 : \tau \mid \epsilon_3}{\Gamma \vdash^s \text{ifz}^{\mathbb{1}} t_1 t_2 t_3 : \tau \mid (\epsilon_1 \sqcup \epsilon_2 \sqcup \epsilon_3)} \text{ (T-IFZ}^{\mathbb{1}})$	$\frac{\Gamma \vdash^{\mathbb{1}} t_1 : \text{frag nat} \mid \epsilon_1 \quad \Gamma \vdash t_2 : \text{rep } \tau \mid \epsilon_2 \quad \Gamma \vdash t_3 : \text{rep } \tau \mid \epsilon_3}{\Gamma \vdash^{\mathbb{1}} \text{ifz}^2 t_1 t_2 t_3 : \text{frag } \tau \mid \top} \text{ (T-IFZ}^2)$		
$\frac{\epsilon_1 = \epsilon_1 \sqcup \epsilon_2 \quad \tau = \tau_1 \rightarrow^{\epsilon_1} \tau_2 \quad \Gamma \vdash^s t : \tau \rightarrow^{\epsilon_2} \tau \mid \epsilon_3}{\Gamma \vdash^s \text{fix}^{\mathbb{1}} t : \tau \mid \epsilon_3} \text{ (T-FIX}^{\mathbb{1}})$	$\frac{\tau = \tau_1 \rightarrow^{\perp} \tau_2 \quad \Gamma \vdash^{\mathbb{1}} t : \text{frag } (\tau \rightarrow^{\perp} \tau) \mid \epsilon}{\Gamma \vdash^{\mathbb{1}} \text{fix}^2 t : \text{frag } \tau \mid \top} \text{ (T-FIX}^2)$		
$\frac{\Gamma \vdash^s t_1 : \tau_1 \mid \epsilon_1 \quad \Gamma, x^s : \tau_1 \vdash^s t_2 : \tau_2 \mid \epsilon_2 \quad \text{WF}^s \tau_1}{\Gamma \vdash^s \text{let } x = t_1 \text{ in } t_2 : \tau_2 \mid (\epsilon_1 \sqcup \epsilon_2)} \text{ (T-LET)}$	$\frac{\emptyset \vdash t : \text{rep } \tau \mid \epsilon}{\Gamma \vdash^{\mathbb{1}} \text{run } t : \tau \mid \perp} \text{ (T-RUN)}$		
$\frac{\Gamma \vdash^{\mathbb{1}} t : \tau \mid \perp}{\Gamma \vdash t : \tau \mid \perp} \text{ (T-PURE)}$	$\frac{\Gamma \vdash^{\mathbb{1}} t : \text{frag } \tau \mid \epsilon}{\Gamma \vdash t : \text{rep } \tau \mid \epsilon} \text{ (T-REP)}$		

Fig. 6. Typing rules for $\lambda_{|2|}$'s surface syntax.

the type distinction, we track if let-insertion could happen in a term using an effect system. Effect labels ϵ indicate whether a term may perform code generation and let-insertion effects ($\top$) or not ($\perp$). Effects naturally form a lattice where $\perp \sqsubseteq \top$, and $\epsilon_1 \sqcup \epsilon_2 = \epsilon_2$ iff $\epsilon_1 \sqsubseteq \epsilon_2$.

Binding-Time Well-Formedness. Figure 5 defines stage-indexed type well-formedness, which the typing rules use to constrain variable stages. Base types nat and unit are well formed at both stages. A function type $\tau_1 \rightarrow^{\epsilon} \tau_2$ is well formed at stage $\mathbb{1}$ if both argument and result types are; at stage 2, both must be well formed at 2 and no further code generation is allowed in the function body ($\epsilon = \perp$). A code fragment type $\text{frag } \tau$ is well formed only at stage $\mathbb{1}$, with τ well formed at 2. In contrast, $\text{rep } \tau$ is *ill-formed at both stages* (thus no corresponding rules), since allowing such code values (other than code x) to be bound would enable code duplication or discarding.

Typing Surface Terms. There are two mutually recursive typing judgments: the top-level judgment $\Gamma \vdash t : \tau \mid \epsilon$ and binding-time sensitive judgment $\Gamma \vdash^s t : \tau \mid \epsilon$. The additional binding-time

Typing Judgments for Administrative Terms

$$\boxed{\Gamma \vdash^s g : \tau \mid \epsilon}$$

$$\begin{array}{c}
\frac{x : \tau^2 \in \Gamma \quad \text{WF}^2 \tau}{\Gamma \vdash^{\mathbb{1}} \text{code } x : \text{frag } \tau \mid \perp} \quad \frac{\Gamma \vdash^2 t : \tau \mid \perp}{\Gamma \vdash^{\mathbb{1}} \text{code } t : \text{rep } \tau \mid \perp} \quad \frac{\Gamma \vdash^2 t : \tau \mid \perp}{\Gamma \vdash^{\mathbb{1}} \text{reflect } t : \text{frag } \tau \mid \top} \\
\text{(T-CODE-X)} \quad \text{(T-CODE)} \quad \text{(T-REFL)} \\
\\
\frac{\Gamma \vdash^2 t_1 : \tau_1 \mid \perp \quad \Gamma, x^2 : \tau_1 \vdash t_2 : \text{rep } \tau_2 \mid \epsilon \quad \text{WF}^2 \tau_1}{\Gamma \vdash^{\mathbb{1}} \text{let}_c x = t_1 \text{ in } t_2 : \text{rep } \tau_2 \mid \perp} \text{(T-LET}_c\text{)} \quad \frac{\Gamma, x : \tau_1^2 \vdash t : \text{rep } \tau_2 \mid \epsilon \quad \text{WF}^2 \tau_1}{\Gamma \vdash^{\mathbb{1}} \lambda_c x.t : \text{frag } (\tau_1 \rightarrow^\perp \tau_2) \mid \top} \text{(T-LAM}_c\text{)}
\end{array}$$

Fig. 7. Typing rules for $\lambda_{|2|}$'s administrative terms.

s indicates that term t is checked at stage s . Since $\lambda_{|2|}$ is a two-stage language, clearly some terms can only be checked at the first stage, including lift t , run t , all 2-terms, and all administrative terms.

Figure 6 presents the typing rules for $\lambda_{|2|}$'s surface terms. Most cases for $\mathbb{1}$ -terms are standard. Most rules for 2-terms assign frag-types, and require the sub-terms to have frag-types too; they also have the $\top$ effect, indicating the term will be reflected and reified by let-insertion later.

In the following, we discuss several interesting rules. Rule T-LAM $^\uparrow$ specifies that lifting a term t of first-stage function type whose argument and return types are both code fragments yields a second-stage function. Both t 's effect ϵ_2 and the function's latent effect ϵ_1 are materialized, thus the resulting second-stage function has no latent let-insertion effect and the effect of the lifting is $\top$. Rule T-RUN requires t to be closed, since $\lambda_{|2|}$ does not support cross-stage persistence.

The first-stage fixpoint operator (T-FIX $^\mathbb{1}$) requires that the actual recursive function's latent effect ϵ_1 is the upper bound effect, since we substitute the fixed function itself into the body. The second-stage conditional expression (T-IFZ 2) allows the branches to be of general code type $\text{rep } \tau$, which can be either terms of frag-type via rule T-REP, or other code constructs from administrative terms (Figure 7) explained below.

Typing Administrative Terms. Figure 7 shows the additional typing rules for administrative terms. Code variables $\text{code } x$ introduced by let-insertion are pure, so they can be typed as $\text{frag } \tau$ (T-CODE-X) or generally $\text{rep } \tau$ (T-CODE). Rule T-CODE-X requires the variable x to be bound at stage 2 with τ well-formed at 2.

Reflection (T-REFL) and λ_c -terms (T-LAMC) are intermediate reduction constructs eventually reified via let-insertion, therefore are typed as $\text{frag } \tau$ with effect $\top$. The function body of a λ_c -term may have arbitrary effects, which are materialized by evaluation (under an R context), therefore the resulting second-stage function has no latent effect. For let_c -bindings (T-LET $_c$), the right-hand side t_1 is a reified code and is typed under $\vdash^2$. The body t_2 may perform further first-stage evaluation and requires the general code type $\text{rep } \tau_2$.

Running Example: Ruling Out Stuck Terms. Recall from Section 2.2 that programs such as `let x = print2 "Hello" in 42` are ill-typed. To see this, consider the typing derivation: the surface typing derives $\text{nat} \mid \top$ at stage $\mathbb{1}$, but neither T-PURE nor T-REP applies for complete programs:

$$\begin{array}{c}
\frac{\Gamma \vdash^{\mathbb{1}} \text{print}^2 \text{"Hello"} : \text{frag unit} \mid \top \quad \Gamma, x^{\mathbb{1}} : \text{frag unit} \vdash^{\mathbb{1}} 42 : \text{nat} \mid \perp \quad \text{WF}^{\mathbb{1}} \text{frag unit}}{\Gamma \vdash^{\mathbb{1}} \text{let } x = \text{print}^2 \text{"Hello"} \text{ in } 42 : \text{nat} \mid \top} \text{(T-LET)} \\
\Gamma \vdash \text{let } x = \text{print}^2 \text{"Hello"} \text{ in } 42 : \tau \mid \epsilon \quad \text{(Neither T-PURE nor T-REP applies } \times \text{)}
\end{array}$$

Term, Type, and Environment Erasure

$$\boxed{|t| = t'} \quad \boxed{|\tau| = \tau'} \quad \boxed{|\Gamma| = \Gamma'}$$

$$\begin{array}{lll}
|()| = () & |\text{fix}^s t| = \text{fix}^{\mathbb{1}} |t| & |\text{unit}| = \text{unit} \\
|n| = n & |\text{ifz}^s t_1 t_2 t_3| = \text{ifz}^{\mathbb{1}} |t_1| |t_2| |t_3| & |\text{nat}| = \text{nat} \\
|x| = x & |t_1 \oplus^s t_2| = |t_1| \oplus^{\mathbb{1}} |t_2| & |\tau_1 \rightarrow^\epsilon \tau_2| = |\tau_1| \rightarrow^{\perp} |\tau_2| \\
|\text{lift } t| = |t| & |\text{code } t| = |t| & |\text{rep } \tau| = |\tau| \\
|\text{run } t| = |t| & |\text{reflect } t| = |t| & |\text{frag } \tau| = |\tau| \\
|\text{app}^s t_1 t_2| = \text{app}^{\mathbb{1}} |t_1| |t_2| & |\lambda x.t| = \lambda x. |t| & |\emptyset| = \emptyset \\
|\text{let } x = t_1 \text{ in } t_2| = \text{let } x = |t_1| \text{ in } |t_2| & |\lambda_c x.t| = \lambda x. |t| & |\Gamma, x^s : \tau| = |\Gamma|, x^{\mathbb{2}} : |\tau| \\
|\text{let}_c x = t_1 \text{ in } t_2| = \text{let } x = |t_1| \text{ in } |t_2| & &
\end{array}$$

Fig. 8. Term, type, and environment erasure for $\lambda_{|2|}$.

Rule T-PURE requires effect $\perp$, while T-REP requires an answer type of the form $\text{frag } \tau$. Thus, $\lambda_{|2|}$ rejects the program, preventing evaluation from producing a let_c with a non-code body, on which the reduction semantics would get stuck.

3.4 Metatheory: Syntactic Type Soundness

The $\lambda_{|2|}$ -calculus enjoys the syntactic type soundness properties of progress and preservation [66].

Progress. Our type system is crucial for progress: many terms can get stuck in the reduction semantics, e.g. $\text{let}_c x = t$ in 0 when t is not a code value. As in typical staging calculi, $\lambda_{|2|}$ allows evaluation under future-stage binders, e.g., $\lambda_c x. \square$ and $\text{let}_c x = t$ in $\square$ are proper evaluation contexts in $\lambda_{|2|}$. Therefore, we need to prove a strengthened progress theorem under general typing contexts including $\mathbb{2}$ -bindings.

LEMMA 3.3 (STRENGTHENED PROGRESS). *If $\Gamma \vdash t : \tau \mid \epsilon$ and $\forall (x^s : \tau) \in \Gamma. s = \mathbb{2}$, then either t is a value v , or there exists t' such that $t \rightarrow t'$.*

THEOREM 3.4 (PROGRESS). *If $\emptyset \vdash t : \tau \mid \epsilon$, then t is a value or $\exists t' \text{ s.t. } t \rightarrow t'$.*

Preservation. To show type preservation, we must additionally prove that lifting λ -terms (ST-LAM[↑]) and let-insertion steps (ST-REFLECT) preserve typing. Preservation also states that effects are preserved: evaluation cannot introduce new effects. For the lifting of λ -terms, we prove a substitution lemma that handles substituting a free variable with an open value that may contain the same free variable. The standard substitution lemma still applies for the beta reduction case.

LEMMA 3.5 (OPEN SUBSTITUTION). *If $\Gamma, x^{\mathbb{1}} : \tau_1 \vdash^{\mathbb{1}} t : \tau_2 \mid \epsilon$ and $\Gamma, x^s : \tau_3 \vdash^{\mathbb{1}} v : \tau_1 \mid \perp$, then $\Gamma, x^s : \tau_3 \vdash^{\mathbb{1}} t[v/x] : \tau_2 \mid \epsilon$.*

THEOREM 3.6 (PRESERVATION). *If $\emptyset \vdash t : \tau \mid \epsilon$ and $t \rightarrow t'$, then $\exists \epsilon' \sqsubseteq \epsilon \text{ s.t. } \emptyset \vdash t' : \tau \mid \epsilon'$.*

4 Erasure and Syntactic Erasure Soundness

Recall that our goal is to ensure that the generated code from staging can be used as a drop-in replacement for the original unstaged program. The programmer begins with a single-stage program and introduces staging annotations where specialization is beneficial for performance. Whether code generation occurs at compile time or run time, the resulting code should remain extensional equivalent to the original program. In this section, we formally introduce the notion of erasure that connects the staged program and its unstaged counterpart.

Term, Type, and Environment Erasure. In Figure 8, we show the definition of term, type, and environment erasure. Term erasure $|t|$ converts all staging annotations s in term to $\mathbb{1}$. Additionally,

it strips administrative terms such as `code`, `reflect`, and λ_c , which are generated during staging. For example, code t is erased to $|t|$. Inserted `letc`-bindings are transformed to standard `let`-bindings.

Type erasure $| \tau |$ recursively removes code-related constructors `rep` and `frag` from types.

Since erased terms no longer induce reification effects, all latent effects in function types are replaced with $\perp$. Environment erasure $|\Gamma|$ applies type erasure to all items in the environment, and sets the binding-time to 2. Clearly, terms and types after erasure are a proper subset of those in $\lambda_{|2|}$. To avoid confusion, we refer to the language after erasure as a separate language $\lambda_{|2|}^\downarrow$.

Typing Erased Terms. Typings of $\lambda_{|2|}^\downarrow$ -terms are admissible under $\lambda_{|2|}$'s typing rules. We can precisely characterize the typing of erased terms using $\Gamma \vdash^2 t : \tau \mid \perp$, where Γ , t , and τ are all obtained by erasure. Intuitively, we are typing terms that coincide with the generated code. The `T-CODE` rule enforces exactly this property, requiring that the code body to be well-typed as $\Gamma \vdash^2 t : \tau \mid \perp$. This typing derivation also restricts staging operators from appearing in the program and enforces the absence of reification effects.

Now we show that erasure preserves syntactic typing:

THEOREM 4.1 (SYNTACTIC ERASURE SOUNDNESS). $\Gamma \vdash t : \tau \mid \epsilon$ implies $|\Gamma| \vdash^2 |t| : |\tau| \mid \perp$.

Reductions of $\lambda_{|2|}^\downarrow$ -terms are also admissible under $\lambda_{|2|}$'s reduction relation $t \rightarrow t'$, whereas rules such as `ST-REFLECT` cannot occur, since erased terms contain no 2-constructs or administrative staging constructs. When defining the logical relation, we do not distinguish between reductions in $\lambda_{|2|}^\downarrow$ and $\lambda_{|2|}$; instead, the binding-time sensitive typing ($\vdash^2$) regulates which steps can occur.

5 Metatheory: Semantics Preservation

With erasure defined, we now develop the metatheory to show that staged evaluation in $\lambda_{|2|}$ is semantics preserving. The central task is to show that generated code is contextually equivalent to its erased counterpart. To this end, we use step-indexed binary logical relations [1, 2] capturing contextual equivalence [8, 57]. We first introduce the necessary definitions of contextual equivalence, then build the logical relation, and finally prove that the multi-stage evaluation of $\lambda_{|2|}$ preserves semantics at every reduction step.

5.1 Contextual Approximation and Equivalence of $\lambda_{|2|}^\downarrow$

Two programs are contextually equivalent iff no well-formed observational context distinguishes them. Since both stage-erased programs and generated programs belong to $\lambda_{|2|}^\downarrow$, an observational context C is a partial $\lambda_{|2|}^\downarrow$ program with hole $\square$. We write $C : (\Gamma \vdash \tau) \Rightarrow (\Gamma' \vdash \tau')$ if $\Gamma \vdash^2 t : \tau \mid \perp$ entails $\Gamma' \vdash^2 C[t] : \tau' \mid \perp$. Figure 9 defines observational contexts and their typing rules.

Because $\lambda_{|2|}^\downarrow$ programs may diverge, we define contextual equivalence in terms of termination: two terms are indistinguishable when they either both terminate or both diverge. In particular, if two terminating terms produce different values, there must exist an observational context that can distinguish them by making one terminate and the other diverge.

Following the standard approach in the literature [1], we factor contextual equivalence into two directions of contextual approximation, i.e., termination of one program implies termination of the other. Contextual equivalence is their conjunction. Because the reduction relation is deterministic (Theorem 3.2), contextual equivalence is transitive.

Definition 5.1 (Contextual Approximation). If $\Gamma \vdash^2 t_1 : \tau \mid \perp$, and $\Gamma \vdash^2 t_2 : \tau \mid \perp$, and $\forall C : (\Gamma \vdash \tau) \Rightarrow (\emptyset \vdash \tau'). C[t_1] \Downarrow \implies C[t_2] \Downarrow$, then t_1 contextually approximates t_2 , written as $\Gamma \models t_1 \leq^{\text{ctx}} t_2 : \tau$. We write $t \Downarrow$ to denote that t terminates, i.e., $t \rightarrow^* v$ for some value v .

Observational Context for Contextual Equivalence

Frame $F ::= \lambda x. \square \mid \text{let } x = \square \text{ in } t \mid \text{let } x = t \text{ in } \square \mid \text{app}^1 \square t \mid \text{app}^1 t \square \mid \text{fix}^1 \square$
 $\mid \text{ifz}^1 \square t_2 t_3 \mid \text{ifz}^1 t_1 \square t_3 \mid \text{ifz}^1 t_1 t_2 \square \mid \square \oplus^1 t_2 \mid t_1 \oplus^1 \square$
 Contexts $C ::= \square \mid C \circ F$

Context Typing Rules

Typing Rules		$F : (\Gamma \vdash \tau) \Rightarrow (\Gamma' \vdash \tau')$	$C : (\Gamma \vdash \tau) \Rightarrow (\Gamma' \vdash \tau')$
$\frac{\text{WF}^2 \tau_1}{\lambda x. \Box : (\Gamma, x^2 : \tau_1 \vdash \tau_2) \Rightarrow (\Gamma \vdash \tau_1 \rightarrow^\perp \tau_2)} \quad (\text{F-LAM})$			
$\frac{\Gamma, x^2 : \tau_1 \vdash^2 t : \tau_2 \mid \perp}{\text{let } x = \Box \text{ in } t : (\Gamma \vdash \tau_1) \Rightarrow (\Gamma \vdash \tau_2)} \quad (\text{F-LET-1})$	$\frac{\Gamma \vdash^2 t : \tau_1 \mid \perp}{\text{let } x = t \text{ in } \Box : (\Gamma, x^2 : \tau_1 \vdash \tau_2) \Rightarrow (\Gamma \vdash \tau_2)} \quad (\text{F-LET-2})$		
$\frac{\Gamma \vdash^2 t : \tau_1 \mid \perp}{\text{app}^1 \Box t : (\Gamma \vdash \tau_1 \rightarrow^\perp \tau_2) \Rightarrow (\Gamma \vdash \tau_2)} \quad (\text{F-APP-1})$	$\frac{\Gamma \vdash^2 t : \tau_1 \rightarrow^\perp \tau_2 \mid \perp}{\text{app}^1 t \Box : (\Gamma \vdash \tau_1) \Rightarrow (\Gamma \vdash \tau_2)} \quad (\text{F-APP-2})$		
$\frac{\tau = \tau_1 \rightarrow^\perp \tau_2}{\text{fix}^1 \Box : (\Gamma \vdash \tau \rightarrow^\perp \tau) \Rightarrow (\Gamma \vdash \tau)} \quad (\text{F-FIX})$	$\frac{\Gamma \vdash^2 t_2 : \tau \mid \perp \quad \Gamma \vdash^2 t_3 : \tau \mid \perp}{\text{ifz}^1 \Box t_2 t_3 : (\Gamma \vdash \text{nat}) \Rightarrow (\Gamma \vdash \tau)} \quad (\text{F-IFZ-1})$		
$\frac{\Gamma \vdash^2 t_1 : \text{nat} \mid \perp \quad \Gamma \vdash^2 t_3 : \tau \mid \perp}{\text{ifz}^1 t_1 \Box t_3 : (\Gamma \vdash \tau) \Rightarrow (\Gamma \vdash \tau)} \quad (\text{F-IFZ-2})$	$\frac{\Gamma \vdash^2 t_1 : \text{nat} \mid \perp \quad \Gamma \vdash^2 t_2 : \tau \mid \perp}{\text{ifz}^1 t_1 t_2 \Box : (\Gamma \vdash \tau) \Rightarrow (\Gamma \vdash \tau)} \quad (\text{F-IFZ-3})$		
$\frac{\Gamma \vdash^2 t_2 : \text{nat} \mid \perp}{\Box \oplus^1 t_2 : (\Gamma \vdash \text{nat}) \Rightarrow (\Gamma \vdash \text{nat})} \quad (\text{F-OP-1})$	$\frac{\Gamma \vdash^2 t_1 : \text{nat} \mid \perp}{t_1 \oplus^1 \Box : (\Gamma \vdash \text{nat}) \Rightarrow (\Gamma \vdash \text{nat})} \quad (\text{F-OP-2})$		
$\frac{}{\Box : (\Gamma \vdash \tau) \Rightarrow (\Gamma \vdash \tau)} \quad (\text{C-HOLE})$		$\frac{C : (\Gamma' \vdash \tau') \Rightarrow (\Gamma'' \vdash \tau'') \quad F : (\Gamma \vdash \tau) \Rightarrow (\Gamma' \vdash \tau')}{C \circ F : (\Gamma \vdash \tau) \Rightarrow (\Gamma'' \vdash \tau'')} \quad (\text{C-CONS})$	

Fig. 9. Context typing rules for $\lambda_{|2|}^\perp$.

Definition 5.2 (Contextual Equiv.). $\Gamma \models t_1 \simeq^{\text{ctx}} t_2 : \tau \stackrel{\text{def}}{=} \Gamma \models t_1 \preceq^{\text{ctx}} t_2 : \tau \wedge \Gamma \models t_2 \preceq^{\text{ctx}} t_1 : \tau$.

THEOREM 5.3 (TRANSITIVITY). *If $\Gamma \models t_1 \simeq^{\text{ctx}} t_2 : \tau$ and $\Gamma \models t_2 \simeq^{\text{ctx}} t_3 : \tau$, then $\Gamma \models t_1 \simeq^{\text{ctx}} t_3 : \tau$.*

5.2 Binary Logical Relations

Figure 10 defines the binary logical relation. Because both erased terms and generated code admit divergence, the relation is step-indexed to be well-founded, following prior work [1, 2].

The value relation is given by the semantic interpretation of types, written $\mathcal{V}[\tau]$. For a type τ , it consists of triples (k, v_1, v_2) , where k is the step index (i.e., a natural number) and v_1, v_2 are closed values. Unit and numbers are related iff they are syntactically equal. Two λ -terms are related if applying them to related arguments at a future step index yields related results.

$$\begin{aligned}
\mathcal{V}[\text{unit}] &= \{(k, (), ())\} \\
\mathcal{V}[\text{nat}] &= \{(k, n, n)\} \\
\mathcal{V}[\tau_1 \rightarrow^\perp \tau_2] &= \{(k, \lambda x.t_1, \lambda x.t_2) \mid \forall j \leq k. \forall v_1, v_2. (j, v_1, v_2) \in \mathcal{V}[\tau_1] \Rightarrow \\
&\quad (j, \text{app}^\perp(\lambda x.t_1) v_1, \text{app}^\perp(\lambda x.t_2) v_2) \in \mathcal{E}[\tau_2]\} \\
\mathcal{E}[\tau] &= \{(k, t_1, t_2) \mid \forall j < k. \forall v_1. t_1 \rightarrow^j v_1 \Rightarrow \\
&\quad \exists v_2. t_2 \rightarrow^* v_2 \wedge (k-j, v_1, v_2) \in \mathcal{V}[\tau]\} \\
\mathcal{G}[\emptyset] &= \{(k, \emptyset, \emptyset)\} \\
\mathcal{G}[\Gamma, x^2 : \tau] &= \{(k, \gamma_1[x \mapsto v_1], \gamma_2[x \mapsto v_2]) \mid (k, \gamma_1, \gamma_2) \in \mathcal{G}[\Gamma] \wedge (k, v_1, v_2) \in \mathcal{V}[\tau]\} \\
\Gamma \models t_1 \leq^{\log} t_2 : \tau &\stackrel{\text{def}}{=} \Gamma \vdash^2 t_1 : \tau \mid \perp \wedge \Gamma \vdash^2 t_2 : \tau \mid \perp \wedge \\
&\quad \forall k \geq 0. \forall \gamma_1, \gamma_2. (k, \gamma_1, \gamma_2) \in \mathcal{G}[\Gamma] \Rightarrow (k, \gamma_1(t_1), \gamma_2(t_2)) \in \mathcal{E}[\tau] \\
\Gamma \models t_1 \simeq^{\log} t_2 : \tau &\stackrel{\text{def}}{=} \Gamma \models t_1 \leq^{\log} t_2 : \tau \wedge \Gamma \models t_2 \leq^{\log} t_1 : \tau
\end{aligned}$$

Fig. 10. The binary interpretation of types and terms, and logical equivalence for $\lambda_{|2|}^1$.

The term relation $\mathcal{E}[\tau]$ is likewise defined semantically by type. Intuitively, $(k, t_1, t_2) \in \mathcal{E}[\tau]$ means that whenever t_1 reduces to a value v_1 in $j < k$ steps, t_2 reduces to some value v_2 such that $(k-j, v_1, v_2) \in \mathcal{V}[\tau]$. The environment relation $\mathcal{G}[\Gamma]$ is defined inductively on the structure of type environments. In the inductive case, two substitutions are related if they map x to values related at type τ , extending substitutions γ_1 and γ_2 that are already related.

The logical approximation $\leq^{\log}$ relates two open terms under Γ if substituting their free variables with related substitutions γ_1 and γ_2 yields related closed terms. We write $\gamma(t)$ for the substitution of each free variable in t by its corresponding value in γ . Logical equivalence between t_1 and t_2 is then defined as approximation in both directions.

Soundness of Logical Relations. The compatibility lemmas are standard and omitted for brevity; they are the semantic counterparts of a subset of the typing rules (Figure 6) and can be found in the mechanized proof. These lemmas yield the fundamental theorem: a syntactically well-typed term is logically related to itself.

THEOREM 5.4 (FUNDAMENTAL PROPERTY). *If $\Gamma \vdash^2 t : \tau \mid \perp$, then $\Gamma \models t \simeq^{\log} t : \tau$.*

The following soundness theorem is our license to use the logical relation to prove contextual equivalence. Moreover, it serves as an important tool in proving the semantics preservation of staging, as contextual equivalence is often difficult to use directly in proofs.

THEOREM 5.5 (SOUNDNESS OF LOGICAL RELATIONS). *If $\Gamma \models t_1 \simeq^{\log} t_2 : \tau$, then $\Gamma \models t_1 \simeq^{ctx} t_2 : \tau$.*

5.3 Semantics Preservation of Staging

Overview. We now state the central meta-theorem of $\lambda_{|2|}$. If a two-stage program t_1 evaluates to code t_2 , then the generated code t_2 is contextually equivalent to the unstaged program $|t_1|$ obtained by erasing staging annotations. This equivalence is captured by the logical relations above. Thus, well-typed staging annotations neither introduce nor eliminate behaviors of the original unstaged program. This guarantee is strictly stronger than syntactic erasure soundness (Theorem 4.1), and we therefore call it *semantics preservation* of staging.

To prove this result, we first establish a stronger invariant: semantics are preserved by *every* reduction step, including those that produce administrative terms. Theorem 5.8 states that a redex and its contractum are contextually equivalent after erasure, with the key lemma showing that let-insertion preserves evaluation order. Here we rely on the soundness of the logical relation

(Theorem 5.5) to derive contextual equivalence. From this single-step result, multi-step preservation follows. Combined with transitivity of contextual equivalence, we can prove that evaluation to a code value preserves semantics (Theorem 5.10). Figure 1 illustrates the proof structure.

Semantics Preservation of Single-Step Reduction. We now turn to the main lemmas. They show that substitution (for ordinary let and β -reduction) and let-insertion both preserve evaluation order after erasure; the remaining reduction cases are routine.

We discuss the substitution case for let $x = v$ in t (ST-LET); the application case is analogous. For let-insertion, the term $E[\text{reflect } t]$ steps under a P -context to let_c $x = t$ in $E[\text{code } x]$. This preserves evaluation order after erasure because $|E|$ remains an evaluation context, since E -contexts are restricted to *pure* frames. We define erasure on contexts $|E|$ directly from term erasure. By contrast, reification contexts do not in general erase to evaluation contexts: for example, $\lambda_c x. \square$ erases to $\lambda x. \square$, but the erased language does not evaluate under λ -binders.

LEMMA 5.6 (SEMANTICS PRESERVATION OF SUBSTITUTION). *Substitution preserves evaluation order: if $\Gamma \vdash \text{let } x = v \text{ in } t : \tau \mid \epsilon$, then $|\Gamma| \models |\text{let } x = v \text{ in } t| \simeq^{ctx} |t[v/x]| : |\tau|$.*

LEMMA 5.7 (SEMANTICS PRESERVATION OF LET-INSERTION). *Let-insertion preserves evaluation order: if $\Gamma \vdash E[\text{reflect } t] : \tau \mid \epsilon$, then $|\Gamma| \models |E[\text{reflect } t]| \simeq^{ctx} |\text{let}_c x = t \text{ in } E[\text{code } x]| : |\tau|$.*

With these lemmas, we prove that a single reduction step preserves semantics:

THEOREM 5.8 (SEMANTICS PRESERVATION OF SINGLE-STEP REDUCTION). *If $\Gamma \vdash t_1 : \tau \mid \epsilon$, and $t_1 \rightarrow t_2$, then $|\Gamma| \models |t_1| \simeq^{ctx} |t_2| : |\tau|$.*

Semantics Preservation of Staging. Finally, we present the central theorem: if a two-stage program t_1 is typed with rep τ , and t_1 generates a program code t_2 , then $|t_1|$ is contextually equivalent to t_2 . It suffices to show a stronger theorem relating an arbitrary redex and its contractum, which gives us a strengthened inductive hypothesis. The proof relies on the transitivity of contextual equivalence to extend single-step semantics preservation to multi-steps.

THEOREM 5.9 (STRENGTHENED SEMANTICS PRESERVATION). *If $\Gamma \vdash t_1 : \tau \mid \epsilon$ and $t_1 \rightarrow^* t_2$, then $|\Gamma| \models |t_1| \simeq^{ctx} |t_2| : |\tau|$.*

THEOREM 5.10 (SEMANTICS PRESERVATION). *If $\emptyset \vdash t_1 : \text{rep } \tau \mid \epsilon$ and $t_1 \rightarrow^* \text{code } t_2$, then $\emptyset \models |t_1| \simeq^{ctx} t_2 : \tau$.*

Remark. Our results provide a precise semantic account of let-insertion in staging and answer the question posed by Inoue and Taha [27, 28]: “When does erasing staging annotations preserve semantics?” For the CBV $\lambda_{|2|}$ -calculus, our answer is simple and definitive: erasing annotations in $\lambda_{|2|}$ always preserves semantics.

6 Extension to Mutable References

We next extend $\lambda_{|2|}$ with mutable references, yielding $\lambda_{|2|}^{\text{ref}}$, to study erasure soundness for stateful staged programs. We first characterize the interactions between staging and references that break semantic preservation. We then adopt a simple, practical design that confines store effects to the second stage while keeping the first stage store-pure. The type system enforces this separation while allowing generated code to allocate and update references, as in the staged power function from Section 2. This section presents the syntax, semantics, and metatheory of $\lambda_{|2|}^{\text{ref}}$; Section 7 discusses possible designs supporting first-stage store effects.

6.1 When Do Store Effects Break Erasure Soundness?

Suppose we extend $\lambda_{|2|}$ with first-order mutable references. Certain interactions between staging and references can then violate erasure soundness. Consider a function `inc` that captures a first-stage reference `r`, initialized to `0`, and increments it; the program ultimately reads `r`. The staged program appears on the left and its stage-erased counterpart on the right.

<pre>(* staged version *) let r = ref 0 in (* r 1st-stage ref *) let inc = lift (fun y -> r := !r+1; y) in lift (!r)</pre>	<pre>(* stage-erased version *) let r = ref 0 in let inc = (fun y -> r := !r+1; y) in !r</pre>
---	---

In the staged program, `r` is allocated in the first stage. When the function is lifted, however, the assignment `r := !r+1` is performed during first-stage evaluation, even though the function is never applied. As a result, the final expression `lift (!r)` residualizes the value `1`, and the generated code is essentially `let inc = (fun y -> y) in 1`. By contrast, in the stage-erased program, the body of `inc` remains under the λ and is never called. Therefore, the increment does not occur, and the final value of `r` remains `0`.

The discrepancy shows that lifting a function forces static effects in its body to occur during first-stage evaluation, whereas after erasure those same effects remain latent and are performed only when the function is applied. Because function bodies are not evaluation contexts after erasure, the two programs no longer exhibit the same effects, and erasure soundness breaks.

The same problem also arises with second-stage conditionals. Consider the following example, in which both branches access the same reference. The then-branch mutates the reference before reading it, whereas the else-branch reads it directly.

<pre>(* staged version *) let r = ref 0 in (* r 1st-stage ref *) if² (lift b) then { r := !r + 1; lift (!r) } else lift (!r)</pre>	<pre>(* stage-erased version *) let r = ref 0 in if b then { r := !r + 1; !r } else !r</pre>
---	--

In the stage-erased program, only one branch and its effects are evaluated. In the staged program, by contrast, residualizing the conditional requires reifying both branches, along with the first-stage effects performed while reifying them. As a result, the generated code may depend on the order in which the branches are reified. In our calculi, the then-branch is reified first. Thus, when the else-branch is reified, it observes that `r` has already been incremented to `1`. Consequently, the generated code is `if b then 1 else 1`, whereas the stage-erased program evaluates to either `1` or `0`.

Developing a principled solution that rules out such counterexamples is non-trivial and beyond the scope of this paper; see Section 7 for discussion. The examples above, however, suggest that residualizing the allocation of `r` to the second stage is benign for semantics preservation. We adopt this discipline in $\lambda_{|2|}^{\text{ref}}$: although it restricts certain use cases, it still permits meaningful manipulation of effectful code fragments, and serves as a stepping stone toward future expressive designs.

6.2 The Syntax, Semantics, and Erasure of $\lambda_{|2|}^{\text{ref}}$

This section describes the extensions that $\lambda_{|2|}^{\text{ref}}$ adds to $\lambda_{|2|}$. Figure 11 presents the new syntax, reduction semantics, and type system, with changed rules relative to $\lambda_{|2|}$ highlighted in red. Similar to $\lambda_{|2|}$, operations on mutable references are attached with a stage annotation (e.g., `allocs`). Locations ℓ are administrative terms and values, which do not appear in source terms. We restrict to first-order stores for examining interactions between staging and references, therefore store σ maps locations to numbers.

Dynamic Semantics. The reduction semantics with stores is mostly standard. We extend the reduction relation of $\lambda_{|2|}$ with a runtime store, $\langle \sigma, t \rangle \rightarrow \langle \sigma', t' \rangle$. Because the type system confines store effects to the second stage, first-stage evaluation always uses the empty store: $\langle \emptyset, t \rangle \rightarrow^* \langle \emptyset, v \rangle$. Nevertheless, when $v = \text{code } t'$, the generated code is then evaluated in the second stage with actual store effects, $\langle \emptyset, t' \rangle \rightarrow^* \langle \sigma, v' \rangle$, yielding the final value. In this way, first-stage evaluation may still manipulate effectful code fragments which later allocate or mutate the store.

Pure frames B are extended for effectful operations. As in $\lambda_{|2|}$, the new rules for effectful 2-terms leave the store unchanged and instead reflect second-stage effectful terms into corresponding first-stage terms. The new rules for effectful 1-terms have the standard form $\sigma, t \rightsquigarrow \sigma', t'$ (Figure 11), including ST-ALLOC^1 , ST-GET^1 , and ST-PUT^1 . These rules may occur during second-stage evaluation under any evaluation context M (ST-MUT). Finally, ST-PURE and ST-REFLECT are extended to thread the store through configurations without changing it.

Static Semantics. $\lambda_{|2|}^{\text{ref}}$ extends the types with mutable references, written $\text{ref } \tau$, whose well-formedness is standard: τ must be well formed at the same stage. The typing rules for second-stage effectful operations (alloc^2 , get^2 , and put^2) follow the existing second-stage rules of $\lambda_{|2|}$. However, to restrict store effects only to the second stage, the first-stage effectful operations (alloc^1 , get^1 , and put^1) are allowed only under second-stage typing judgments ($\vdash^2$), where they appear solely as generated code. The run operation must be restricted as well, as running code that contains store effects could introduce locations back into the first stage. Thus, rule T-RUN requires t to be store-effect free, *i.e.*, it contains no alloc^s , get^s , or put^s operations; the full definition of storeFree can be found in the extended version of this paper [56].

Although $\lambda_{|2|}^{\text{ref}}$ includes references and locations, we do not need store or location typing for the sake of proving syntactic type soundness of first-stage evaluation and semantics preservation. Since first-stage evaluation has no store effects, the rules in Figure 11 are sufficient to prove syntactic type soundness. For the logical relation, we instead use a stronger world model for annotation-erased terms and generated code, which likewise avoids syntactic store typing.

The erasure operation on new terms and types are straightforward and structurally defined, similar to Figure 8; they are omitted and can be found in the mechanization.

6.3 Type Soundness and Semantics Preservation for $\lambda_{|2|}^{\text{ref}}$

The $\lambda_{|2|}^{\text{ref}}$ -calculus enjoys all meta-theoretical properties as $\lambda_{|2|}$, including syntactic type soundness, syntactic erasure soundness, and semantics preservation. Syntactic type soundness is standard, as the type system ensures that first-stage evaluation is store-free.

THEOREM 6.1 (PROGRESS). *If $\vdash t : \tau \mid \epsilon$, then t is a value or $\exists t' \text{ s.t. } \langle \emptyset, t \rangle \rightarrow \langle \emptyset, t' \rangle$.*

THEOREM 6.2 (PRESERVATION). *If $\vdash t : \tau \mid \epsilon$ and $\langle \emptyset, t \rangle \rightarrow \langle \emptyset, t' \rangle$, then $\exists \epsilon' \sqsubseteq \epsilon \text{ s.t. } \emptyset \vdash t' : \tau \mid \epsilon'$.*

THEOREM 6.3 (SYNTACTIC ERASURE SOUNDNESS). *If $\Gamma \vdash t : \tau \mid \epsilon$, then $|\Gamma| \vdash^2 |t| : |\tau| \mid \perp$.*

Below we outline the extension of the logical relation to relate stores via a world model, and then sketch our approach to semantics preservation.

The World Model. Unlike $\lambda_{|2|}$ that has no store and uses step-indexed logical relations alone, $\lambda_{|2|}^{\text{ref}}$ must relate stores across two runs. We introduce worlds $W \subseteq \mathbb{N} \times \mathbb{N}$ to relate locations in two stores (Figure 12). Because $\lambda_{|2|}^{\text{ref}}$ restricts stores to natural numbers, there are no cyclic references, thus the world need not be recursively indexed by types and steps. A world W is a partial bijection on locations (each location paired with at most one counterpart). We write $(\ell_1, \ell_2) \in W$ when

Syntax

Terms t	$::= \dots \mid \text{alloc}^s t \mid \text{get}^s t \mid \text{put}^s t_1 t_2$	Locations ℓ	$\in \mathbb{N}$
Administrative Terms g	$::= \dots \mid \ell$	Values v	$::= \dots \mid \ell$
Store σ	$::= \emptyset \mid \sigma[\ell \mapsto n]$	Types τ	$::= \dots \mid \text{ref } \tau$
Pure Frames B	$::= \dots \mid \text{alloc}^s \square \mid \text{get}^s \square \mid \text{put}^s \square t \mid \text{put}^s v \square$		

Dynamic Semantics

		$t \rightsquigarrow t'$	$\sigma, t \rightsquigarrow \sigma', t'$	$\langle \sigma, t \rangle \rightarrow \langle \sigma', t' \rangle$
(ST-ALLOC ²)	$\text{alloc}^2(\text{code } t)$	$\rightsquigarrow$	$\text{reflect}(\text{alloc}^1 t)$	
(ST-GET ²)	$\text{get}^2(\text{code } t)$	$\rightsquigarrow$	$\text{reflect}(\text{get}^1 t)$	
(ST-PUT ²)	$\text{put}^2(\text{code } t_1)(\text{code } t_2)$	$\rightsquigarrow$	$\text{reflect}(\text{put}^1 t_1 t_2)$	
(ST-ALLOC ¹)	$\sigma, \text{alloc}^1 n$	$\rightsquigarrow$	$\sigma[\ell \mapsto n], \ell$	where $\ell \notin \text{DOM}(\sigma)$
(ST-GET ¹)	$\sigma, \text{get}^1 \ell$	$\rightsquigarrow$	σ, n	where $\sigma(\ell) = n$
(ST-PUT ¹)	$\sigma, \text{put}^1 \ell n$	$\rightsquigarrow$	$\sigma[\ell \mapsto n], ()$	
	$t \rightsquigarrow t'$	(ST-PURE)	$\sigma, t \rightsquigarrow \sigma', t'$	(ST-MUT)
	$\langle \sigma, M[t] \rangle \rightarrow \langle \sigma, M[t'] \rangle$		$\langle \sigma, M[t] \rangle \rightarrow \langle \sigma', M[t'] \rangle$	
	x is fresh	(ST-REFLECT)	$\langle \sigma, P[E[\text{reflect } t]] \rangle \rightarrow \langle \sigma, P[\text{let}_c x = t \text{ in } E[\text{code } x]] \rangle$	

Static Semantics

$\frac{\text{WF}^s \tau}{\text{WF}^s(\text{ref } \tau)}$	(WF-REF)	$\frac{\text{storeFree } t \quad \emptyset \vdash t : \text{rep } \tau \mid \epsilon}{\Gamma \vdash^1 \text{run } t : \tau \mid \perp}$	(T-RUN)
$\frac{\Gamma \vdash^2 t : \text{nat} \mid \epsilon}{\Gamma \vdash^2 \text{alloc}^1 t : \text{ref nat} \mid \epsilon}$	(T-ALLOC ¹)	$\frac{\Gamma \vdash^1 t : \text{frag nat} \mid \epsilon}{\Gamma \vdash^1 \text{alloc}^2 t : \text{frag}(\text{ref nat}) \mid \top}$	(T-ALLOC ²)
$\frac{\Gamma \vdash^2 t : \text{ref nat} \mid \epsilon}{\Gamma \vdash^2 \text{get}^1 t : \text{nat} \mid \epsilon}$	(T-GET ¹)	$\frac{\Gamma \vdash^1 t : \text{frag}(\text{ref nat}) \mid \epsilon}{\Gamma \vdash^1 \text{get}^2 t : \text{frag nat} \mid \top}$	(T-GET ²)
$\frac{\Gamma \vdash^2 t_1 : \text{ref nat} \mid \epsilon_1 \quad \Gamma \vdash^2 t_2 : \text{nat} \mid \epsilon_2}{\Gamma \vdash^2 \text{put}^1 t_1 t_2 : \text{unit} \mid (\epsilon_1 \sqcup \epsilon_2)}$	(T-PUT ¹)	$\frac{\Gamma \vdash^1 t_1 : \text{frag}(\text{ref nat}) \mid \epsilon_1 \quad \Gamma \vdash^1 t_2 : \text{frag nat} \mid \epsilon_2}{\Gamma \vdash^1 \text{put}^2 t_1 t_2 : \text{frag unit} \mid \top}$	(T-PUT ²)

Fig. 11. The syntax and semantics of $\lambda_{|2|}^{\text{ref}}$. Only extended formalizations on top of $\lambda_{|2|}$ are included.

locations are related, and $(\sigma_1, \sigma_2) : W$ when both stores agree on all related locations. Worlds are ordered by extension: $W' \sqsupseteq W$ means that W' adds new related locations while preserving relations in W .

Semantics Preservation. The logical relation is extended to relate stores via the world model (Figure 12). Value, term, and environment interpretations are indexed by both the step k and the world W . For example, $(k, W, t_1, t_2) \in \mathcal{E}[\tau]$ relates two terms under the initial stores $(\sigma_1, \sigma_2) : W$, and if in k reduction steps we obtain values v_1, v_2 , then the values are related in the extended world $W' \sqsupseteq W$. Logical approximation $\preceq^{\text{log}}$ is defined as in $\lambda_{|2|}$, but with quantification over all worlds W . The observational contexts and their typing rules are likewise extended with frames for the

$$\begin{aligned}
W' \sqsupseteq W &\stackrel{\text{def}}{=} \forall (\ell_1, \ell_2) \in W. (\ell_1, \ell_2) \in W' \\
(j, W') \sqsupseteq (k, W) &\stackrel{\text{def}}{=} j \leq k \wedge W' \sqsupseteq W \\
(\sigma_1, \sigma_2) : W &\stackrel{\text{def}}{=} \forall (\ell_1, \ell_2) \in W. \sigma_1(\ell_1) = \sigma_2(\ell_2) \\
\mathcal{V}[\text{unit}] &= \{(k, W, (), ())\} \\
\mathcal{V}[\text{nat}] &= \{(k, W, n, n)\} \\
\mathcal{V}[\text{ref nat}] &= \{(k, W, \ell_1, \ell_2) \mid (\ell_1, \ell_2) \in W\} \\
\mathcal{V}[\tau_1 \rightarrow^\perp \tau_2] &= \{(k, W, \lambda x.t_1, \lambda x.t_2) \mid \forall (j, W') \sqsupseteq (k, W). \forall v_1, v_2. (j, W', v_1, v_2) \in \mathcal{V}[\tau_1] \Rightarrow \\
&\quad (j, W', \text{app}^\perp(\lambda x.t_1) v_1, \text{app}^\perp(\lambda x.t_2) v_2) \in \mathcal{E}[\tau_2]\} \\
\mathcal{E}[\tau] &= \{(k, W, t_1, t_2) \mid \forall j < k. \forall \sigma_1, \sigma_2, \sigma'_1, v_1. (\sigma_1, \sigma_2) : W \wedge \langle \sigma_1, t_1 \rangle \rightarrow^j \langle \sigma'_1, v_1 \rangle \Rightarrow \\
&\quad \exists \sigma'_2, v_2, W'. \langle \sigma_2, t_2 \rangle \rightarrow^* \langle \sigma'_2, v_2 \rangle \wedge (k - j, W') \sqsupseteq (k, W) \wedge \\
&\quad (\sigma'_1, \sigma'_2) : W' \wedge (k - j, W', v_1, v_2) \in \mathcal{V}[\tau]\} \\
\mathcal{G}[\emptyset] &= \{(k, W, \emptyset, \emptyset)\} \\
\mathcal{G}[\Gamma, x^2 : \tau] &= \{(k, W, \gamma_1[x \mapsto v_1], \gamma_2[x \mapsto v_2]) \mid (k, W, \gamma_1, \gamma_2) \in \mathcal{G}[\Gamma] \wedge (k, W, v_1, v_2) \in \mathcal{V}[\tau]\} \\
\Gamma \models t_1 \leq^{\log} t_2 : \tau &\stackrel{\text{def}}{=} \Gamma \vdash^2 t_1 : \tau \mid \perp \wedge \Gamma \vdash^2 t_2 : \tau \mid \perp \wedge \\
&\quad \forall k \geq 0. \forall \gamma_1, \gamma_2, W. (k, W, \gamma_1, \gamma_2) \in \mathcal{G}[\Gamma] \Rightarrow (k, W, \gamma_1(t_1), \gamma_2(t_2)) \in \mathcal{E}[\tau] \\
\Gamma \models t_1 \simeq^{\log} t_2 : \tau &\stackrel{\text{def}}{=} \Gamma \models t_1 \leq^{\log} t_2 : \tau \wedge \Gamma \models t_2 \leq^{\log} t_1 : \tau
\end{aligned}$$

Fig. 12. The world model, binary logical relations, and logical equivalence definitions for $\lambda_{|2|}^{\text{ref}\downarrow}$.

Observational Context for Contextual Equivalence

$$\text{Frame } F ::= \dots \mid \text{alloc}^\perp \square \mid \text{get}^\perp \square \mid \text{put}^\perp \square t_2 \mid \text{put}^\perp t_1 \square$$

Context Typing Rules

$$F : (\Gamma \vdash \tau) \Rightarrow (\Gamma' \vdash \tau')$$

$$\begin{array}{c}
\frac{}{\text{alloc}^\perp \square : (\Gamma \vdash \text{nat}) \Rightarrow (\Gamma \vdash \text{ref nat})} \quad \frac{}{\text{get}^\perp \square : (\Gamma \vdash \text{ref nat}) \Rightarrow (\Gamma \vdash \text{nat})} \\
\text{(F-ALLOC)} \qquad \text{(F-GET)} \\
\\
\frac{\Gamma \vdash^2 t_2 : \text{nat} \mid \perp}{\text{put}^\perp \square t_2 : (\Gamma \vdash \text{ref nat}) \Rightarrow (\Gamma \vdash \text{unit})} \quad \frac{\Gamma \vdash^2 t_1 : \text{ref nat} \mid \perp}{\text{put}^\perp t_1 \square : (\Gamma \vdash \text{nat}) \Rightarrow (\Gamma \vdash \text{unit})} \\
\text{(F-PUT-1)} \qquad \text{(F-PUT-2)}
\end{array}$$

Fig. 13. Additional context typing rules for $\lambda_{|2|}^{\text{ref}\downarrow}$. Only extended formalizations on top of $\lambda_{|2|}^\perp$ are included.

effectful operations in Figure 13, yielding the contextual equivalence $\simeq^{\text{ctx}}$ used in Theorem 6.4. We also prove the fundamental property and soundness of the logical relation.

Finally, we prove semantics preservation with the new step-indexed Kripke logical relation:

THEOREM 6.4 (SEMANTICS PRESERVATION). *If $\vdash t_1 : \text{rep } \tau \mid \epsilon$ and $\langle \emptyset, t_1 \rangle \rightarrow^* \langle \emptyset, \text{code } t_2 \rangle$, then $\emptyset \models |t_1| \simeq^{\text{ctx}} t_2 : \tau$.*

Remarks. With the $\lambda_{|2|}^{\text{ref}}$ -calculus, we show that it is possible to design a staging calculus with mutable references that enjoys erasure soundness. We again provide a simple and definitive answer to Inoue and Taha [27, 28]’s question: erasing annotations in $\lambda_{|2|}^{\text{ref}}$ always preserves semantics.

7 Open Questions and Future Work

We have presented $\lambda_{|2|}$ and $\lambda_{|2|}^{\text{ref}}$, along with their type soundness and semantic preservation results. A central open question is whether one can design an expressive multi-stage language with flexible

store effects while still preserving semantics. As discussed in Section 6.1, the main obstacle is that λ_c -terms and 2nd-stage conditionals cease to be evaluation contexts after erasure, so two-stage evaluation may produce store effects absent from the stage-erased program.

Still, it is not entirely hopeless. Whereas $\lambda_{|2|}^{\text{ref}}$ enforces *syntactic purity* at the first stage, a more flexible approach could enforce *semantic purity* [5, 47], admitting store effects that provably do not interfere with staging. For example, lifting a λ -term whose local store is fully encapsulated can preserve contextual equivalence. One promising direction is to adapt Rust-style resource-tracking type systems [6, 12, 38, 60, 69] to ensure that the store reachable during first-stage evaluation is disjoint from that accessed by λ_c -terms. Such separation could make effects semantically invariant under reordering, duplication, or omission. Region-based binding-time analyses in partial evaluation offer a related precedent: Dussart and Thiemann [20] use region information to approximate disjoint portions of the store, suggesting a basis for identifying non-interfering effects.

For conditionals, one could similarly require the locations reachable from the two branches to be disjoint, so that the order would not matter. Another possible approach, at the cost of code duplication, is polyvariant specialization: branches together with their shared continuations are specialized under different stores or more fine-grained stack-like structure of stores [20].

More than a decade ago, Inoue and Taha [27] identified erasure soundness for imperative MSP as an open challenge, observing that “imperative MSP is not yet ready.” Recent advances in resource-tracking type systems bring flexible, semantics-preserving imperative MSP within reach. Our results provide a foundation for exploring this rich design space.

Extensions. We discuss several possible extensions to $\lambda_{|2|}^{\text{ref}}$ that are left for future work.

- $\lambda_{|2|}^{\text{ref}}$ models second-stage first-order mutable store for simplicity. Extending it to support *second-stage higher-order store* should be straightforward following existing logical relations and world models for higher-order store [1, 2].
- The two-stage setting covers many practical applications and provides a foundation for N -stage programs ($N > 2$). Our binary logical relation could serve as the final-stage component, relating the last stage to its generated code. Semantic typing [58] and logical relations for code types offer promising paths toward this generalization.
- $\lambda_{|2|}$ ’s context decomposition $P[E[\text{reflect } t]]$ determines a unique, semantically sound position for inserting let-bindings. This design represents one point in a broader space of code-motion strategies: pure computations need not always be let-inserted, and impure computations may be moved as well as long as they preserve semantics. A finer-grained effect system that tracks dependencies [13] could enable more flexible insertion strategies.

8 Mechanization in Lean

One of our main contributions is the mechanization of the $\lambda_{|2|}/\lambda_{|2|}^{\text{ref}}$ calculi and their metatheory in Lean (v. 4.29.0). All results are machine-checked. Table 1 summarizes the structure and effort of our development, comprising about 10k lines for $\lambda_{|2|}$ and 12k lines for $\lambda_{|2|}^{\text{ref}}$.

Locally Nameless with de Bruijn Levels. Variables are represented in the locally nameless style [14]: free variables use de Bruijn levels and bound variables use de Bruijn indices. Using de Bruijn indices for bound variables simplifies the generation of fresh variables in let-insertion and avoids α -equivalence. We use de Bruijn levels rather than explicit names for free variables, which simplifies the introduction of new free variables during typing.

Level-Indexed Reduction and Contexts. Because reification contexts introduce second-stage bindings and can be plugged into terms in which second-stage variables occur free, our reduction

Table 1. Summary of the mechanization in Lean.

Component	Description	LoC of $\lambda_{ 2 }$	LoC of $\lambda_{ 2 }^{\text{ref}}$
Syntax	Syntax and syntactic operations using locally nameless.	2k	2.5k
OperationalSemantics	Static semantics and Determinism.	2k	2.5k
SyntacticTyping	Dynamic semantics and Syntactic Erasure Soundness.	1k	1.1k
SyntacticSoundness	Preservation and Progress.	1.5k	1.7k
CtxEquiv	Contextual equivalence.	0.2k	0.3k
LogicalEquiv	Logical relations and the Fundamental theorem.	1.9k	2k
SemanticsPreservation	Single-step and Multi-step Semantic Preservation.	1.4k	1.7k
Total LoC		10k	11.8k

relation is indexed by the current de Bruijn level. Evaluation contexts also track the current de Bruijn level, and their levels inform the reduction relation. For example, $\lambda_c x. \square$ increases the current level by one when entering its body. At the top level, the well-formed levels of reduction and evaluation contexts are 0.

Deterministic Memory Allocation. Without loss of generality, we simplify the design of our store in the formalization by modeling it as a stack rather than a heap. Our locations start at 0 and increment continuously with each allocation. This simplification still ensures that the properties we aim to discuss remain sufficiently strong.

9 Related Work

General Staged Programming. MetaML-style staging languages [33, 52, 53, 64] provide syntactic annotations that can change evaluation strategies. LMS [43, 45], by contrast, uses host-language operator overloading and does not rely on explicit syntax for staging. To model LMS-style staging that involves let-insertion, [Amin and Rompf](#) [3] introduced the untyped multi-stage $\lambda_{\uparrow\downarrow}$ -calculus with automatic let-insertion. Building on this foundation, we specialize the calculus to two stages, develop a sound type-and-effect system, extend it with mutable references, and prove semantic preservation under erasure. Our results formalize the conditions under which let-insertion preserves semantics and clarify its interaction with effects. This semantic guarantee also provides a foundation for stage-polymorphic programming [3, 26, 40], enabling programmers to abstract over stages and write stage-generic code that behaves uniformly across staging choices.

Let-Insertion in Staging and Specialization. Let-insertion has long supported specialization of effectful programs in partial evaluation [15, 20, 25, 29, 37]. Modern staging systems realize let-insertion in several ways. For example, BER MetaOCaml provides the explicit `genLet` primitive [35], implemented using delimited control [17, 32]. Similarly, [Kameyama et al.](#) [30, 31] combined delimited control with code generation and let-insertion. LMS performs let-insertion automatically through its dependency-tracking graph IR [13, 43, 45]. Squid also also performs A-normal form transformation for object code [22, 41, 42].

Our calculi, like $\lambda_{\uparrow\downarrow}$ [3], model let-insertion through small-step reduction. [Wei et al.](#) [63] instead present several big-step definitional interpreters using states, one-CPS, two-CPS, or control operators, together with an abstract machine for the same flavor of let-insertion. $\lambda_{|2|}$ and $\lambda_{|2|}^{\text{ref}}$ require no explicit delimited-control operators; their type-and-effect systems instead track let-insertion as a control effect. MetaOCaml similarly distinguishes general code fragments from object-code variables produced by `genLetv`. Our mechanized metatheory formalizes this distinction and establishes its role in type and erasure soundness.

In principle, MetaML-style staging can simulate behavior similar to our calculi by inserting `genLet` ubiquitously. However, rather than permitting unconstrained code-generation behavior as in MetaML, we advocate semantics-preserving staging as a more disciplined and predictable

default. Programmers can always opt out explicitly when necessary, much like `unsafe` in Rust. This design makes potentially semantics-breaking behavior visible and intentional, instead of leaving it implicit. In practice, staging libraries such as LMS take this approach as well, relying on the host-language Scala’s by-name arguments to support code duplication.

Reasoning Multi-Stage Programs and Partial Evaluation. In partial evaluation, Jones et al. [29] established the correctness of LambdaMix which supports general recursion, and Filinski [21] proved TDPE sound and complete for a pure call-by-name λ -calculus. Lawall and Thiemann [37] proved on-the-fly let-insertion sound, but did not consider effects performed during specialization, as discussed in Section 6.1.

Yang [71] was the first to propose the *erasure soundness* property, and regarded it as a criterion of “semantic correctness” for code generation, complementing the “syntactic correctness” guarantee of well-typed generated code. He proved erasure soundness for a CBN two-level language without mutable state, and identified duplication of effects as the key reason why a naive CBV two-level language fails to preserve semantics. Although let-insertion was proposed as a potential solution, a formal treatment or proofs were not provided in his work.

Inoue and Taha [27, 28] shared the same concern about the correctness of staging and developed equational theories and applicative bisimulation for the λ^U -calculus. They established termination conditions for erasure soundness: under CBV, it requires both the staged program and its stage-erased counterpart to terminate at first-order constants. In their setting, the calculus permits divergence but has no store effects, and their results posit stronger purity assumptions. Both Yang [71] and our work instead assume that first-stage evaluation terminates, ensuring that object code exists, and then relate that code to the erased program in the unstaged base language. In contrast, Inoue and Taha [27, 28] studied the equational theory of the staged language itself.

Independently and concurrently with this work, Wong et al. [65] mechanized an LMS-like calculus and proved a similar semantics-preservation theorem in Agda. Although the top-level theorems are similar, there are several key differences: First, we use a small-step reduction semantics, whereas Wong et al. [65] follows the big-step stateful semantics of Rompf [43]. Second, our calculi model both divergence and store effects, whereas the calculus of Wong et al. [65] requires strong normalization and does not include store effects. Finally, the proof techniques differ: we use binary Kripke logical relations that do not directly model code types, whereas Wong et al. [65] directly incorporates the equivalence between a two-stage term and its generated code into the value-type relation for code types.

Berger and Tratt [9] developed a Hoare logic for reasoning about Mini-ML_e[□]. In a dependently typed setting, Kovács [36] developed two-level type theory (2LTT), where staging is given by the evaluation of 2LTT syntax in a semantic domain. He also proved the conservativity of 2LTT.

10 Conclusion

In this paper, we study the question posed by Inoue and Taha [27]: when do staging annotations preserve the semantics of the underlying single-stage program? To this end, we design the $\lambda_{|2|}$ - and $\lambda_{|2|}^{\text{ref}}$ -calculi that support automatic let-insertion and include a lightweight type-and-effect system for tracking code-generation effects. Our main results establish syntactic type soundness via progress and preservation, and semantics preservation via step-indexed Kripke logical relations. We believe that the design of these calculi, together with their mechanized semantics-preservation proofs, fills a long-standing gap in reasoning about effectful multi-stage programs using modern logical-relation techniques. We hope this work serves as a solid foundation for future developments in expressive, reliable, and semantics-preserving multi-stage languages.

Data-Availability Statement

Our artifact includes the mechanization of the $\lambda_{|2|}$ -calculus and the $\lambda_{|2|}^{\text{ref}}$ -calculus, as well as examples and their meta-theoretic results in Lean 4. All theorems in the paper are mechanically proved in the artifact. The artifact is available at <https://doi.org/10.5281/zenodo.21392703> [55] and <https://github.com/instar-lang/mechanization/tree/oopsla26>.

Acknowledgments

We thank the reviewers of OOPSLA 2026 and PLDI 2026 for their detailed and constructive comments. We thank Tiark Rompf, Cameron Wong, and Peter Thiemann for helpful discussions.

References

- [1] Amal Ahmed, Derek Dreyer, and Andreas Rossberg. 2009. State-dependent representation independence. In *POPL*. ACM, 340–353. doi:10.1145/1480881.1480925
- [2] Amal Jamil Ahmed. 2004. *Semantics of types for mutable state*. Ph.D. Dissertation. Princeton University.
- [3] Nada Amin and Tiark Rompf. 2018. Collapsing towers of interpreters. *Proc. ACM Program. Lang.* 2, POPL (2018), 52:1–52:33. doi:10.1145/3158140
- [4] Michael Ballantyne, Raffaello Sanna, Jason Hemann, William E. Byrd, and Nada Amin. 2025. Multi-stage Relational Programming. *Proc. ACM Program. Lang.* 9, PLDI (2025), 1591–1615. doi:10.1145/3729314
- [5] Yuyan Bao and Tiark Rompf. 2025. Type, Ability, and Effect Systems: Perspectives on Purity, Semantics, and Expressiveness. *CoRR* abs/2510.07582 (2025).
- [6] Yuyan Bao, Guannan Wei, Oliver Bracevac, Yuxuan Jiang, Qiyang He, and Tiark Rompf. 2021. Reachability types: tracking aliasing and separation in higher-order functional programs. *Proc. ACM Program. Lang.* 5, OOPSLA (2021), 1–32. doi:10.1145/3485516
- [7] Alan Bawden. 1999. Quasiquotation in Lisp. In *PEPM*. University of Aarhus, 4–12.
- [8] Nick Benton, Andrew Kennedy, Lennart Beringer, and Martin Hofmann. 2007. Relational semantics for effect-based program transformations with dynamic allocation. In *PPDP*. ACM, 87–96. doi:10.1145/1273920.1273932
- [9] Martin Berger and Laurence Tratt. 2015. Program Logics for Homogeneous Generative Run-Time Meta-Programming. *Logical Methods in Computer Science* Volume 11, Issue 1, Article 5 (Mar 2015). doi:10.2168/LMCS-11(1:5)2015
- [10] Ulrich Berger, Matthias Eberl, and Helmut Schwichtenberg. 1998. Normalisation by Evaluation. In *Prospects for Hardware Foundations (Lecture Notes in Computer Science, Vol. 1546)*. Springer, 117–137. doi:10.1007/3-540-49254-2_4
- [11] Ulrich Berger and Helmut Schwichtenberg. 1991. An Inverse of the Evaluation Functional for Typed lambda-calculus. In *LICS*. IEEE Computer Society, 203–211. doi:10.1109/lics.1991.151645
- [12] Aleksander Boruch-Gruszecki, Martin Odersky, Edward Lee, Ondrej Lhoták, and Jonathan Immanuel Brachthäuser. 2023. Capturing Types. *ACM Trans. Program. Lang. Syst.* 45, 4 (2023), 21:1–21:52. doi:10.1145/3618003
- [13] Oliver Bracevac, Guannan Wei, Songlin Jia, Supun Abeysinghe, Yuxuan Jiang, Yuyan Bao, and Tiark Rompf. 2023. Graph IRs for Impure Higher-Order Languages: Making Aggressive Optimizations Affordable with Precise Effect Dependencies. *Proc. ACM Program. Lang.* 7, OOPSLA2 (2023), 400–430. doi:10.1145/3622813
- [14] Arthur Charguéraud. 2012. The Locally Nameless Representation. *J. Autom. Reason.* 49, 3 (2012), 363–408. doi:10.1007/s10817-011-9225-2
- [15] Olivier Danvy. 1996. Type-Directed Partial Evaluation. In *POPL*. ACM Press, 242–257. doi:10.1145/237721.237784
- [16] Olivier Danvy. 2006. An analytical approach to program as data objects.
- [17] Olivier Danvy and Andrzej Filinski. 1990. Abstracting Control. In *LISP and Functional Programming*. ACM, 151–160. doi:10.1145/91556.91622
- [18] Olivier Danvy, Karoline Malmkjær, and Jens Palsberg. 1996. Eta-Expansion Does The Trick. *ACM Trans. Program. Lang. Syst.* 18, 6 (1996), 730–751. doi:10.1145/236114.236119
- [19] Leonardo de Moura and Sebastian Ullrich. 2021. The Lean 4 Theorem Prover and Programming Language. In *CADE (Lecture Notes in Computer Science, Vol. 12699)*. Springer, 625–635. doi:10.1007/978-3-030-79876-5_37
- [20] Dirk Dussart and Peter Thiemann. 1997. Partial evaluation for higher-order languages with state. *Berichte des Wilhelm-Schickard-Instituts WSI-97-XX, Universit at Tübingen* (1997).
- [21] Andrzej Filinski. 1999. A Semantic Account of Type-Directed Partial Evaluation. In *PPDP (Lecture Notes in Computer Science, Vol. 1702)*. Springer, 378–395. doi:10.1007/10704567_23
- [22] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. 1993. The Essence of Compiling with Continuations. In *PLDI*. ACM, 237–247. doi:10.1145/155090.155113

- [23] Yoshihiko Futamura. 1971. Partial evaluation of computation process-an approach to a compiler-compiler. *Systems, computers, controls* 2, 5 (1971), 45–50.
- [24] John Hatcliff and Olivier Danvy. 1994. A Generic Account of Continuation-Passing Styles. In *POPL*. ACM Press, 458–471. doi:10.1145/174675.178053
- [25] John Hatcliff and Olivier Danvy. 1997. A Computational Formalization for Partial Evaluation. *Math. Struct. Comput. Sci.* 7, 5 (1997), 507–541. doi:10.1017/s0960129597002405
- [26] Fritz Henglein and Christian Mossin. 1994. Polymorphic Binding-Time Analysis. In *ESOP (Lecture Notes in Computer Science, Vol. 788)*. Springer, 287–301. doi:10.1007/3-540-57880-3_19
- [27] Jun Inoue and Walid Taha. 2012. Reasoning about Multi-stage Programs. In *ESOP (Lecture Notes in Computer Science, Vol. 7211)*. Springer, 357–376. doi:10.1007/978-3-642-28869-2_18
- [28] Jun Inoue and Walid Taha. 2016. Reasoning about multi-stage programs. *J. Funct. Program.* 26 (2016), e22. doi:10.1017/s0956796816000253
- [29] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. 1993. *Partial evaluation and automatic program generation*. Prentice Hall.
- [30] Yuki Yoshi Kameyama, Oleg Kiselyov, and Chung-chieh Shan. 2009. Shifting the stage: staging with delimited control. In *PEPM*. ACM, 111–120. doi:10.1145/1480945.1480962
- [31] Yuki Yoshi Kameyama, Oleg Kiselyov, and Chung-chieh Shan. 2011. Shifting the stage - Staging with delimited control. *J. Funct. Program.* 21, 6 (2011), 617–662. doi:10.1017/s0956796811000256
- [32] Oleg Kiselyov. 2012. Delimited control in OCaml, abstractly and concretely. *Theor. Comput. Sci.* 435 (2012), 56–76. doi:10.1016/j.tcs.2012.02.025
- [33] Oleg Kiselyov. 2014. The Design and Implementation of BER MetaOCaml - System Description. In *FLOPS (Lecture Notes in Computer Science, Vol. 8475)*. Springer, 86–102. doi:10.1007/978-3-319-07151-0_6
- [34] Oleg Kiselyov. 2018. Reconciling Abstraction with High Performance: A MetaOCaml approach. *Found. Trends Program. Lang.* 5, 1 (2018), 1–101. doi:10.1561/25000000038
- [35] Oleg Kiselyov. 2026. MetaOCaml: ten years later - System description. *Sci. Comput. Program.* 250 (2026), 103397. doi:10.1016/j.scico.2025.103397
- [36] András Kovács. 2022. Staged compilation with two-level type theory. *Proc. ACM Program. Lang.* 6, ICFP (2022), 540–569. doi:10.1145/3547641
- [37] Julia L. Lawall and Peter Thiemann. 1997. Sound Specialization in the Presence of Computational Effects. In *TACS (Lecture Notes in Computer Science, Vol. 1281)*. Springer, 165–190. doi:10.1007/bfb0014551
- [38] Danielle Marshall, Michael Vollmer, and Dominic Orchard. 2022. Linearity and Uniqueness: An Entente Cordiale. In *ESOP (Lecture Notes in Computer Science, Vol. 13240)*. Springer, 346–375. doi:10.1007/978-3-030-99336-8_13
- [39] Flemming Nielson and Hanne Riis Nielson. 1992. *Two-level functional languages*. Cambridge tracts in theoretical computer science, Vol. 34. Cambridge University Press. doi:10.1017/cbo9780511526572
- [40] Georg Offenbeck, Tiark Rompf, and Markus Püschel. 2017. Staging for generic programming in space and time. In *Proceedings of the 16th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE 2017, Vancouver, BC, Canada, October 23-24, 2017*, Matthew Flatt and Sebastian Erdweg (Eds.). ACM, 15–28. doi:10.1145/3136040.3136060
- [41] Lionel Parreaux, Amir Shaikhha, and Christoph E. Koch. 2017. Squid: type-safe, hygienic, and reusable quasiquotes. In *SCALA@SPLASH*. ACM, 56–66. doi:10.1145/3136000.3136005
- [42] Lionel Parreaux, Antoine Voizard, Amir Shaikhha, and Christoph E. Koch. 2018. Unifying analytic and statically-typed quasiquotes. *Proc. ACM Program. Lang.* 2, POPL (2018), 13:1–13:33. doi:10.1145/3158101
- [43] Tiark Rompf. 2016. The Essence of Multi-stage Evaluation in LMS. In *A List of Successes That Can Change the World (Lecture Notes in Computer Science, Vol. 9600)*. Springer, 318–335. doi:10.1007/978-3-319-30936-1_17
- [44] Tiark Rompf. 2016. Reflections on LMS: exploring front-end alternatives. In *SCALA@SPLASH*. ACM, 41–50. doi:10.1145/2998392.2998399
- [45] Tiark Rompf and Martin Odersky. 2010. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. In *GPCE*. ACM, 127–136. doi:10.1145/1868294.1868314
- [46] Tiark Rompf and Martin Odersky. 2012. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. *Commun. ACM* 55, 6 (2012), 121–130. doi:10.1145/2184319.2184345
- [47] Amr Sabry. 1998. What is a Purely Functional Language? *J. Funct. Program.* 8, 1 (1998), 1–22. doi:10.1017/s0956796897002943
- [48] Tim Sheard and Simon L. Peyton Jones. 2002. Template meta-programming for Haskell. *ACM SIGPLAN Notices* 37, 12 (2002), 60–75. doi:10.1145/636517.636528
- [49] Nicolas Stucki, Aggelos Biboudis, and Martin Odersky. 2018. A practical unification of multi-stage programming and macros. In *GPCE*. ACM, 14–27. doi:10.1145/3278122.3278139

- [50] Nicolas Stucki, Jonathan Immanuel Brachthäuser, and Martin Odersky. 2021. Multi-stage programming with generative and analytical macros. In *GPCE*. ACM, 110–122. doi:10.1145/3486609.3487203
- [51] Don Syme. 2020. The early history of F#. *Proc. ACM Program. Lang.* 4, HOPL (2020), 75:1–75:58. doi:10.1145/3386325
- [52] Walid Taha and Tim Sheard. 1997. Multi-Stage Programming with Explicit Annotations. In *PEPM*. ACM, 203–217. doi:10.1145/258993.259019
- [53] Walid Mohamed Taha. 1999. *Multistage programming: its theory and applications*. Oregon Graduate Institute of Science and Technology.
- [54] Ruby Y. Tahboub, Grégory M. Essertel, and Tiark Rompf. 2018. How to Architect a Query Compiler, Revisited. In *SIGMOD Conference*. ACM, 307–322. doi:10.1145/3183713.3196893
- [55] Jun Tan and Guannan Wei. 2026. *Artifact for When Do Staging Annotations Preserve Semantics? Mechanizing Typed Semantics-Preserving Multi-Stage Programming with Let-Insertion*. doi:10.5281/zenodo.21392703
- [56] Jun Tan and Guannan Wei. 2026. When Do Staging Annotations Preserve Semantics? Mechanizing Typed Semantics-Preserving Multi-Stage Programming with Let-Insertion (Extended Version). arXiv:2606.30854 [cs.PL] <https://arxiv.org/abs/2606.30854>
- [57] Jacob Thamsborg and Lars Birkedal. 2011. A kripke logical relation for effect-based program transformations. In *ICFP*. ACM, 445–456. doi:10.1145/2034773.2034831
- [58] Amin Timany, Robbert Krebbers, Derek Dreyer, and Lars Birkedal. 2024. A Logical Approach to Type Soundness. *J. ACM* 71, 6 (2024), 40:1–40:75. doi:10.1145/3676954
- [59] Fei Wang, James M. Decker, Xilun Wu, Grégory M. Essertel, and Tiark Rompf. 2018. Backpropagation with Callbacks: Foundations for Efficient and Expressive Differentiable Programming. In *NeurIPS*. 10201–10212.
- [60] Guannan Wei, Oliver Bracevac, Songlin Jia, Yuyan Bao, and Tiark Rompf. 2024. Polymorphic Reachability Types: Tracking Freshness, Aliasing, and Separation in Higher-Order Generic Programs. *Proc. ACM Program. Lang.* 8, POPL (2024), 393–424. doi:10.1145/3632856
- [61] Guannan Wei, Yuxuan Chen, and Tiark Rompf. 2019. Staged abstract interpreters: fast and modular whole-program analysis via meta-programming. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 126:1–126:32. doi:10.1145/3360552
- [62] Guannan Wei, Songlin Jia, Ruiqi Gao, Haotian Deng, Shangyin Tan, Oliver Bracevac, and Tiark Rompf. 2023. Compiling Parallel Symbolic Execution with Continuations. In *ICSE*. IEEE, 1316–1328. doi:10.1109/icse48619.2023.00116
- [63] Guannan Wei, Jun Tan, and Dinghong Zhong. 2026. Let It Be Optimized: Building Multi-stage Evaluators with Let-Insertion and Optimizations in Small Pieces (Functional Pearl). *Proc. ACM Program. Lang.* 10, ICFP, Article 278 (Aug. 2026), 31 pages. doi:10.1145/3828676
- [64] Edwin M. Westbrook, Mathias Ricken, Jun Inoue, Yilong Yao, Tamer Abdelatif, and Walid Taha. 2010. Mint: Java multi-stage programming using weak separability. In *PLDI*. ACM, 400–411. doi:10.1145/1809028.1806642
- [65] Cameron Wong, Stephen Chong, and Nada Amin. 2026. agda-staging: An Agda mechanization of a formally verified type-directed staging system. <https://github.com/CT075/agda-staging>. GitHub repository. Accessed: 2026-06-28.
- [66] Andrew K. Wright and Matthias Felleisen. 1994. A Syntactic Approach to Type Soundness. *Inf. Comput.* 115, 1 (1994), 38–94. doi:10.1006/inco.1994.1093
- [67] Ningning Xie, Matthew Pickering, Andres Löb, Nicolas Wu, Jeremy Yallop, and Meng Wang. 2022. Staging with class: a specification for typed template Haskell. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–30. doi:10.1145/3498723
- [68] Ningning Xie, Leo White, Olivier Nicole, and Jeremy Yallop. 2023. MacoCaml: Staging Composable and Compilable Macros. *Proc. ACM Program. Lang.* 7, ICFP (2023), 604–648. doi:10.1145/3607851
- [69] Yichen Xu, Aleksander Boruch-Gruszecki, and Martin Odersky. 2024. Degrees of Separation: A Flexible Type System for Safe Concurrency. *Proc. ACM Program. Lang.* 8, OOPSLA1 (2024), 1181–1207. doi:10.1145/3649853
- [70] Hiroto Yaguchi and Yukiyo Kameyama. 2025. Staged Gradual Typing. In *GPCE*. ACM, 94–104. doi:10.1145/3742876.3742880
- [71] Zhe Yang. 2000. Reasoning about code-generation in two-level languages. *BRICS Report Series* 7, 46 (2000). doi:10.7146/brics.v7i46.20213

Received 2026-03-17; accepted 2026-06-10