# Multi-Stage Programming with Splice Variables

ANONYMOUS AUTHOR(S)

Multi-stage programming is a popular approach to typed meta-programming, reducing abstraction overhead and producing performant programs. However, the traditional quote-and-splice staging syntax, as introduced by Davies 1996, can introduce complexities in managing expression evaluation, and also often necessitates sophisticated mechanisms for advanced features such as code pattern matching. This paper introduces $\lambda^{\bigcirc\triangleright}$, a novel staging calculus featuring *let-splice bindings*, a construct that explicitly binds splice expressions to *splice variables*, providing flexibility in managing, sharing, and reusing splice computations. Inspired by *contextual modal type theory*, our type system associates types with a typing context to capture *variables dependencies* of splice variables. We demonstrate that this mechanism seamlessly scales to features like code pattern matching, by formalizing $\lambda^{\bigcirc\triangleright}_{\text{pat}}$, an extension of $\lambda^{\bigcirc\triangleright}$ with code pattern matching and rewriting. We establish the syntactic type soundness of both calculi. Furthermore, we define a denotational semantics using a Kripke-style model, and prove adequacy results. All proofs have been fully mechanized using the Agda proof assistant.

## 1 Introduction

Typed meta-programming allows programs to generate and manipulate code fragments, reducing abstraction overhead and producing performant programs. A prominent approach to typed meta-programming is *multi-stage programming* based on temporal logic [Davies 1996], which has been implemented in several languages, including OCaml [Kiselyov 2014; Xie et al. 2023], Scala [Stucki et al. 2018, 2021], and Haskell [Sheard and Peyton Jones 2002; Xie et al. 2022], and has been successfully applied to improve performance in various domains [Jonnalagedda et al. 2014; Kiselyov et al. 2017; Wang et al. 2019; Willis et al. 2020; Yallop 2017]. Multi-stage programming integrates type and scope checking of code expressions into the type system, allowing meta-programs to be type-checked at compile time, leading to safer and more predictable code generation.

A classic example of staged programming is the power function. We present the example below using the *quoting* and *splicing* mechanisms, where a quotation `<..>` represents the code fragment of an expression, and a splice `$(..)` extracts out the expression from its code fragment:

```
let rec power x n = if n == 0 then <1>
                    else <$(x) * $(power x (n - 1))>
```

Given a quoted expression `<e>` and a known integer `n`, this function generates an expression with $n$ repeated multiplications, avoiding recursion at runtime and thus reducing overhead for any specific `e`. For example, with `n` being `5`, we have:

```
<fun x → $(power <x> 5)>    -- generates <fun x → x * x * x * x * x * 1>
```

While the quote-and-splice syntax offers a useful staging mechanism, it also introduces several complexities. Specifically, managing expression evaluation is subtle and often involved a *level-index reduction relation* [Calcagno et al. 2003; Taha and Sheard 2000], complicating both implementations and formal semantics. Moreover, supporting pattern matching on code often requires sophisticated mechanisms [Jang et al. 2022; Parreaux et al. 2017; Stucki et al. 2018].

In this paper, we introduce *let-splice bindings*, a language construct that explicitly binds splice expressions to *splice variables*. Unlike the splice operator, let-splice bindings offer precise control over splice evaluation order, drawing inspiration from the target calculus of Typed Template Haskell [Xie et al. 2022]. Extending Xie et al. [2022], our let-splice bindings provide greater flexibility and enable the sharing and reuse of splice computations. Our design includes a novel type system that tracks the *variable dependencies* of splice variables, allowing splice expressions to be defined in contexts where certain variables are not yet available. Inspired by *contextual modal type theory* [Jang

et al. 2022; Nanevski et al. 2008], this type system associates types with a typing context to capture these variable dependencies, ensuring well-typed splice definitions. When a splice variable is used, its associated dependencies must be provided. These contexts can be nested to specify intricate dependency structures. Furthermore, we demonstrate that our design naturally supports advanced meta-programming features such as *unhygienic functions* [Barzilay et al. 2011], *code pattern matching* [Stucki et al. 2021], and *code rewriting* [Parreaux et al. 2017], using the same mechanism for managing variable dependencies, highlighting the expressiveness of our approach.

We present two calculi: (1) $\lambda^{\bigcirc\triangleright}$, a temporal-style multi-stage calculus supporting let-splice bindings, featuring a novel contextual modality ($\Delta \triangleright$) for managing variable dependencies; and (2) $\lambda^{\bigcirc\triangleright}_{\text{pat}}$, an extension of $\lambda^{\bigcirc\triangleright}$ which integrates code pattern matching and code rewriting. Both calculi are equipped with a small-step operational semantics, and enjoy syntactic type soundness. Moreover, we define a denotational semantics based on a Kripke-style model for both calculi, and establish the adequacy results for $\lambda^{\bigcirc\triangleright}$. The calculi along with all proofs are fully formalized in the Agda proof assistant. In particular, we offer the following contributions:

- §3 and 4 present $\lambda^{\bigcirc\triangleright}$, a staging calculus featuring let-splice bindings with splice variables, dependency tracking with nested typing context, a temporal-style code type for code expressions, and a separate contextual modality for managing variable dependencies.
- §5 proves soundness and completeness of $\lambda^{\bigcirc\triangleright}$'s type system with respect to *constructive linear-time temporal logic* [Kojima and Igarashi 2011], and then provides a type-preserving translation from $\lambda^{\bigcirc\triangleright}$ to $\lambda^{\bigcirc}$ [Davies 1996], providing insight into their relationships.
- §6 introduces $\lambda^{\bigcirc\triangleright}_{\text{pat}}$, an extension of $\lambda^{\bigcirc\triangleright}$ that enables pattern matching on code and code rewriting, allowing for inspection and manipulation of code expressions.
- §7 defines a denotational semantics for $\lambda^{\bigcirc\triangleright}$ and $\lambda^{\bigcirc\triangleright}_{\text{pat}}$ using a Kripke-style model, establishing termination of the operational semantics and adequacy of the denotational semantics.
- We formalize $\lambda^{\bigcirc\triangleright}$ and $\lambda^{\bigcirc\triangleright}_{\text{pat}}$ in the Agda proof assistant (§8), and establish key properties and theorems including progress, preservation, and adequacy. The Agda definitions and proofs of all stated lemmas and theorems are provided in the supplementary materials.

Lastly, §9 compares our approach to related work. For space reasons, some rules of our calculi are elided in the paper; the full judgments and the complete set of rules are provided in the appendix.

## 2 Examples

This section outlines the design of our calculus and illustrates its expressiveness through examples. We use OCaml-like syntax throughout this section.

### 2.1 Staged Power Function

Recall the staged power function from the introduction:

```
val power : int¹ code → int⁰ → int¹ code
let rec power x n = if n == 0 then <1>
                    else <$(x) * $(power x (n - 1))>
```

Here, a *quotation* `<expr>` represents the code fragment of the expression, and a *splice* `$(expr)` extracts out the expression from the code fragment. The type constructor `code` annotates typed code expressions. Formally, the type system associates each variable and expression with a *level* (i.e. "time" in temporal logic). Intuitively, levels indicate the evaluation stage of expressions. Quotations increase the level of the quoted expression, splices decrease the level, and local variables have the same level as their surrounding context and can only be used at that level. For clarity, we annotate base types with their levels.

While the quote-and-splice syntax is useful, it also introduces several complexities. Consider evaluating `(e1 <e2 $(e3)>)`. Here, `e1` and `e3` are evaluated, but `e2` is not, as `e2` represents an expression in the next stage. Therefore, evaluating a staged program necessitates a *level-indexed reduction relation* [Calcagno et al. 2003; Taha and Sheard 2000] to track an evaluation level, which is dynamically updated by quotations and splices during evaluation. This adds complexity to both implementations and formal semantics. Moreover, it is difficult to precisely control the evaluation of splice expressions (see also §2.2). Such fine-grain control is also needed for *nested splices* in compile-time code generation systems [Sheard and Peyton Jones 2002; Xie et al. 2022, 2023], where splices without surrounding quotations are evaluated at compile-time. For instance, in `$(e1) $($(e2))`, the relative evaluation order of `e1` and `e2` is unclear: while `e1` appears first, `e2` has more splices. Recently, Xie et al. [2022] addresses this by prioritizing more deeply nested splices through an elaboration process. However, the resulting target calculus is not directly accessible to programmers.

In this paper, we present *let-splice bindings*, a new construct that defines *splice variables*. This approach explicitly defines splice expressions and thus specify their evaluation order, eliminating the need for a level-indexed reduction relation. For example, the staged power function can be written as follows using our syntax:

```
val power : int¹ code → int⁰ → int¹ code
let rec power x n = if n == 0 then <1>
                    else let$ s1 : int¹ = x in                  -- lifted
                         let$ s2 : int¹ = power x (n - 1) in    -- lifted
                         <s1 * s2>
```

Here, let-splice bindings, defined using `let$`, replace the splice operation. There definitions bind a code expression to a splice variable. In this case, the splice variables `s1` and `s2` represent the splices of `x` and `power x (n - 1)`, respectively. Since these variables represent splices in the original program, they can be used directly within the quotaiton, as in `<s1 * s2>`. Formally, let-splice bindings unwraps a `code`-typed expression, introducing a variable which is one level higher and thus can be used within quotations. Let-splice bindings are evaluated sequentially; in this case, `s1` is evaluated before `s2`, making the evaluation order of splice expressions explicit.

More interestingly, let-splice bindings can be annotated with a list of *variable dependencies*. This provides flexibility for splice expressions which depend on values that are only available when the splice variable is used. For instance, we can define `power5` as:

```
let$ power5 : (x : int¹ ⊢ int¹)  -- variable dependency on x
   = power <x> 5                 -- generates <x * x * x * x * x * 1>
```

As `(power <x> 5)` refers to the variable `x`, the splice variable `power5` is declared with a dependency on `x : int¹`, allowing `x` to be used within its definition. Our type system tracks splice variables' dependencies, in addition to their levels. In this case, `power5` has type `(x : int¹ ⊢ int¹)`, indicating that it is a level-1 splice with type `int¹` that depends on a variable `x : int¹`.

To use a splice variable with variable dependencies like `power5`, the syntax `(power5 with x = e)` provides a *delayed substitution*. This allows variable dependencies to be replaced with specific values when a splice variable is actually used:

```
let thirty_two = <power5 with x = 2>  -- <2 * 2 * 2 * 2 * 2 * 1>
```

The right-hand side of the substitution can be any expression or variable in general. For brevity, we often write `s with x` as a shorthand for `s with x = x`. For example:

```
let fpower = <fun x → power5 with x>  -- <fun x → x * x * x * x * x * 1>
```

In practice, a compiler may automatically capture dependencies from the context, allowing identical entries like `with x = x` to be omitted entirely.

## 2.2 Reuse of Splice Variables

We demonstrate that let-splice bindings and splice variables offer finer-grained control over splice expression evaluation. In particular, consider below a program using the quote-and-splice syntax, which generates a pair of functions that apply a function `f` to its argument, and increment and decrement the result, respectively:

```
<(fun x → $(f <x>) + 1, fun y → $(f <y>) - 1)>  -- quote and splice
```

For example, given `f z = (let w = very_long_computation in <$(z) * 2>)`, this generates

```
<(fun x → x * 2 + 1, fun y → y * 2 - 1)>
```

However, the program evaluates the two splices `$(f <x>)` and `$(f <y>)` sequentially, leading to duplicated computations of `very_long_computation`.

To avoid duplicated computations, we may pre-compute the splice expression's result, bind it to a variable, and use that variable in the definition:

```
let s = <fun z → $(f <z>)> in
<(fun x → $(s) x + 1, fun y → $(s) y - 1)>
```

Unfortunately, the generated code is different:

```
<(fun x → ((fun z → z * 2) x) + 1, fun y → ((fun z → z * 2) y) - 1)>
```

introducing two additional beta-redexes, since `s` produces a function being applied to `x`. This may not be desirable, for example, when we can inspect the generated code using pattern matching (§2.5).

In our calculus, we can easily share splice computations using splice variables. Specifically, we can write the original computation as:

```
let$ s : (z : int¹ ⊢ int¹) = f <z> in  -- splice variable
<(fun x → (s with z = x) + 1, fun y → (s with z = y) - 1)>
```

Here, `let$` declares a splice variable `s` with a dependency on `z`. The expression `f <z>` is evaluated, which can refer to variable `z`. The `(s with z = x)` syntax directly substitutes `z` with `x`, and `(s with z = y)` substitutes `z` with `y`. In this case, the `very_long_computation` is evaluated only once, and the generated code is the desired `<(fun x → x * 2 + 1, fun y → y * 2 - 1)>`.

## 2.3 Variable Dependencies

Building upon variable dependencies, our system extends support for dependency declarations to standard let bindings and function arguments. As an example, consider the following program:

```
val w : (x : string¹; y : int¹ ⊢ int¹ code)
let w = <if y == 0 then "hello" else x>
```

This let binding declares a normal variable `w` which depends on variables `x` and `y`, and produces a value of type `int¹ code`. The following function `f` takes an argument of this type, substituting `x` with `"world"` and `y` with `42`:

```
val f : (x : string¹; y : int¹ ⊢ int¹ code) → int¹ code
let f z = (z with x = "world"; y = 42)
```

We can then apply `f` to `w`:

```
f w  -- <if 42 == 0 then "hello" else "world">
```

We refer to `w` as an *unhygienic value*, and `f` as an *unhygienic function*; we will revisit this in §2.4.

Moreover, we note that dependencies can also be nested, allowing dependencies themselves to depend on other dependencies. For example, we can define:

```
val z : (s : (x : int¹ ⊢ string¹) ⊢ string¹ code)
let z = <cat (s with x = 2) (s with x = 2 + 1)>
```

Here, `z` depends on `s`, which in turn has a dependency on `x`. When `z` is used, such as in `z with s = e`, `s` is replaced by `e`, with `e` potentially using `x`. This `x` is subsequently replaced by `2` and `2 + 1`, respectively. As a concrete example, we have:

```
(z with s = string_of_int x)  -- <cat (string_of_int 2) (string_of_int (2 + 1))>
```

Lastly, a type can express variable dependencies across multiple levels, and the corresponding term can take expressions from these levels, as in the following program:

```
val h : (x : bool¹; y : int² ⊢ string² code code)
let h = <if x then <string_of_int y> else <string_of_int (y + 1)>>
```

## 2.4 Unhygienic Functions

*Hygienic macros* [Kohlbecker et al. 1986] guarantee that macro expansion does not accidentally capture variables. While hygienic macros are well established, they can sometimes also be insufficient. Barzilay et al. [2011] observed common kinds of practically useful, yet *unhygienic*, macro patterns, particularly those which can implicitly introduces bindings. Two common examples include a looping macro (e.g. `while`) that implicitly binds a variable (e.g. `abort`) that can be used to escape the loop inside the loop body [Clinger 1991], and *anaphoric conditionals* which introduces a binding to hold the value of the tested expression. Barzilay et al. [2011] described these unhygienic macros as *"notoriously difficult to deal with"*.

In this paper, we use *unhygienic functions* to mean functions whose arguments may depend on additional, *later-stage* variables provided when the function is used. We show how our system supports unhygienic functions within this context. Considering anaphoric conditionals as an example, we would like to create a function `aif`, with which we can write the following program:

```
aif <big_long_calculation> <foo it ...> <bar it ...>
```

Here, both the then- and else-branches use the variable `it` to represent the result of `big_long_calculation`. The program will expand to:

```
<let it = big_long_calculation in
 if it then (foo it ...) else (bar it ...)>
```

In statically-typed languages, `it` will obviously stand for `True` in the then-branch and `False` in the else-branch. In dynamically-typed languages like Scheme, however, `it` in the else-branch is not necessarily `False`, making the function more practical. While we focus on anaphoric conditionals for their simplicity, the underlying principles extend to more complex examples like looping.

Using variable dependencies, we define `aif` with the following type signature. Note that the second and third arguments have an additional dependency on the variable `it`:

```
val aif : bool¹ code
      → (it : bool¹ ⊢ 'a¹ code) → (it : bool¹ ⊢ 'a¹ code) → 'a¹ code
```

When `aif` is applied, its type signature tells the type checker to introduce `it` into the scope of the second and third arguments, allowing them to directly use it.

Now that we have the type signature, we can implement `aif` as follows:

```
let aif cond foo bar =
  let$ s1 : bool¹ = cond in
  let$ s2 : (it : bool¹ ⊢ 'a¹) = (foo with it) in
  let$ s3 : (it : bool¹ ⊢ 'a¹) = (bar with it) in
  <let it = s1 in
   if it then (s2 with it) else (s3 with it)>
```

The `aif` function takes three code arguments: `cond`, `foo`, and `bar`, with the latter two depending on the variable `it`. The `let$` unwraps these code arguments and bind them to splice variables `s1`, `s2`, and `s3`, respectively. The quoted expression then constructs the resulting code. Notably, while code is generated for both branches, only the branch corresponding to the value of `it` will be evaluated at runtime. Supporting unhygienic functions shows the expressiveness of our calculus, enabling a broader range of meta-programming patterns, including those that intentionally manipulate lexical scoping in a type-safe manner.

### 2.5 Pattern Matching on Code

We have so far focused on *generative* meta-programming, where smaller code fragments are combined to construct larger ones, as illustrated by the `power` and `aif` examples. *Analytic* meta-programming, in contrast, can inspect or deconstruct code, enabling useful techniques like code rewriting (§2.6) for optimization. This is often realized through pattern matching on code in staging calculi [Jang et al. 2022; Parreaux et al. 2017; Stucki et al. 2021].

We extend our calculus with support for pattern matching on code, integrating it seamlessly through our mechanism of splice variables, showcasing the expressiveness of our design. As an example, we can write the following program that simply swaps two addends:

```
val swap : int¹ code ⟶ int¹ code
let swap n =  match$ n with | (x `+ y) ⟶ <y + x>
                            | _ ⟶ n
```

The `match$` construct performs pattern matching on code. Patterns describe the structure *within* the quotation, distinguishing between *pattern variables* (e.g. `x` and `y`) that match any code expression, and identifier literals like `` `+ `` and `` `* `` that match those exact identifiers. Intuitively, pattern variables behave similarly to splice variables, as they match inside quotations and may only be used within quotations. In this case, therefore, we have `x : int¹` and `y : int¹`. When matching succeeds, we return `<y + x>`. As a result, we have:

```
swap <1 + 2>    -- generates <2 + 1>
```

This demonstrates how our calculus naturally supports code pattern matching using a mechanism similar to splice variables. In fact, code pattern matching can be viewed as a generalization of let-splice bindings (§6), with additional rules for typing code patterns.

One key challenge in code pattern matching, however, is handling open code. Specifically, consider the following match expression:

```
match$ <fun x ⟶ x + 1> with | (fun x ⟶ y) ⟶ e
```

A natural result would be for `y` to match on `x + 1`. However, this results in `y` storing a code fragment that refers to the bound variable `x`. When `y` is used in `e`, `x` is no longer in the scope! To address this issue, Stucki et al. [2018] require writing the pattern as `fun x ⟶ $(y(` x))`, which binds `y` to a function `fun x ⟶ <$(x) + 1>` with type `int¹ code ⟶ int¹ code`. In other words, `y` has been eta-expanded to take the open variable (`x`) as an input. However, explicitly applying

pattern variables to free variables is inconvenient, and binding the pattern variable to a function may be undesirable. Moreover, Stucki et al. [2018] restrict patterns to the simply typed lambda calculus, as it remains unclear how the approach generalizes to quotes, splices, or matches themselves.[1]

In our calculus, just like pattern matching can be viewed as a generalization of let-splice bindings, pattern variables with open code work similarly to splice variables with variable dependencies. In the above example, our system binds `y` to `x + 1` with type `(x : int¹ ⊢ int¹)`. We can thus use `y` by providing specific `x`, e.g.

```
match$ <fun x → x + 1> with
  | (fun x → y) → <y with (x = 2)>  -- generates <2 + 1>
```

Here, crucially, `x` is not matched by name. Rather, the pattern matches any code expression with the function form, and we can refer to the bound variable as `x` within the function body. That is, pattern matching is *binding-aware*, and operates on the expression's binding structure.

Pattern variables with open code offer a useful technique for code manipulation. As a larger example, consider a program that computes the partial derivative of an arithmetic expression as a code fragment. The following `partial` function recursively matches its argument `e`, generating code for its partial derivative with respect to an variable `var`:

```
val (+) (*) : int¹ → int¹ → int¹
val partial : (var : int¹ ⊢ int¹ code → int¹ code)
let rec partial e = match$ e with
    | (`var) → <1>
    | (g `+ h) →
      let$ dg = (partial with var) <g> in
      let$ dh = (partial with var) <h> in <dg + dh>
    | (g `* h) →
      let$ dg = (partial with var) <g> in
      let$ dh = (partial with var) <h> in <g * dh + h * dg>
    | _ → <0>
```

We can apply `partial` by providing `var` and the expression. For example, the following program:

```
let df : (x : int¹, y : int¹ ⊢ int¹ code) = (partial with var = x) <x * y + 1>
```

generates `<(1 * y + x * 0) + 0>`. We then use `df` by providing specific `x` and `y`, such as `df with x = 1, y = 2`, which produces `<(1 * 2 + 1 * 0) + 0>`.

Using pattern variables with open code, we can perform pattern matching under a binder. Specifically, consider computing the partial derivative of a let expression `let (y : int) = f in g`. Using the chain rule, the derivative is computed as:

$$\partial_x\, g(x, f(x)) = \partial_x\, g(x, y)\,|_{y=f(x)} + \partial_y\, g(x, y)\,|_{y=f(x)} \cdot \partial_x\, f(x)$$

This can be implemented with an additional pattern matching case:

```
match$ e with
  | ...
  | (let (y : int¹) = f in g) →
    let$ df = (partial with var) <f> in
    let$ dg1 : (y : int¹ ⊢ int¹) = (partial with var) <g with y> in
```

[1] Contextual modal type systems [Jang et al. 2022] and similar calculi [Parreaux et al. 2017] inherently support open code. These systems behave quite differently, requiring code's type to explicitly specify its open variable (see §9).

```
let$ dg2 : (y : int¹ ⊢ int¹) = (partial with var = y) <g with y> in
<let (y : int¹) = f in (dg1 with y) + (dg2 with y) * df>
```

Again, `y` is not matched by name, and the pattern matches any code expression with the let form. In this case, `g` is matched as a splice variable with a variable dependency on `y`. Within the pattern case, `dg1` computes the derivative of `g` with respect to the given variable `var`, `dg2` computes the derivative of `g` with respect to `y`, and `df` computes the derivative of `f`. The final expression combines these derivatives according to the chain rule.

### 2.6 Code Rewriting

Lastly, we support *code rewriting*, a primitive that replaces instances of a pattern within a target expression with a replacement expression. This is particularly useful for optimizing generated code, which often contains redundancies. For example, the `df` example using the `partial` function above generates `<(1 * y + x * 0) + 0>`, where the operations `1 *`, `* 0`, and `+ 0` can be simplified.

Code rewriting is expressed as:

```
e_target rewrite p ⟶ e_replacement
```

where `p` is a code pattern, and `e_target` and `e_replacement` are code expressions. Similar to pattern matching, the pattern `p` can introduce pattern variables, potentially containing open code, which can be then used inside `e_replacement`.

As an example, the following program uses code rewriting to simplify the code generated by `df`:

```
let df_opt : (x : int¹, y : int¹ ⊢ int¹ code) =
  <(df with x; y)>
    rewrite (z `* `0) ⟶ <0>
    rewrite (z `+ `0) ⟶ <0>
    rewrite (`1 `* z) ⟶ <z>
```

Here, the first rewrite simplifies `z `* `0` to `<0>`, where `` `* `` and `` `0 `` match those exact identifiers, and `z` is a pattern variable that matches any code expression, which can be used within `<z>`. Thus, any code expression matching `z `* `0` is simplified to `0`. The other rewrites work similarly. The entire program simplifies the code `<(1 * y + x * 0) + 0>` to `<y>`.

As another example, we consider a program where find all local variables defined as pairs, and rewrite their projections into direct component accesses [Parreaux et al. 2017]. The following program shows how to use `rewrite` to traverse a term `pgm` and perform these rewrites:

```
pgm rewrite (let p : int¹ * int¹ = (a, b) in body) ⟶      -- a: int¹; b: int¹
  let$ body2 : (p : int¹ * int¹ ⊢ 't¹) =       -- body: (p: int¹ * int¹ ⊢ 't¹)
    <body with p>
      rewrite (`fst `p) ⟶ <a>
      rewrite (`snd `p) ⟶ <b>
  in <body2 with p = (a, b)>
```

The first `rewrite` identifies let bindings that define a variable `p` to be a pair of `a` and `b`. Here, `a` and `b` are pattern variables, and `body` is matched as a splice variable with a variable dependency on `p`. Within `body with p`, we rewrite every occurrence of `` `fst `p `` to `<a>`, and `` `snd `p `` to `<b>`, and bind its splice result to `body2`. The program then returns `<body2 with p = (a, b)>`, substituting `p` with `(a,b)` within `body2`. For example, given `<let y = (1+2, 3) in (fst y + snd y, y)>`, the body matches `<(fst y + snd y, y)>`. Then, `body2` binds the rewritten program `((1+2)+3, y)`, and then the substitution turns it into `<((1+2)+3, (1+2, 3))>`.

## 3 A Staging Calculus with Splice Variables

This section presents $\lambda^{\bigcirc\triangleright}$, a typed lambda calculus with staging, quotations, and let-splice bindings. We begin by introducing the syntax of types and expressions (§3.1-3.2), and then present the typing rules (§3.3). The dynamic semantics are introduced in the following section (§4).

### 3.1 Types and Typing Contexts

***Contexts.*** $\lambda^{\bigcirc\triangleright}$ uses typing contexts to track *variable dependencies*. These contexts enable unhygienic values and serve as the foundation for code pattern matching, which is introduced in §6. Contexts are defined by the following grammar:

$$\Gamma, \Delta \ ::= \ \cdot \mid \Gamma, x : [\Delta \vdash^n A]$$

Each variable $x$ within a context is associated with three components:

- A *context* $\Delta$, which tracks the variable dependencies of $x$. When $\Delta$ is empty, we write $x : A^n$ as shorthand for $x : [\cdot \vdash^n A]$.
- An integer *level* $n$ at which $x$ is bound. The level has the same meaning as in Davies [1996]: higher values correspond to later stages, while lower values correspond to earlier stages.
- A *type* $A$, which describes the type of $x$. As we will see, types themselves can contain further variable dependencies, thus allowing for nested variable dependencies.

The *restriction* of a context $\Gamma$ to level $n$, written $\Gamma\restriction_n$, removes all variables in $\Gamma$ with levels less than $n$. Importantly, restriction preserves *well-stagedness* (we will define well-stagedness shortly): if $\Gamma$ is well-staged at some level $n$, then $\Gamma\restriction_m$ is well-staged at level $m$.

***Types.*** Types are defined by the following grammar:

$$A, B \ ::= \ \mathbf{bool} \mid [\Delta \vdash A] \to B \mid \bigcirc A \mid \Delta \triangleright A$$

- **bool** represents booleans.
- $[\Delta \vdash A] \to B$ represents *unhygienic functions* from $A$ to $B$, where the argument may additionally depend on variables in $\Delta$. We write $A \to B$ as shorthand for $[\cdot \vdash A] \to B$.
- $\bigcirc A$ represents quoted expressions of type $A$, whose computations happen at the next stage.
- $\Delta \triangleright A$ represents *unhygienic values* of type $A$ with dependencies $\Delta$. This type allows any expressions to depend on later-stage variables, enabling unhygienic values to be used as first-class citizens. For example, we can express function types like $(\Delta \triangleright A) \to (\Delta' \triangleright B)$.

We note that while $[\Delta \vdash A] \to B$ can be expressed as $(\Delta \triangleright A) \to B$[2], we include the former in the language as it allows for more natural definitions of unhygienic functions, without requiring explicit wrapping and unwrapping (§3.2).

***Well-stagedness.*** We consider only *well-staged* contexts and types in our typing rules (§3.3).

$\boxed{\Gamma^n}$ *(Well-staged Contexts)* $\qquad$ $\boxed{A^n}$ *(Well-staged Types)*

$$\text{WS-Empty}\ \frac{}{(\cdot)^n} \qquad \text{WS-Nest}\ \frac{\Gamma^n \quad A^m \quad \Delta^m \quad m \geq n}{(\Gamma, x : [\Delta \vdash^m A])^n} \qquad \text{WS-Bool}\ \frac{}{\mathbf{bool}^n}$$

$$\text{WS-CtxArr}\ \frac{\Delta^{n+1} \quad A^n \quad B^n}{([\Delta \vdash A] \to B)^n} \qquad \text{WS-Code}\ \frac{A^{n+1}}{(\bigcirc A)^n} \qquad \text{WS-Wrap}\ \frac{\Delta^{n+1} \quad A^n}{(\Delta \triangleright A)^n}$$

Note that a context $\Gamma$ being well-staged at level $n$ requires each entry $x : [\Delta \vdash^m A]$ to have $m \geq n$. In other words, stage levels can only stay the same or increase as the nesting of [] becomes deeper. Similarly, for type $[\Delta \vdash A] \to B$ to be well-staged at level $n$, $\Delta$ must be well-staged at level $n + 1$.

[2]We have $wrapToArr : ((\Delta \triangleright A) \to B) \to ([\Delta \vdash A] \to B) \coloneqq \lambda f.\ \lambda_\Delta x : A.\ f\ (\mathbf{wrap}_\Delta x_{\mathrm{id}_\Delta})$; and $arrToWrap : ([\Delta \vdash A] \to B) \to ((\Delta \triangleright A) \to B) \coloneqq \lambda f.\ \lambda x : (\Delta \triangleright A).\ f\ (\mathbf{let}\ \mathbf{wrap}_\Delta\ y : A = x\ \mathbf{in}\ y_{\mathrm{id}_\Delta})$.

We often write $\Gamma^n$ and $A^n$ to indicate the levels of a context or a type. This notation binds more tightly than type constructors. For example, $[\Delta^{n+1} \vdash A^n] \to B^n$ is a well-staged type at level $n$.

### 3.2 Expressions

We define the syntax of expressions in $\lambda^{\bigcirc\triangleright}$ as follows.

$$
\begin{array}{rcll}
e & ::= & x_\sigma & \textit{(Variables)} \\
& | & \mathbf{true} \mid \mathbf{false} \mid \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 & \textit{(Booleans)} \\
& | & \lambda_\Delta x : A.\ e \mid e_1\ e_2 & \textit{(Functions)} \\
& | & \langle e \rangle \mid \mathbf{let}_\Delta \langle x : A \rangle = e_1\ \mathbf{in}\ e_2 & \textit{(Quote and Unquote)} \\
& | & \mathbf{wrap}_\Delta e \mid \mathbf{let\ wrap}_\Delta\ x : A = e_1\ \mathbf{in}\ e_2 & \textit{(Wrap and Unwrap)}
\end{array}
$$

$$
\begin{array}{rcll}
\sigma & ::= & \cdot & \textit{(Empty)} \\
& | & \sigma, x \mapsto y & \textit{(Renaming)} \\
& | & \sigma, x \mapsto e & \textit{(Substitution)}
\end{array}
$$

Within expressions, $x_\sigma$ represents a variable $x$ paired with a *delayed substitution* $\sigma$. This substitution maps dependencies of $x$ expressions in the current context. The substitution is applied when $x$ is substituted with a concrete expression (see §4.1 for the formal definition of substitution).

An unhygienic function $\lambda_\Delta x : A.\ e$ allows its argument $x$ to depend on variables in $\Delta$. An application $e_1\ e_2$ applies a function $e_1$ to an argument $e_2$. A quotation $\langle e \rangle$ quotes an expression $e$ into a code expression. Dually, a let-splice binding $\mathbf{let}_\Delta \langle x : A \rangle = e_1\ \mathbf{in}\ e_2$ unquotes a code expression $e_1$ that can depend on variables in $\Delta$, introducing a next-stage variable $x$ with dependencies $\Delta$, which can be used inside quotations within $e_2$. Similarly, $\mathbf{wrap}_\Delta e$ wraps an expression $e$ with dependencies $\Delta$, allowing $e$ to refer to variables in $\Delta$, while $\mathbf{let\ wrap}_\Delta\ x : A = e_1\ \mathbf{in}\ e_2$ unwraps a wrapped expression $e_1$ with dependencies $\Delta$, binding it to a current-stage variable $x$ in $e_2$.

Substitutions $\sigma$ can contain two kinds of entries: $x \mapsto y$ renames a dependency $x$ to another variable $y$, and $x \mapsto e$ maps a dependency $x$ to an expression $e$. We write $\mathrm{id}_\Gamma$ for the identity substitution that maps each variable in $\Gamma$ to the same variable, i.e. $x_1 \mapsto x_1, x_2 \mapsto x_2 \ldots$ for $x_i \in \Gamma$. Moreover, the notation $\sigma\restriction_n$ removes all entries in $\sigma$ with levels less than $n$, similar to context restriction. Lastly, for well-typed substitutions (§3.3), we write $x^m_\Delta \mapsto y$ or $x^m_\Delta \mapsto e$ to indicate the dependencies and stage level of a substitution entry.

*Mapping between concrete and abstract syntax.* As examples, (1) `x with y = e1; z = e2` is expressed as $x_{y \mapsto e_1, z \mapsto e_2}$; (2) `fun x : (Δ ⊢ A) → e` as $\lambda_\Delta x : A.\ e$; (3) `let x : (Δ ⊢ A) = e1 in e2` as $(\lambda_\Delta x : A.\ e_2)\ e_1$; (4) `let$ x : (Δ ⊢ A) = e1 in e2` as $\mathbf{let}_\Delta \langle x : A \rangle = e_1\ \mathbf{in}\ e_2$.

### 3.3 Typing Rules

The typing judgment $\Gamma \vdash^n e : A$ assigns a type $A$ to an expression $e$ under the context $\Gamma$, at stage level $n$. We assume that all contexts contain distinct variables, and both the context $\Gamma$ and the type $A$ are well-staged at level $n$. The judgment depends on another judgment, $\Gamma \vdash \sigma : \Gamma'$, which types a substitution $\sigma$ under context $\Gamma$, producing a context $\Gamma'$. Intuitively, the substitution maps variables in $\Gamma'$ to variables or expressions typed under $\Gamma$. Thus, we refer to $\Gamma'$ the domain of $\sigma$ and $\Gamma$ the codomain. The typing rules are defined in Fig. 1.

Rule VarSubst type-checks variables. A variable $x : [\Delta \vdash^n A]$ can only be used at level $n$, and must be provided a substitution $\sigma$ that maps each dependency variable in $\Delta$ to a variable or an expression with the corresponding type under $\Gamma$. If $\Delta$ is empty, as is the case for normal variables, then $\sigma$ is also empty. The next rules True, False, and If are standard.

Rule CtxAbs types an unhygienic function, which allows its argument to depend on $\Delta$. Notably, a dependency context $\Delta$ is always staged one level higher than its surrounding context. The rule puts $(x : [\Delta^{n+1} \vdash^n A])$ into the context to type-check the body. Rule CtxApp type-checks an application. If $e_1$ has type $[\Delta^{n+1} \vdash A] \to B$, the rule extends the context with $\Delta$ to type-check the argument $e_2$.

$\boxed{\Gamma \vdash^n e : A}$ *(Expression Typing)*

$$\textsc{VarSubst}\ \frac{\Gamma \ni x : [\Delta \vdash^n A] \qquad \Gamma \vdash \sigma : \Delta}{\Gamma \vdash^n x_\sigma : A} \qquad \textsc{True}\ \frac{}{\Gamma \vdash^n \mathbf{true} : \mathbf{bool}} \qquad \textsc{False}\ \frac{}{\Gamma \vdash^n \mathbf{false} : \mathbf{bool}}$$

$$\textsc{If}\ \frac{\Gamma \vdash^n e_1 : \mathbf{bool} \qquad \Gamma \vdash^n e_2 : A \qquad \Gamma \vdash^n e_3 : A}{\Gamma \vdash^n \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 : A} \qquad \textsc{CtxAbs}\ \frac{\Gamma, x : [\Delta^{n+1} \vdash^n A] \vdash^n e : B}{\Gamma \vdash^n \lambda_\Delta x : A.\ e : [\Delta \vdash A] \to B}$$

$$\textsc{CtxApp}\ \frac{\Gamma \vdash^n e_1 : [\Delta^{n+1} \vdash A] \to B \qquad \Gamma, \Delta \vdash^n e_2 : A}{\Gamma \vdash^n e_1\ e_2 : B} \qquad \textsc{Quote}\ \frac{\Gamma\restriction_{n+1} \vdash^{n+1} e : A}{\Gamma \vdash^n \langle e \rangle : \bigcirc A}$$

$$\textsc{LetQuote}\ \frac{\Gamma, \Delta^{n+1} \vdash^n e_1 : \bigcirc A \qquad \Gamma, x : [\Delta \vdash^{n+1} A] \vdash^n e_2 : B}{\Gamma \vdash^n \mathbf{let}_\Delta \langle x : A \rangle = e_1\ \mathbf{in}\ e_2 : B} \qquad \textsc{Wrap}\ \frac{\Gamma, \Delta^{n+1} \vdash^n e : A}{\Gamma \vdash^n \mathbf{wrap}_\Delta e : \Delta \rhd A}$$

$$\textsc{LetWrap}\ \frac{\Gamma \vdash^n e_1 : \Delta \rhd A \qquad \Gamma, x : [\Delta \vdash^n A] \vdash^n e_2 : B}{\Gamma \vdash^n \mathbf{let\,wrap}_\Delta\ x : A = e_1\ \mathbf{in}\ e_2 : B}$$

$\boxed{\Gamma \vdash \sigma : \Gamma'}$ *(Substitution Typing)*

$$\textsc{S-Empty}\ \frac{}{\Gamma \vdash \cdot : \cdot} \qquad \textsc{S-Rename}\ \frac{\Gamma \vdash \sigma : \Gamma' \qquad \Gamma \ni y : [\Delta \vdash^m A]}{\Gamma \vdash (\sigma, x \mapsto y) : \Gamma', x : [\Delta \vdash^m A]} \qquad \textsc{S-Subst}\ \frac{\Gamma \vdash \sigma : \Gamma' \qquad \Gamma\restriction_m, \Delta \vdash^m e : A}{\Gamma \vdash (\sigma, x \mapsto e) : \Gamma', x : [\Delta \vdash^m A]}$$

Fig. 1. $\lambda^{\bigcirc\rhd}$ typing rules

Rule Quote increases the level to $n+1$ and restricts the context to $\Gamma\restriction_{n+1}$ (§3.1) to type-check the quoted expression $e$. If $e$ has type $A$, then $\langle e \rangle$ has the code type $\bigcirc A$. Rule LetQuote unquotes a code expression $e_1$, and binds it to a variable $x$ at the next level. Note that $e_1$ is typed under the current context extended with $\Delta$, and thus $x$ represents an open code fragment that may additionally depend on variables in a context $\Delta$. The rule put $x$ in the context to type-check $e_2$.

To type-check $\Delta \rhd A$, rule Wrap extends the context with $\Delta$ to type-check $e$. Note that unlike rule Quote, rule Wrap does not change the stage level of the expression. Rule LetWrap unwraps a wrapped expression and binds it to a contextual variable $x : [\Delta \vdash^n A]$ when typing $e_2$.

For typing substitutions, rule S-Rename checks that renaming preserves the stage level and dependencies of a variable. Rule S-Subst checks that substitution maps an $m$-level variable $x$ to an $m$-level expression $e$, where $\Gamma$ is restricted to level $m$ and is then extended with $\Delta$ to type-check $e$. Given the typing rules, we have $\Gamma \vdash \mathrm{id}_\Gamma : \Gamma$ (§3.2). Also, if $\Gamma_2 \vdash \sigma : \Gamma_1$, then $\Gamma_2\restriction_n \vdash \sigma\restriction_n : \Gamma_1\restriction_n$.

## 4 Dynamics

This section presents a small-step, call-by-value operational semantics for $\lambda^{\bigcirc\rhd}$, based on substitution.

### 4.1 Substitution

We define a substitution function $e[\sigma]$, which applies a typed substitution $\sigma$ to a typed expression $e$, and $\sigma_1[\sigma]$, which applies $\sigma$ to another typed substitution $\sigma_1$. Since expressions and substitutions can contain each other, these functions are defined mutually. These functions perform *simultaneous substitution*, meaning that all open variables in the expression are replaced at once. As a consequence, the domain of $\sigma$ must match the context of the expression being substituted (see also Lemma 4.2).

The definitions as presented below, with $\Gamma_1 \vdash^n e : A$ and $\Gamma_2 \vdash \sigma : \Gamma_1$ (Lemma 4.2). For space reasons, we show a selection of rules; the complete set of rules can be found in Appendix A.

$\boxed{e[\sigma]}$ *(Expression Substitution)*

$$
\begin{aligned}
(x_{\sigma_1})[\sigma] &\coloneqq \begin{cases} y_{(\sigma_1[\sigma])} & \text{if } \sigma(x) = y, \\ e[\mathsf{id}_{\Gamma_2\restriction_m}, \sigma_1[\sigma]] & \text{if } \sigma(x) = e. \end{cases} \\
(\langle e \rangle)[\sigma] &\coloneqq \langle e[\sigma\restriction] \rangle \\
(\mathbf{let}_\Delta \langle x : A \rangle = e_1 \mathbf{\ in\ } e_2)[\sigma] &\coloneqq \mathbf{let}_\Delta \langle x : A \rangle = (e_1[\sigma, \mathsf{id}_\Delta]) \mathbf{\ in\ } (e_2[\sigma, x \mapsto x]) \\
(\mathbf{wrap}_\Delta e)[\sigma] &\coloneqq \mathbf{wrap}_\Delta (e[\sigma, \mathsf{id}_\Delta]) \\
(\mathbf{let\ wrap}_\Delta\ x : A = e_1 \mathbf{\ in\ } e_2)[\sigma] &\coloneqq \mathbf{let\ wrap}_\Delta\ x : A = (e_1[\sigma]) \mathbf{\ in\ } (e_2[\sigma, x \mapsto x])
\end{aligned}
$$

$\boxed{\sigma_1[\sigma]}$ *(Substitution on Substitutions)*

$$
\begin{aligned}
(\cdot)[\sigma] &\coloneqq \cdot & (\sigma_1, x \mapsto y)[\sigma] &\coloneqq \sigma_1[\sigma], x \mapsto \sigma(y) \\
& & (\sigma_1, x^m_\Delta \mapsto e)[\sigma] &\coloneqq \sigma_1[\sigma], x \mapsto e[\sigma\restriction_m, \mathsf{id}_\Delta]
\end{aligned}
$$

For expression substitutions $e[\sigma]$, most cases simply extend or restrict the substitution to match the contexts of the subexpressions and then recurse. The most interesting case is substitution on a variable, $(x_{\sigma_1})[\sigma]$. Intuitively, applying $\sigma$ to both $x$ and $\sigma_1$ should yield $\sigma(x)_{(\sigma_1[\sigma])}$. Thus, when $\sigma(x) = y$, the function simply returns $y_{(\sigma_1[\sigma])}$. When $\sigma(x) = e$, however, since only variables can have delayed substitutions, the function must further apply $\sigma_1[\sigma]$ to $e$. In other words, the delayed substitution is *applied* at this point. The termination of this process is less obvious and requires a subtle argument, which is discussed in §4.1.1. The extended identity substitutions are for proofs (Lemma 4.2); a practical implementation may omit them.

To substitute a substitution, $\sigma_1[\sigma]$ simply applies $\sigma$ to each entry of $\sigma_1$. If an entry is a variable $y$, it is replaced with the corresponding entry $\sigma(y)$. Otherwise, if the entry is an expression $e$, the function applies $\sigma$ to $e$, restricting and extending $\sigma$ to match the level and context of $e$.

*4.1.1* ***Termination.*** The substitution $e[\sigma]$ defined above is not structurally recursive on $e$, so its termination is not immediately obvious. The problematic case is the second clause of $(x_{\sigma_1})[\sigma]$:

$$(x_{\sigma_1})[\sigma] = e[\mathsf{id}_{\Gamma_2\restriction_m}, \sigma_1[\sigma]] \quad \text{if } \sigma(x) = e.$$

Here, the expression $e$ is not a sub-expression of $x_{\sigma_1}$, but rather an element of the substitution $\sigma$. Therefore, we cannot argue for termination based solely on the size of the input expression. A similar issue arises in Boespflug and Pientka [2011], where termination of single substitution is shown by demonstrating that the size of the input expression's context decreases.

In our case, since we work with simultaneous substitution, all expression entries within the substitution must be considered. To prove termination, we define the *depth* of a substitution $\sigma$ as the maximum context depth among all its expression entries:

$\boxed{\mathsf{depth}(\Gamma)}$ *(Context Depth)* $\boxed{\mathsf{depth}(\sigma)}$ *(Substitution Depth)*

$$
\begin{aligned}
\mathsf{depth}(\cdot) &\coloneqq 0 & \mathsf{depth}(\cdot) &\coloneqq 0 \\
\mathsf{depth}(\Gamma, x : [\Delta \vdash^m A]) &\coloneqq & \mathsf{depth}(\sigma, x \mapsto y) &\coloneqq \mathsf{depth}(\sigma) \\
\mathsf{depth}(\Gamma) \sqcup (\mathsf{depth}(\Delta) + 1) & & \mathsf{depth}(\sigma, x^m_\Delta \mapsto e) &\coloneqq \mathsf{depth}(\sigma) \sqcup (\mathsf{depth}(\Delta) + 1)
\end{aligned}
$$

Intuitively, the depth of a substitution bounds the number of times the problematic case can occur in a recursive call. To prove this, we show that the depth decreases in the problematic case and does not increase in other cases. Non-increase follows because extending with variable entries

does not change the depth, as they are not counted, and restriction does not increase the depth since the depth is defined as a maximum. In the problematic case, we need to show that

$$\mathsf{depth}(\mathsf{id}_{\Gamma_2\restriction_m}, \sigma_1[\sigma]) < \mathsf{depth}(\sigma).$$

Let $\Delta$ be the domain of $\sigma_1$. By induction, $\sigma_1[\sigma]$ is well-defined and $\Gamma_2 \vdash \sigma_1[\sigma] : \Delta$. Since adding variable entries does not change the depth, and since the depth of a substitution is at most the depth of its domain, we have $\mathsf{depth}(\mathsf{id}_{\Gamma_2\restriction_m}, \sigma_1[\sigma]) \leq \mathsf{depth}(\sigma_1[\sigma]) \leq \mathsf{depth}(\Delta)$. Since $x^m_\Delta \mapsto e \in \sigma$, it follows that $\mathsf{depth}(\Delta) < \mathsf{depth}(\sigma)$. Thus, we get $\mathsf{depth}(\mathsf{id}_{\Gamma_2\restriction_m}, \sigma_1[\sigma]) \leq \mathsf{depth}(\Delta) < \mathsf{depth}(\sigma)$ as required. Therefore, the substitution function terminates and is well-defined on all inputs.

*4.1.2* ***Substitution Properties.*** We prove that the substitution functions enjoy the usual associativity and commutativity properties. These lemmas are crucial when proving adequacy of the denotational semantics in §7.4, which employs a logical relations argument that famously relies on these properties [Abel et al. 2019; Acevedo and Weirich 2023].

Lemma 4.1 (Substitution Properties).

**(Identity)** *(1) If* $\Gamma_1 \vdash^n e : A$ *then* $e[\mathsf{id}_{\Gamma_1}] = e$. *(2) If* $\Gamma_1 \vdash \sigma_1 : \Delta$ *then* $\sigma_1[\mathsf{id}_{\Gamma_1}] = \sigma_1 = \mathsf{id}_\Delta[\sigma_1]$.
**(Associativity)** *(1) If* $\Gamma_1 \vdash^n e : A$ *and* $\Gamma_2 \vdash \sigma : \Gamma_1$ *and* $\Gamma_3 \vdash \sigma' : \Gamma_2$ *then* $e[\sigma][\sigma'] = e[\sigma[\sigma']]$.
*(2) If* $\Gamma_1 \vdash \sigma_1 : \Delta$ *and* $\Gamma_2 \vdash \sigma : \Gamma_1$ *and* $\Gamma_3 \vdash \sigma' : \Gamma_2$ *then* $\sigma_1[\sigma][\sigma'] = \sigma_1[\sigma[\sigma']]$.
**(Commutativity)** *If* $\Gamma, \Delta \vdash^n e : A$, $\Gamma' \vdash \sigma_1 : \Gamma$, *and* $\Gamma' \vdash \sigma_2 : \Delta$, *then* $e[\sigma_1, \sigma_2] = e[\sigma_1, \mathsf{id}_\Delta][\mathsf{id}_{\Gamma'}, \sigma_2]$.

Since substitution functions are not structurally recursive, some of these properties need to be proven by induction on depths. Associativity, for instance, is proven by induction on sum of the depths of both substitutions, i.e. $\mathsf{depth}(\sigma) + \mathsf{depth}(\sigma')$. The full proofs are available in Agda (§8).

Lastly, we show that substitution preserves typing:

Lemma 4.2 (Substitution Lemma). • *If* $\Gamma_1 \vdash^n e : A$ *and* $\Gamma_2 \vdash \sigma : \Gamma_1$ *then* $\Gamma_2 \vdash^n e[\sigma] : A$.
• *if* $\Gamma_1 \vdash \sigma_1 : \Delta$ *and* $\Gamma_2 \vdash \sigma : \Gamma_1$ *then* $\Gamma_2 \vdash \sigma_1[\sigma] : \Delta$.

## 4.2 Operational Semantics

We now present our reduction rules. First, values and evaluation contexts are defined below.

$$\begin{array}{lrl}
\text{Values} & v & ::= \mathbf{true} \mid \mathbf{false} \mid \lambda_\Delta x : A.\, e \mid \langle e \rangle \mid \mathbf{wrap}_\Delta\, v \\
\text{Evaluation Contexts} & E & ::= [\,] \mid E\, e_2 \mid v_1\, E \mid \mathbf{if}\, E\, \mathbf{then}\, e_2\, \mathbf{else}\, e_3 \mid \mathbf{let}_\Delta \langle x : A \rangle = E\, \mathbf{in}\, e_2 \\
& & \quad \mid \mathbf{wrap}_\Delta E \mid \mathbf{let}\, \mathbf{wrap}_\Delta\, x : A = E\, \mathbf{in}\, e_2
\end{array}$$

Evaluation contexts are expressions with a hole ([]) that can be filled with another expression, indicating the part of the expression being reduced. We write $E[e]$ for the expression obtained by filling the hole of $E$ with $e$. Note that unlike Calcagno et al. [2003]; Taha and Sheard [2000], our definition of values does not require level indexing.

The call-by-value reduction is defined in Fig. 2. We write $\longrightarrow_\beta$ for a step of reduction, and $\longrightarrow$ for evaluation under an evaluation context. Rule CtxAppAbs applies a lambda to an argument value by substituting the argument into the lambda body. Rule LetQuoteQuote unwaps a quoted expression $\langle e_1 \rangle$ and substitutes it into the body $e_2$; since any quoted expression is already a value, $e_1$ itself is not necessarily a value. Rule IfTrue and Rule IfFalse reduce an if-then-else expression to the appropriate branch. Rule LetWrapWrap unwraps a wrapped value $\mathbf{wrap}_\Delta\, v$ and substitutes it into the body $e_2$. Unlike Rule LetQuoteQuote, the wrapped value is first reduced to a value before substitution. Rule Cong reduces the subexpression of an evaluation context.

$\boxed{\Gamma \vdash^n e \longrightarrow_\beta e'}$ $\boxed{\Gamma \vdash^n e \longrightarrow e'}$ *(Call-by-value Reduction)*

$$\textsc{CtxAppAbs}\ \frac{}{\Gamma \vdash^n (\lambda_\Delta x : A.\ e_1)\ v_2 \longrightarrow_\beta e_1[\mathrm{id}_\Gamma, x \mapsto v_2]} \qquad \textsc{LetQuoteQuote}\ \frac{}{\Gamma \vdash^n \mathbf{let}_\Delta \langle x : A\rangle = \langle e_1\rangle \mathbf{\ in\ } e_2 \longrightarrow_\beta e_2[\mathrm{id}_\Gamma, x \mapsto e_1]}$$

$$\textsc{IfTrue}\ \frac{}{\Gamma \vdash^n \mathbf{if\ true\ then}\ e_2\ \mathbf{else}\ e_3 \longrightarrow_\beta e_2} \qquad \textsc{IfFalse}\ \frac{}{\Gamma \vdash^n \mathbf{if\ false\ then}\ e_2\ \mathbf{else}\ e_3 \longrightarrow_\beta e_3}$$

$$\textsc{LetWrapWrap}\ \frac{}{\Gamma \vdash^n \mathbf{let\ wrap}_\Delta\ x : A = \mathbf{wrap}_\Delta\ v_1\ \mathbf{in}\ e_2 \longrightarrow_\beta e_2[\mathrm{id}_\Gamma, x \mapsto v_1]} \qquad \textsc{Cong}\ \frac{\Gamma \vdash^n e_1 \longrightarrow_\beta e_2}{\Gamma' \vdash^n E[e_1] \longrightarrow E[e_2]}$$

Fig. 2. $\lambda^{\bigcirc\triangleright}$ operational semantics

*4.2.1* ***Type Soundness.*** We prove preservation, which a corollary of the substitution lemma (4.2):

Theorem 4.3 (Preservation). *If* $\Gamma \vdash^n e : A$ *and* $\Gamma \vdash^n e \longrightarrow e'$ *then* $\Gamma \vdash^n e' : A$.

Progress is subtler. Specifically, since we allow arbitrary nesting of dependencies, delayed substitutions are crucial for progress to hold in our calculus. For example, consider the program:

$$\mathbf{let}_{x:[z:\mathbf{bool} \vdash^1 \mathbf{bool}]} \langle y : \mathbf{bool}\rangle = (\mathbf{let}\ \langle z : \mathbf{bool}\rangle = \langle \mathbf{true}\rangle\ \mathbf{in}\ \langle x_{z \mapsto z}\rangle)\ \mathbf{in}\ \langle \mathbf{true}\rangle$$

Here, $y$ is declared with a dependency $x$, which in turn depends on $z$. To evaluate the inner let binding, we need to substitute $z$ with **true** within the let body. Without allowing delayed substitutions to contain arbitrary expressions (e.g. $x_{z \mapsto \mathbf{true}}$), this substitution would do nothing, and then the evaluation would be stuck. In contrast, the core calculus of Xie et al. [2022] does not allow nested dependencies. As a result, in their system, variables can only capture their variable dependencies from the context.

We prove that progress holds for expressions that do not contain variables at the current level. This reflects our definition of "unhygienic values": values in this calculus are not necessarily closed terms but may include variables from later stages.

Theorem 4.4 (Progress). *If* $\Gamma^{n+1} \vdash^n e : A$ *then either* $e$ *is a value or there exists* $e'$ *such that* $\Gamma^{n+1} \vdash^n e \longrightarrow e'$.

## 4.3 Example

We demonstrate the operational semantics of the calculus with a larger example. For improved readability, we present the example using the concrete syntax; recall the mapping between the concrete syntax and the abstract syntax provided in §3.2. Consider the following program:

```
let$ y : (x : (z : bool¹ ⊢ bool¹) ⊢ bool¹) =                    1
  let$ z = <true> in <x with z = z>                             2
in                                                              3
  let$ x : (z : bool¹ ⊢ bool¹) = <not z> in                     4
  let$ z = <false> in                                           5
  <(y with x = x) and z>                                        6
```

The program first defines `y`, using the same definition as in §4.2.1. In line 4-5, it defines `x`, which depends on `z`, and then defines `z`. Finally, the program returns the code of the `and` operator applied to `y with x = x` and `z`. In line 6, which definition of `z` is supplied to `x`?

To answer this, let us go through the reduction steps. First, line 2 reduces to `<x with z = true>`, as discussed in the previous subsection.

```
let$ y : (x : (z : bool¹ ⊢ bool¹) ⊢ bool¹) =
  <x with z = true>
in ...
```

Then, `y` is substituted with `<x with z = true>` in the let body, where the delayed substitution `with x = x` is applied.

```
let$ x : (z : bool¹ ⊢ bool¹) = <not z> in
let$ z = <false> in
<(x with z = true) and z>
```

Next, `x` is substituted with `<not z>`. The delayed substitution `with z = true` is then applied, yielding `<not true>`.

```
let$ z = <false> in
<(not true) and z>
```

Finally, `z` is substituted with `<false>`, resulting in the value:

```
<(not true) and false>
```

Therefore, the answer to our earlier question is that `x` uses `z = true`.

What if we wanted `x` to use the definition `z = false` instead? We can achieve that by either modifying the definition of `y` to explicitly capture `z`:

```
let$ y : (x : (z : bool¹ ⊢ bool¹); z : bool¹ ⊢ bool¹) =
  <x with z = z>
in ...
  <(y with x = x; z = z) and z>
```

or by changing the definition of `y` to capture a non-capturing version of `x`:

```
let$ y : (x : bool¹ ⊢ bool¹) =
  <x>
in ...
  <(y with x = (x with z = z)) and z>
```

These examples illustrate the ability of our type system to express and enforce different kinds of variable dependencies within unhygienic programs.

## 5 Translation Between $\lambda^{\bigcirc\triangleright}$ and Linear-Time Temporal Logic

This section presents translations between $\lambda^{\bigcirc\triangleright}$ and Davies [1996]'s staging calculus $\lambda^{\bigcirc}$, which is based on linear-time temporal logic (LTL). §5.1 first translates $\lambda^{\bigcirc\triangleright}$'s types and judgments to formulas in Kojima and Igarashi [2011]'s constructive LTL (CLTL), the Hilbert-style counterpart of LTL. §5.2 then present a type-preserving translation from $\lambda^{\bigcirc\triangleright}$ to $\lambda^{\bigcirc}$. §5.3 subsequently shows that CLTL formulas are provable in $\lambda^{\bigcirc\triangleright}$. Thus, $\lambda^{\bigcirc\triangleright}$ is sound and complete with respect to CLTL. Finally. §5.4 briefly discusses a translation from $\lambda^{\bigcirc}$ to $\lambda^{\bigcirc\triangleright}$ following the approach in Xie et al. [2022].

### 5.1 $\lambda^{\bigcirc\triangleright}$ Types and Judgments to CLTL Formulas

CLTL [Kojima and Igarashi 2011] is a Hilbert-style axiomatization of $\lambda^{\bigcirc}$, which is characterized by the following axioms and rules:

**Axioms** (1) any intuitionistic tautology instance
(2) $\mathbf{K} : \bigcirc(A \to B) \to \bigcirc A \to \bigcirc B$ (3) $\mathbf{CK} : (\bigcirc A \to \bigcirc B) \to \bigcirc(A \to B)$

**Rules** (1) If $A \to B$ and $A$, then $B$. (2) If $A$, then $\bigcirc A$.

We translate types and judgments in $\lambda^{\bigcirc\triangleright}$ to CLTL formulas. Intuitively, this translation involves adding the appropriate number of circles to correspond to the staging level. For example, type $[\Delta \vdash A] \to B$ corresponds to $(\bigcirc\Delta \to A) \to B$, and $\Delta \triangleright A$ corresponds to $\bigcirc\Delta \to A$. Since CLTL has the equivalence $(\bigcirc A \to \bigcirc B) \leftrightarrow \bigcirc(A \to B)$, the specific placement of the circle modalities is not crucial when considering provability, although it will affect the term translation (§5.2). The formal translation for types is defined as follows:

$\boxed{[\![A^n]\!]}$ *(Type Translation)*

$$[\![([\Delta \vdash A] \to B)^n]\!] \coloneqq (\Delta \downarrow^n [\![A^n]\!]) \to [\![B^n]\!] \qquad [\![\mathbf{bool}^n]\!] \coloneqq \mathbf{bool}$$
$$[\![(\Delta \triangleright A)^n]\!] \coloneqq \Delta \downarrow^n [\![A^n]\!] \qquad [\![(\bigcirc A)^n]\!] \coloneqq \bigcirc [\![A^{n+1}]\!]$$

$\boxed{\Gamma \downarrow^n A}$ *(Context to Implications Translation)*

$$\cdot \downarrow^n A \coloneqq A \qquad (\Gamma, x : [\Delta \vdash^m A]) \downarrow^n B \coloneqq \Gamma \downarrow^n \left(\bigcirc^{m-n}(\Delta \downarrow^m [\![A]\!]) \to B\right)$$

The notation $(\Gamma \downarrow^n A)$ flattens $\Gamma$ into a nested chain of implications pointing to $A$, adding $\bigcirc$ to lower each item from its original level to level $n$, such that if $\Gamma = x_1 : [\Delta_1 \vdash^{m_1} A_1], \ldots, x_k : [\Delta_k \vdash^{m_k} A_k]$, then $\Gamma \downarrow^n A = \bigcirc^{m_1-n}(\Delta_1 \downarrow^{m_1} [\![A_1]\!]) \to \cdots \to \bigcirc^{m_k-n}(\Delta_k \downarrow^{m_k} [\![A_k]\!]) \to A$..

Then, a $\lambda^{\bigcirc\triangleright}$ typing judgment $\Gamma \vdash^n e : A$ corresponds to the CLTL formula $\Gamma \downarrow^n [\![A]\!]$. We prove that the translation is sound by induction on the typing derivations.

Lemma 5.1 (Translation Soundness). *If $\Gamma \vdash^n e : A$ for some $e$ in $\lambda^{\bigcirc\triangleright}$, then $\vdash \Gamma \downarrow^n [\![A]\!]$ in CLTL.*

## 5.2 Translation from $\lambda^{\bigcirc\triangleright}$ to $\lambda^{\bigcirc}$

We now define a translation from $\lambda^{\bigcirc\triangleright}$ to $\lambda^{\bigcirc}$. The translation preserves types but may introduce addition beta redexes inside quotations, similar to the example in §2.2. The translation from $\lambda^{\bigcirc\triangleright}$ contexts to $\lambda^{\bigcirc}$ contexts is given below, where each entry is flattened using the CLTL translation.

$\boxed{[\![\Gamma]\!]}$ *(Context to Context Translation)*

$$[\![\cdot]\!] \coloneqq \cdot \qquad [\![\Gamma, x : [\Delta \vdash^m A]]\!] \coloneqq [\![\Gamma]\!], x : (\Delta \downarrow^m [\![A]\!])^m$$

We then define the term translation with selected rules as follows, where $\langle e \rangle^n$ quotes $e$ by $n$ times, $\$^n(e)$ splices $e$ by $n$ times, $(\boldsymbol{\lambda}\Delta.\ e)$ abstracts an unhygienic term $e$ with respect to $\Delta$ using lambda abstractions, and $(x \bullet \sigma)$ applies an variables $x$ to each translated element in $\sigma$.

$\boxed{[\![e]\!]}$ *(Expression Translation)*

$$[\![x_\sigma]\!] \coloneqq x \bullet \sigma \qquad [\![\mathbf{let}_\Delta \langle x : A \rangle = e_1 \mathbf{\ in\ } e_2]\!] \coloneqq \mathbf{let}\ x = \langle \boldsymbol{\lambda}\Delta.\ \$[\![e_1]\!] \rangle \mathbf{\ in\ } ([\![e_2]\!][\$(x)/x])$$
$$[\![\lambda_\Delta x : A.\ e]\!] \coloneqq \lambda x.\ [\![e]\!]$$
$$[\![e_1\ e_2]\!] \coloneqq [\![e_1]\!]\ (\boldsymbol{\lambda}\Delta.\ [\![e_2]\!]) \qquad [\![\mathbf{wrap}_\Delta e]\!] \coloneqq \boldsymbol{\lambda}\Delta.\ [\![e]\!]$$
$$[\![\langle e \rangle]\!] \coloneqq \langle [\![e]\!] \rangle \qquad [\![\mathbf{let\ wrap}_\Delta\ x : A = e_1 \mathbf{\ in\ } e_2]\!] \coloneqq \mathbf{let}\ x = [\![e_1]\!] \mathbf{\ in\ } [\![e_2]\!]$$

$\boxed{\boldsymbol{\lambda}\Delta.\ e}$ *(Dependency Abstraction)*

$$\boldsymbol{\lambda}(\cdot).\ e \coloneqq e$$
$$\boldsymbol{\lambda}(\Delta, x : [\Delta' \vdash^m A]).\ e \coloneqq \boldsymbol{\lambda}\Delta.\ (\lambda x.\ e[\$^{m-n}\ x/x])$$

$\boxed{x \bullet \sigma}$ *(Dependency Application)*

$$x \bullet (\cdot) \coloneqq x$$
$$x \bullet (\sigma, y^m_\Delta \mapsto e) \coloneqq (x \bullet \sigma)\ \langle \boldsymbol{\lambda}\Delta.\ [\![e]\!] \rangle^{m-n}$$
$$x \bullet (\sigma, y^m_\Delta \mapsto z) \coloneqq (x \bullet \sigma)\ \langle z \rangle^{m-n}$$

We prove that the translation preserves typing:

Lemma 5.2 ($[\![\cdot]\!]$ preserves typing). *If $\Gamma \vdash^n e : A$ in $\lambda^{\bigcirc\triangleright}$ then $[\![\Gamma]\!] \vdash^n [\![e]\!] : [\![A]\!]$ in $\lambda^{\bigcirc}$.*

### 5.3 CLTL Axioms as $\lambda^{\bigcirc\triangleright}$ Terms

Next, we show completeness of $\lambda^{\bigcirc\triangleright}$ with respect to CLTL through a reverse translation. Specifically, the axioms of CLTL [Kojima and Igarashi 2011] can be proven by the following $\lambda^{\bigcirc\triangleright}$ terms:

$$\mathbf{K} : \bigcirc(A \to B) \to \bigcirc A \to \bigcirc B \coloneqq \lambda f.\, \lambda x.\, \mathbf{let}\, \langle f' : A \to B \rangle = f\, \mathbf{in}\, \mathbf{let}\, \langle x' : A \rangle = x\, \mathbf{in}\, \langle f'\, x' \rangle$$

$$\mathbf{CK} : (\bigcirc A \to \bigcirc B) \to \bigcirc(A \to B) \coloneqq \lambda f.\, \mathbf{let}_{x:A^{n+1}} \langle y : B \rangle = f\, \langle x \rangle\, \mathbf{in}\, \langle \lambda x.\, y_{x \mapsto x} \rangle$$

where $A$ and $B$ are types staged at level $n+1$. Notably, the ability to introduce dependencies in rule LetQuote is crucial for proving **CK**. We have thus demonstrated that the $\bigcirc$ fragment of $\lambda^{\bigcirc\triangleright}$ is complete with respect to CLTL. With Lemma 5.1, $\lambda^{\bigcirc\triangleright}$ is sound and complete with respect to CLTL.

### 5.4 Translation from $\lambda^{\bigcirc}$ to $\lambda^{\bigcirc\triangleright}$

A direct translation from $\lambda^{\bigcirc}$ to $\lambda^{\bigcirc\triangleright}$ can be achieved using a translation similar to the one described by Xie et al. [2022] for translating $\lambda^{\bigcirc}$ to their $F^{[\![]\!]}$ calculus. Their translation lifts splice expressions and binds them as *splice environments* associated with the innermost enclosing quotation at the same level. Because $\lambda^{\bigcirc\triangleright}$ generalizes the splice environments of $F^{[\![]\!]}$, their translation can be readily adapted. In our case, this involves lifting the splice expressions and binding them as let-splice bindings instead. The lifting process captures all encountered lambda bindings as variable dependencies, and these splice variables are provided with delayed substitutions. As an example,

$$\langle \lambda y.\ \$(power\, \langle y \rangle\, 2) + 3 \rangle \quad \text{translates to} \quad \mathbf{let}_{(y:\mathbf{int})} \langle z : \mathbf{int} \rangle = power\, \langle y \rangle\, 2\, \mathbf{in}\, \langle \lambda y : \mathbf{int}.\ z_{y \mapsto y} + 3 \rangle$$

Once again, the ability to introduce variable dependencies plays a crucial role in the translation. For a more detailed explanation, we refer the reader to Xie et al. [2022].

## 6 Code Pattern Matching

This section introduces $\lambda^{\bigcirc\triangleright}_{\text{pat}}$, an extension of $\lambda^{\bigcirc\triangleright}$ with code pattern matching and code rewriting. We present its syntax (§6.1), typing rules (§6.2), and operational semantics (§6.3).

### 6.1 Syntax

We extend the syntax of $\lambda^{\bigcirc\triangleright}$ with two new expression forms: if-let expressions for code pattern matching and rewrite expressions for code rewriting.

$$e ::= \ldots \mid \mathbf{if\ let}_{\Delta} \langle p \rangle = e_1\, \mathbf{then}\, e_2\, \mathbf{else}\, e_3 \mid e_1\, \mathbf{rewrite}\, \langle p \rangle \mapsto e_2 \qquad \Gamma, \Delta, \Pi ::= \cdot \mid \Gamma, x : [\Delta \vdash^n A]$$

$$p ::= \hat{x} : A \mid \textit{(inherits every production of } e\textit{)} \qquad \pi ::= \cdot \mid \pi, x \mapsto y \mid \pi, x \mapsto p$$

$\mathbf{if\ let}_{\Delta} \langle p \rangle = e_1\, \mathbf{then}\, e_2\, \mathbf{else}\, e_3$ matches the code expression $e_1$ against the pattern $p$. This can be seen as a generalization of the $\mathbf{let}_{\Delta} \langle x : A \rangle = e_1\, \mathbf{in}\, e_2$ expression in $\lambda^{\bigcirc\triangleright}$, where $x : A$ becomes a general pattern $p$ and thus the match always succeeds. Inside the if-let expression, if the match succeeds, $e_2$ is evaluated with the pattern variables in $p$ bound to the match results. Otherwise, $e_3$ is evaluated, with the pattern variables unavailable. The multi-branch `match$` expression used in the concrete syntax can be desugared into nested if-let expressions. Compared to multi-branch matching, if-let expressions are more convenient for formalization and ensure matching totality. $e_1\, \mathbf{rewrite}\, \langle p \rangle \mapsto e_2$ replaces occurrences of the pattern $p$ within $e_1$ with $e_2$, where pattern variables in $p$ match sub-expressions in $e_1$ and become available within $e_2$.

Code patterns $p$ are expressions with pattern variables. To distinguish pattern variables from regular variables, we use $\hat{x}$ to denote pattern variables and $x$ for regular variables. We assume pattern variables are distinct within a pattern. For simplicity, Substitution patterns $\pi$ are used to match substitutions, where each entry can be either a variables or a pattern. Lastly, we write $\Pi$ to denote contexts of pattern variables, which contain same entries as regular contexts $\Gamma$ and $\Delta$.

### 6.2 Typing Rules

We extend expression typing $\Gamma \vdash^n e : A$ with the following two rules:

$\boxed{\Gamma \vdash^n e : A}$ *(Expression Typing (extended))*

$$\textsc{IfLet}\ \frac{\Gamma\restriction_{n+1}; \Delta^{n+1} \vdash^{n+1} p : A \rightsquigarrow \Pi^{n+1} \qquad \Gamma, \Delta \vdash^n e_1 : \bigcirc A \qquad \Gamma, \Pi \vdash^n e_2 : B \qquad \Gamma \vdash^n e_3 : B}{\Gamma \vdash^n \textbf{if let}_\Delta \langle p \rangle = e_1 \textbf{ then } e_2 \textbf{ else } e_3 : B}$$

$$\textsc{Rewrite}\ \frac{\Gamma \vdash^n e_1 : \bigcirc A \qquad \Gamma\restriction_{n+1}; \cdot \vdash^{n+1} p : B \rightsquigarrow \Pi^{n+1} \qquad \Gamma, \Pi \vdash^n e_2 : \bigcirc B}{\Gamma \vdash^n e_1 \textbf{ rewrite } \langle p \rangle \mapsto e_2 : \bigcirc A}$$

Rule IfLet first type-checks the pattern $p$, yielding a type $A$ and a pattern variable context $\Pi^{n+1}$ (the typing rules for patterns are explained below). It then checks that $e_1$ has type $\bigcirc A$, and extends the typing context with $\Pi^{n+1}$ when checking $e_2$. The rule can be seen as a generalization of rule LetQuote, which extends the context with only a single variable ($x : [\Delta \vdash^{n+1} A]$) when checking $e_2$. Finally, the expressions $e_2$ and $e_3$ must have the same type.

Rule Rewrite is similar. It first type-checks the pattern $p$, and extends the typing context with $\Pi^{n+1}$ when checking $e_1$. The rule ensures that both $e_1$ and $e_2$ have code types. While the expression $e_2$ can have any type $\bigcirc B$, only sub-expressions with the same type as $p$ are considered for rewriting.

The pattern typing judgment $\Gamma; \Delta \vdash^n p : A \rightsquigarrow \Pi$ checks the pattern $p$ under contexts $\Gamma$ and $\Delta$, producing a type $A$ and a context $\Pi$ of pattern variables. The judgment uses two typing contexts: the expression context $\Gamma$ contains variables from the surrounding context of the expression (see rules IfLet and Rewrite), allowing patterns to refer to existing variables, while the local context $\Delta$ contains local variables introduced either by $\textbf{let}_\Delta$ (rule IfLet) or within the pattern $p$. Separating these contexts ensures that each pattern variable captures the correct dependencies. For example, in $\langle (\lambda x.\ \hat{y})\ \hat{z} \rangle$, the pattern variable $\hat{y}$ will capture $x$ since it matches on a sub-expression that may contain $x$, while $\hat{z}$ will not. We present selected typing rules below.

$\boxed{\Gamma; \Delta \vdash^n p : A \rightsquigarrow \Pi}$ *(Code Pattern Typing (excerpt))*

$$\textsc{P-PVar}\ \frac{}{\Gamma; \Delta \vdash^n (\hat{x} : A) : A \rightsquigarrow x : [\Delta \vdash^n A]} \qquad \textsc{P-VarSubst1}\ \frac{\Gamma \ni x : [\Delta' \vdash^n A] \qquad \Gamma, \Delta \vdash \sigma : \Delta'}{\Gamma; \Delta \vdash^n x_\sigma : A \rightsquigarrow \cdot}$$

$$\textsc{P-VarSubst2}\ \frac{\Delta \ni x : [\Delta' \vdash^n A] \qquad \Gamma; \Delta \vdash \pi : \Delta' \rightsquigarrow \Pi}{\Gamma; \Delta \vdash^n x_\pi : A \rightsquigarrow \Pi} \qquad \textsc{P-CtxAbs}\ \frac{\Gamma; \Delta, x : [\Delta'^{n+1} \vdash^n A] \vdash^n p : B \rightsquigarrow \Pi}{\Gamma; \Delta \vdash^n (\lambda_{\Delta'} x : A.\ p) : [\Delta' \vdash A] \to B \rightsquigarrow \Pi}$$

$$\textsc{P-CtxApp}\ \frac{\Gamma; \Delta \vdash^n p_1 : [\Delta' \vdash A] \to B \rightsquigarrow \Pi_1 \qquad \Gamma; \Delta, \Delta' \vdash^n p_2 : A \rightsquigarrow \Pi_2}{\Gamma; \Delta \vdash^n p_1\ p_2 : B \rightsquigarrow \Pi_1, \Pi_2}$$

Rule P-PVar types pattern variables, producing a single pattern variable that captures the local context $\Delta$. Rule P-VarSubst1 matches variables in the expression context $\Gamma$ using a constant substitution $\sigma$, thereby enabling matching from any expression context. This contrasts with contextual type systems [Jang et al. 2022; Parreaux et al. 2017], which restrict matching to variables from the context the code is closed at. Rule P-VarSubst2 matches variables in the local context $\Delta$, allowing for further matching against the variable's associated substitution using a substitution pattern $\pi$. This distinction between rule P-VarSubst1 and rule P-VarSubst2 ensures distinct pattern variables under substitution. Specifically, variables in $\Gamma$ may be substituted with arbitrary terms. For example, consider a pattern $\langle x_{y \mapsto \hat{z}} \rangle$ where $x \in \Gamma$ and $\hat{z}$ is a pattern variable. If $x$ is substituted

with a term where $y$ is not used linearly, such as $0$ or $y + y$, then $\hat{z}$ would appear zero or multiple times, respectively.[3]

In rule P-CtxAbs, the local variable $x$ is added to $\Delta$ to check the pattern $p$. In rule P-CtxApp, the pattern variables produced by $p_1$ and $p_2$ are combined in the result.

Similarly, the judgment $\Gamma; \Delta \vdash \pi : \Gamma' \rightsquigarrow \Pi$ type-checks substitution patterns, producing a typing context $\Gamma'$ and a context $\Pi$ of pattern variables:

$\boxed{\Gamma; \Delta \vdash \pi : \Gamma' \rightsquigarrow \Pi}$ *(Substitution Pattern Typing)*

$$\textsc{P-S-Empty}\ \frac{}{\Gamma; \Delta \vdash \cdot : \cdot \rightsquigarrow \cdot} \qquad \textsc{P-S-Var}\ \frac{\Gamma; \Delta \vdash \pi : \Gamma' \rightsquigarrow \Pi \qquad \Gamma, \Delta \ni y : [\Delta' \vdash^m A]}{\Gamma; \Delta \vdash (\pi, x \mapsto y) : \Gamma', x : [\Delta' \vdash^m A] \rightsquigarrow \Pi}$$

$$\textsc{P-S-Pattern}\ \frac{\Gamma; \Delta \vdash \pi : \Gamma' \rightsquigarrow \Pi_1 \qquad \Gamma\restriction_m; \Delta\restriction_m, \Delta'^m \vdash^m p : A \rightsquigarrow \Pi_2}{\Gamma; \Delta \vdash (\pi, x \mapsto p) : \Gamma', x : [\Delta' \vdash^m A] \rightsquigarrow \Pi_1, \Pi_2}$$

Rule P-S-Empty is trivial. When the substitution entry maps to a variable $y$, rule P-S-Var ensures that $y$ is in either $\Gamma$ or $\Delta$, and recursively types $\pi$. When the substitution entry maps to a pattern $p$, rule P-S-Pattern type-checks the pattern $p$, and combines $\Pi_1$ and $\Pi_2$.

### 6.3 Operational Semantics

Substitutions and evaluation contexts are mostly straightforward extensions of those in $\lambda^{\bigcirc\triangleright}$.

$$E ::= \ \ldots \mid \textbf{if let}_\Delta \langle p \rangle = E \textbf{ then } e_2 \textbf{ else } e_3 \mid E \textbf{ rewrite } \langle p \rangle \mapsto e_2 \mid v_1 \textbf{ rewrite } \langle p \rangle \mapsto E$$

The reduction rules for if-let and rewrite expressions are defined as follows, which rely on the meta-level functions match (§6.3.1) and rewrite (§6.3.2), respectively.

$\boxed{\Gamma \vdash^n e_1 \longrightarrow_\beta e_2}$ *(Reduction (extended))*

$$\textsc{IfLetQuote1}\ \frac{\mathsf{match}(p; e_1) = \sigma}{\Gamma \vdash^n \textbf{if let}_\Delta \langle p \rangle = \langle e_1 \rangle \textbf{ then } e_2 \textbf{ else } e_3 \longrightarrow_\beta e_2[\mathsf{id}_\Gamma, \sigma]} \qquad \textsc{IfLetQuote2}\ \frac{\mathsf{match}(p; e_1)\ \textbf{undefined}}{\Gamma \vdash^n \textbf{if let}_\Delta \langle p \rangle = \langle e_1 \rangle \textbf{ then } e_2 \textbf{ else } e_3 \longrightarrow_\beta e_3}$$

$$\textsc{RewriteQuoteQuote}\ \frac{}{\Gamma \vdash^n \langle e_1 \rangle \textbf{ rewrite } \langle p \rangle \mapsto \langle e_2 \rangle \longrightarrow_\beta \langle \mathsf{rewrite}_{\mathsf{BottomUp}}(e_1; p; e_2) \rangle}$$

We have proven the preservation (Theorem 4.3) and progress (Theorem 4.4) theorems for $\lambda^{\bigcirc\triangleright}_{\mathrm{pat}}$. The proofs are available in Agda (§8).

*6.3.1* ***Pattern Matching.*** Matching is defined as partial functions match. We present a selection of the rules below. Note that $\mathsf{match}(p; e)$ is defined up to $\alpha$-equivalence on $e$: that is, we allow renaming of bound variables in $e$. For contexts introduced by $\lambda_\Delta$, $\textbf{let}_\Delta$, or $\textbf{if let}_\Delta$, renaming is allowed, but reordering is not. These align with the de Bruijn representation used in the formalization (§8).

$\boxed{\mathsf{match}(p; e), \mathsf{match}(\pi; \sigma)}$ *(Expression and Substitution Matching (excerpt))*

$$\mathsf{match}(\hat{x} : A; e) \coloneqq x \mapsto e \qquad \mathsf{match}((\lambda_\Delta x : A.\ p); (\lambda_\Delta x : A.\ e)) \coloneqq \mathsf{match}(p; e)$$

$$\mathsf{match}(x_\pi; x_\sigma) \coloneqq \mathsf{match}(\pi; \sigma) \qquad \mathsf{match}(p_1\ p_2; e_1\ e_2) \coloneqq \mathsf{match}(p_1; e_1), \mathsf{match}(p_2; e_2)$$

[3] Allowing a pattern variable to appear multiple times within a pattern is straightforward, by simply requiring it to map to the same expression during the merging of $\Pi$ contexts (e.g. in rule P-CtxApp). Conversely, a pattern variable must continue appearing at least once, as it can be used within the matching branch. Mœbius [Jang et al. 2022] declares all pattern variables upfront, and the operational semantics requires a substitution of every declared pattern variable when matching an expression. Consequently, their operational semantics is non-deterministic when a pattern variable is unused.

*6.3.2* ***Rewriting.*** Rewriting builds on the matching function by applying it to sub-expressions within a target expression, replacing any matches with a specified replacement expression. Given a target expression $\Gamma \vdash^n e_1 : A$, a pattern $\Gamma; \cdot \vdash^n p : B \rightsquigarrow \Pi$, and a replacement expression $\Gamma, \Pi \vdash^n e_2 : B$, a meta-level function rewrite produces an expression of the same type as $e_1$.

The rewrite function can be implemented using various strategies. For example, $\mathsf{rewrite}_{\mathsf{BottomUp}}$ rewrites sub-expressions before rewriting their parent expression. We define a helper function $\mathsf{applyBottomUp}(F; e)$, which applies a meta-level function $F$ to each sub-expression of $e$ with same level as $e$, replacing them in bottom-up order. For instance, $\mathsf{applyBottomUp}(F; (\lambda x.\ f\ x)) = F(\lambda x.\ F(F(f)\ F(x)))$. The complete definition of applyBottomUp is given in Appendix B.4.

$\boxed{\mathsf{rewrite}_{\mathsf{BottomUp}}(e_1; p; e_2)}$ *(Bottom-up Rewriting)*

$$\mathsf{rewrite}_{\mathsf{BottomUp}}(e_1; p; e_2) \coloneqq \mathsf{applyBottomUp}(F; e_1), \text{ where}$$

$$F(e) \coloneqq \begin{cases} e_2[\mathsf{id}_\Gamma, \sigma] & \text{if } p \text{ and } e \text{ have the same type and } \mathsf{match}(p; e) = \sigma, \\ e & \text{otherwise.} \end{cases}$$

At each sub-expression $e$ of $e_1$, the function $F$ checks if $e$ matches the pattern $p$. If so, it substitutes $e_2$ with the match result $\sigma$ and replace $e$ with it. Otherwise, it leaves $e$ unchanged. Other strategies, such as rewriting only the top-most occurrences of the pattern, can also be defined. We include the top-most rewriting strategy in the appendix and in our Agda development. We may also iterate the rewriting process until a fixpoint is reached [Parreaux et al. 2017].

Lastly, we note that in rule RewriteQuoteQuote, the replacement expression is evaluated into a value $\langle e_2 \rangle$, before $\mathsf{rewrite}_{\mathsf{BottomUp}}$ takes place. Thus, while the replacement expression may refer to pattern variables in $p$, its evaluation cannot access the specific values matched by $p$. In some cases, we may want to inspect the match results within the replacement expression (§2.6). This requires a different semantics where the replacement expression is evaluated *at each pattern match*, with the corresponding match results available. We defer the definition of this alternative semantics to §7.3, where it can be more naturally presented after the introduction of the denotational semantics.

## 7 Denotational Semantics

This section presents the denotational semantics for $\lambda^{\bigcirc\triangleright}$ and $\lambda^{\bigcirc\triangleright}_{\mathsf{pat}}$, providing interpretations of types(§7.1), contexts (§7.2), and expressions (§7.3). We then define a logical relation (§7.4) and establish the fundamental property (Theorem 7.1), which implies termination of the operational semantics and the adequacy of the denotational semantics.

At a high-level, since evaluation happens at a specific stage $n$, we refer to this stage $n$ as the *current stage*, and any stages from $n+1$ onward as *later stages*. Semantic interpretations are given to current-stage terms, while later-stage terms remain syntactic. We employ a Kripke-style model [Asai et al. 2014; Mitchell and Moggi 1991], where "worlds" are later-stage contexts $\Gamma^{n+1}$ related by later-stage substitutions $\Gamma'^{n+1} \vdash \sigma : \Gamma^{n+1}$. Types and contexts are interpreted as sets indexed by later-stage-context, where these sets are required to be *monotonic* with respect to the index. Formally, given an indexed set $D_{(\cdot)}$ and a later-stage substitution $\Gamma' \vdash \sigma : \Gamma$, there is a function $D_\Gamma \to D_{\Gamma'}$ mapping elements from $D_\Gamma$ to $D_{\Gamma'}$. Categorically, it can be viewed as a presheaf model [Altenkirch et al. 2005; Kavvos 2024] over the category of later-stage substitutions.

### 7.1 Type Interpretation

Interpretations of level-$n$ types are defined by recursion on the type structure:

$\boxed{(\!|\, A^n \,|\!)_\Gamma}$ *(Type Interpretation)*

$$(\!|\,[\Delta \vdash A] \to B\,|\!)_\Gamma := \prod_{(\Gamma' \vdash \sigma : \Gamma)} (\!|\,A\,|\!)_{\Gamma',\Delta} \to (\!|\,B\,|\!)_{\Gamma'} \qquad (\!|\,\mathbf{bool}\,|\!)_\Gamma := \{\mathsf{True}, \mathsf{False}\}$$
$$(\!|\,\Delta \rhd A\,|\!)_\Gamma := (\!|\,A\,|\!)_{\Gamma,\Delta} \qquad (\!|\,\bigcirc A\,|\!)_\Gamma := \{\, e \mid \Gamma \vdash^{n+1} e : A \,\}$$

Function types are interpreted as dependent functions that, given a later-stage substitution from $\Gamma$ to $\Gamma'$ (where $\Gamma'$ can be any later-stage context), and map elements of $(\!|\,A\,|\!)_{\Gamma',\Delta}$ to elements of $(\!|\,B\,|\!)_{\Gamma'}$. $\Delta \rhd A$ is interpreted as the interpretation of $A$ under the extended context $\Gamma, \Delta$. **bool** is interpreted as the set of Booleans. $\bigcirc A$ is interpreted as the set of syntactic expressions of type $A$ at level $n+1$, modulo $\alpha$-equivalence.

We show that the interpretation is indeed monotonic. Given a type $A$, an element $d \in (\!|\,A\,|\!)_\Gamma$ (denoted $d^A$), and a later-stage substitution $\Gamma' \vdash \sigma : \Gamma$, we define $d[\sigma] \in (\!|\,A\,|\!)_{\Gamma'}$ as the result of applying the substitution $\sigma$ to $d$. The definition proceeds by recursion on the type structure:

$\boxed{d[\sigma]}$ *(Element Substitution)*

$$f^{A\to B}[\sigma] := \lambda\sigma'.\, f\ (\sigma[\sigma']) \qquad b^{\mathbf{bool}}[\sigma] := b$$
$$d^{\Delta \rhd A}[\sigma] := d^A[\sigma, \mathrm{id}_\Delta] \qquad e^{\bigcirc A}[\sigma] := e[\sigma]$$

## 7.2 Context Interpretation

Typing contexts at level $n$ are interpreted as the product of the interpretations of their entries, where the interpretation of each entry depends on whether it is at the current stage $n$. Note that in the following definition, the context $\Gamma$ being interpreted is at level $n$, while the context $\Gamma'$ and $\Gamma''$ used as indices are at level $n+1$.

$\boxed{(\!|\,\Gamma\,|\!)_{\Gamma'}}$ *(Context Interpretation (Environments))*

$$(\!|\,\Gamma^n\,|\!)_{\Gamma'} := \prod_{\Gamma \ni x : [\Delta \vdash^m A]} \begin{cases} \prod_{(\Gamma'' \vdash \sigma : \Gamma')} (\!|\,\Delta\,|\!)_{\Gamma''} \to (\!|\,A\,|\!)_{\Gamma''} & \text{if } m = n, \\ \{\, y \mid \Gamma' \ni y : [\Delta \vdash^m A] \,\} \cup \{\, e \mid \Gamma'\restriction_m, \Delta \vdash^m e : A \,\} & \text{if } m > n. \end{cases}$$

Specifically, current-stage entries $\Gamma \ni x : [\Delta \vdash^n A]$ are interpreted as dependent functions from $(\!|\,\Delta\,|\!)$ to $(\!|\,A\,|\!)$, similar to the interpretation of function types. Later-stage entries $\Gamma \ni x : [\Delta \vdash^m A]$ (where $m > n$) are interpreted as syntactic substitution entries under $\Gamma'$, which can be either a variable $\Gamma' \ni y : [\Delta \vdash^m A]$ (rule S-Rename) or an expression $\Gamma'\restriction_m, \Delta \vdash^m e : A$ (rule S-Subst).

We call an element $\rho \in (\!|\,\Gamma\,|\!)_{\Gamma'}$ an *environment*, and write $\rho(x)$ to denote the entry corresponding to $x$ in $\rho$. As with types, we show that the interpretation is monotonic. Given an environment $\rho \in (\!|\,\Gamma\,|\!)_{\Gamma'}$ and a later-stage substitution $\Gamma'' \vdash \sigma : \Gamma'$, we define $\rho[\sigma] \in (\!|\,\Gamma\,|\!)_{\Gamma''}$ by applying the substitution $\sigma$ to each entry in $\rho$:

$\boxed{\rho[\sigma]}$ *(Environment Substitution)*

$$\rho[\sigma](x) := \begin{cases} \lambda\sigma'.\, \rho(x)\ (\sigma[\sigma']) & \text{if } m = n, \\ e[\sigma\restriction_m, \mathrm{id}_\Delta] & \text{if } m > n \text{ and } \rho(x) = e, \\ \sigma(y) & \text{if } m > n \text{ and } \rho(x) = y, \end{cases} \quad \text{for each } \Gamma \ni x : [\Delta \vdash^m A].$$

where the case for $m = n$ is defined similarly to the function type, and the cases for $m > n$ are handled by substituting the substitution entry as defined in §4.1.

We write $\rho\restriction_{n+1}$ to remove all entries from $\rho$ that are at level $n$, turning it into a substitution at level $n+1$. We can lift an element $d \in (\!|\,A\,|\!)_{\Gamma,\Delta}$ to a singleton environment $\{x^n \mapsto d\}$, whose definition is given as follows:

$$\{x^n \mapsto d\} \in \textstyle\prod_{\Gamma' \vdash \sigma : \Gamma} (\!|\,\Delta\,|\!)_{\Gamma'} \to (\!|\,A\,|\!)_{\Gamma'} \qquad \{x^n \mapsto d\} := \lambda\sigma\ \rho.\, d[\sigma', \rho\restriction_{n+1}]$$

Note that in the above definition, since $\Delta$ is already at level $n+1$, $\rho\restriction_{n+1}$ does not remove any entry from $\rho$, but instead just converts it into a substitution. We write $\rho \cup \rho'$ to add entries to an environment, where $\rho'$ can either be an environment or a later-stage substitution.

### 7.3 Expression Interpretation

Given a later-stage context $\Gamma'$, expressions $\Gamma \vdash^n e : A$ are interpreted as functions $(\!|\, \Gamma \,|\!)_{\Gamma'} \to (\!|\, A \,|\!)_{\Gamma'}$, and substitutions $\Gamma \vdash \sigma : \Delta$ are interpreted as functions $(\!|\, \Gamma \,|\!)_{\Gamma'} \to (\!|\, \Delta \,|\!)_{\Gamma'}$. This interpretation can be viewed as a definitional interpreter for the language, which recursively interprets an expression under a given environment $\rho \in (\!|\, \Gamma \,|\!)_{\Gamma'}$ and extending the environment as needed. We present a selection of rules for expression interpretation for $\lambda^{\bigcirc\triangleright}$:

$\boxed{(\!|\, e \,|\!)_{\Gamma'}, (\!|\, \sigma \,|\!)_{\Gamma'}}$ *(Expression and Substitution Interpretation (excerpt))*

$$(\!|\, x_{\sigma_1} \,|\!)_{\Gamma'} \coloneqq \lambda\rho.\ \rho(x)\ \mathrm{id}_{\Gamma'}\ (\!|\, \sigma_1 \,|\!)_{\Gamma'} \qquad (\!|\, \lambda_\Delta x : A.\ e \,|\!)_{\Gamma'} \coloneqq \lambda\rho.\ \lambda\sigma'\ d.\ (\!|\, e \,|\!)_{\Gamma'}\ (\rho[\sigma'] \cup \{x^n \mapsto d\})$$

$$(\!|\, \langle e \rangle \,|\!)_{\Gamma'} \coloneqq \lambda\rho.\ e[\rho\restriction_{n+1}] \qquad (\!|\, e_1\ e_2 \,|\!)_{\Gamma'} \coloneqq \lambda\rho.\ (\!|\, e_1 \,|\!)_{\Gamma'}\ \rho\ \mathrm{id}_{\Gamma'}\ ((\!|\, e_2 \,|\!)_{\Gamma',\Delta}\ (\rho \cup \mathrm{id}_\Delta))$$

$$(\!|\, \mathbf{let}_\Delta \langle x : A \rangle = e_1\ \mathbf{in}\ e_2 \,|\!)_{\Gamma'} \coloneqq \lambda\rho.\ \mathrm{let}\ e = (\!|\, e_1 \,|\!)_{\Gamma',\Delta}\ (\rho \cup \mathrm{id}_\Delta)\ \mathrm{in}\ (\!|\, e_2 \,|\!)_{\Gamma'}\ (\rho \cup (x \mapsto e))$$

For $\lambda^{\bigcirc\triangleright}_{\mathrm{pat}}$, the additional expression forms are interpreted as follows:

$$(\!|\, \mathbf{if\ let}_\Delta \langle p \rangle = e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 \,|\!)_{\Gamma'} \coloneqq \lambda\rho.\ \begin{cases} (\!|\, e_2 \,|\!)_{\Gamma'}\ (\rho \cup \sigma) & \text{if } \mathrm{match}\big(p;\ (\!|\, e_1 \,|\!)_{\Gamma',\Delta}\ (\rho \cup \mathrm{id}_\Delta)\big) = \sigma, \\ (\!|\, e_3 \,|\!)_{\Gamma'}\ \rho & \text{otherwise.} \end{cases}$$

$$(\!|\, e_1\ \mathbf{rewrite}\ \langle p_2 \rangle \mapsto e_2 \,|\!)_{\Gamma'} \coloneqq \lambda\rho.\ \mathrm{rewrite}_{\mathsf{BottomUp}}(p_2[\rho\restriction_{n+1}];\ (\!|\, e_2 \,|\!)_{\Gamma',\Pi}\ (\rho \cup \mathrm{id}_\Pi);\ (\!|\, e_1 \,|\!)_{\Gamma'}\ \rho)$$

The rewrite interpretation corresponds to the small-step semantics (§6.3), where the replacement expression $e_2$ is evaluated before rewriting happens. Using the denotational semantics, we can alternatively evaluate the replacement expression at each occurrence of the pattern $p$ within $e_1$, thus allowing the replacement expression to inspect the specific match results. This approach is analogous to the big-step semantics of the rewrite construct in Parreaux et al. [2017].

$$(\!|\, e_1\ \mathbf{rewrite}\ \langle p_2 \rangle \mapsto e_2 \,|\!)_{\Gamma'} \coloneqq \lambda\rho.\ \mathsf{applyBottomUp}\Big(\lambda e.\ \begin{cases} (\!|\, e_2 \,|\!)_{\Gamma'}\ (\rho \cup \sigma) & \text{if } p_2 \text{ and } e \text{ have the same type and } \mathrm{match}(p_2[\rho\restriction_{n+1}]; e) = \sigma, \\ e & \text{otherwise.} \end{cases};\ (\!|\, e_1 \,|\!)_{\Gamma'}\ \rho\Big)$$

This interpretation uses the applyBottomUp function to apply $e_2$'s interpretation to every matching sub-expression of the target expression $e_1$. Crucially, the environment $\rho$ is extended with the substitution $\sigma$ that represents the match result, rather than $\mathrm{id}_\Pi$ as in the previous definition. This allows $e_1$ to inspect the match result using $\lambda^{\bigcirc\triangleright}_{\mathrm{pat}}$'s analytic capabilities. The alternative semantics for rewriting is strictly more expressive than the small-step version, since the the original semantics can be encoded in the alternative semantics by evaluating the replacement expression before rewriting.

### 7.4 Adequacy

To prove the adequacy of the denotational semantics, we define the logical relations $\mathcal{V}$, $\mathcal{E}$, and $\mathcal{G}$, which relate values, expressions, and environments with their donotaion, respectively.

$\boxed{\mathcal{V}^A_\Gamma, \mathcal{E}^A_\Gamma}$ *(Value and Expression Relations)*

$$\mathcal{V}^{[\Delta\vdash A]\to B}_\Gamma \coloneqq \{\ (\lambda_\Delta x : A.\ e; f) \mid \forall(\Gamma' \vdash \sigma : \Gamma).\ \forall(v; d) \in \mathcal{V}^A_{\Gamma',\Delta}.\ (e[\sigma, x \mapsto v]; f\ \sigma\ d) \in \mathcal{E}^B_{\Gamma'}\ \}$$

$$\mathcal{V}^{\Delta\triangleright A}_\Gamma \coloneqq \{\ (\mathbf{wrap}_\Delta e; d) \mid (e; d) \in \mathcal{V}^A_{\Gamma,\Delta}\ \} \qquad \mathcal{V}^{\mathbf{bool}}_\Gamma \coloneqq \{(\mathbf{true}; \mathrm{True}), (\mathbf{false}; \mathrm{False})\}$$

$$\mathcal{V}^{\bigcirc A}_\Gamma \coloneqq \{\ (\langle e \rangle; e) \mid \Gamma \vdash^{n+1} e : A\ \}$$

$$\mathcal{E}^A_\Gamma \coloneqq \{\ (e; d) \mid \exists v.\ (\Gamma \vdash^n e \longrightarrow^* v) \wedge (v; d) \in \mathcal{V}^A_\Gamma\ \}$$

$\boxed{\mathcal{G}^\Gamma_{\Gamma'}}$ *(Environment Relation)*

$$\mathcal{G}^{\Gamma}_{\Gamma'} \coloneqq \prod_{\Gamma \ni x:[\Delta \vdash^m A]} \begin{cases} \{\, (e; f) \mid \forall(\Gamma'' \vdash \sigma : \Gamma').\ \forall(\sigma'; \rho') \in \mathcal{G}^{\Delta}_{\Gamma''}.\ (e[\sigma, \sigma']; f\ \sigma\ \rho') \in \mathcal{E}^{A}_{\Gamma''} \,\} & \text{if } m = n, \\ \{\, (y; y) \mid \Gamma' \ni y : [\Delta \vdash^m A] \,\} \cup \{\, (e; e) \mid \Gamma' \restriction_m, \Delta \vdash^m e : A \,\} & \text{if } m > n. \end{cases}$$

The relations generally follow the structure of the denotational semantics. The $\mathcal{V}^{A}_{\Gamma}$ relation relates values of type $A$ under context $\Gamma$ to elements in $(\!|\, A \,|\!)_{\Gamma}$. For base types **bool** and $\bigcirc A$, $\mathcal{V}$ is essentially the identity relation, relating each value to a unique element. For function types, a lambda value $(\lambda_{\Delta} x : A.\ e)$ is related to a function $f$ if, for any later-stage substitution $\sigma$ and any value $v$ related to $d$, substituting $e$ with $\sigma$ and $v$ yields an expression related to $f\ \sigma\ d$. For $\Delta \triangleright A$, the relation extends the context with $\Delta$, as in the denotational semantics. The expression relation $\mathcal{E}$ relates an expression to an element if the expression reduces to a value related to that element. The environment relation $\mathcal{G}^{\Gamma}_{\Gamma'}$ relates substitutions with type $\Gamma' \vdash \sigma : \Gamma$ to elements in $(\!|\, \Gamma \,|\!)_{\Gamma'}$. For current-stage entries, an expression entry $e$ is related to a function $f$ in the same way lambda values are related to functions, while variable entries are not related. For later-stage entries, it relates the entry to itself.

We prove the fundamental property of the logical relations:

Theorem 7.1 (Fundamental Property).

- *If* $\Gamma \vdash^n e : A$ *and* $(\sigma; \rho) \in \mathcal{G}^{\Gamma}_{\Gamma'}$ *then* $(e[\sigma]; (\!|\, e \,|\!)_{\Gamma'}\ \rho) \in \mathcal{E}^{A}_{\Gamma'}$.
- *If* $\Gamma \vdash \sigma_1 : \Delta$ *and* $(\sigma; \rho) \in \mathcal{G}^{\Gamma}_{\Gamma'}$ *then* $(\sigma_1[\sigma]; (\!|\, \sigma \,|\!)_{\Gamma'}\ \rho) \in \mathcal{G}^{\Delta}_{\Gamma'}$.

The fundamental property implies termination of the operational semantics and the adequacy of the denotational semantics. In particular, consider a level-$n$ expression $\Gamma^{n+1} \vdash^n e : A$. Given $(\mathrm{id}_{\Gamma}; \mathrm{id}_{\Gamma}) \in \mathcal{G}^{\Gamma}_{\Gamma}$, it follows that $(e; (\!|\, e \,|\!)_{\Gamma}\ \mathrm{id}_{\Gamma}) \in \mathcal{E}^{A}_{\Gamma}$. Therefore, reducing $e$ terminates at some value $v$. Furthermore, $\mathcal{V}$ relates $v$ to $((\!|\, e \,|\!)_{\Gamma}\ \mathrm{id}_{\Gamma})$, the result of evaluating $e$ under the denotational semantics. Since the $\mathcal{V}$ is the identity relation at base types **bool** and $\bigcirc A$, we have $v = (\!|\, e \,|\!)_{\Gamma}\ \mathrm{id}_{\Gamma}$. That is, the result of the operational semantics coincides with the denotational semantics. We have proven the fundamental property in Agda for $\lambda^{\bigcirc\triangleright}$, using an inductive proof on the typing derivation and employing the associativity and commutativity properties of substitution (§4.1.2). We have not yet fully established the fundamental property for $\lambda^{\bigcirc\triangleright}_{\mathrm{pat}}$, but we expect it to hold as well. The main technical challenge is proving the substitution properties for $\lambda^{\bigcirc\triangleright}_{\mathrm{pat}}$, which is more involved because the number of expression forms doubles with the introduction of patterns.

## 8 Formalization in Agda

We formalize the syntax, typing rules, semantics, translations, stated lemmas and theorems in Agda. Our formalization leverages the agda-stdlib library [The Agda Community 2024] and adopts the style of *Programming Language Foundations in Agda* [Wadler et al. 2022]. We check all proofs with the `safe` flag to ensure soundness. Table 1 presents the structure of our definitions and proofs.

In the formalization, all contexts and types are intrinsically well-staged, and all expressions are intrinsically typed. The modules Data.StagedList and Data.StagedTree define intrinsically well-staged lists and rose trees, respectively. These modules are developed in a self-contained manner, making them reusable for other projects requiring well-staged data structures. In our formalization, they are used to represent the nested structure of typing contexts. In Data.StagedList, concatenation of well-staged lists is defined as a constructor rather than a function, effectively making it a tree rather than a list. This design choice allows other context operations to commute with concatenation by definition, simplifying proofs by reducing the need for casting between equivalent contexts. Variables are represented namelessly using de Bruijn indices. These simplifications contribute to a more concise formalization and ensure that pattern matching respects α-equivalence.

| **Definitions** | | $\lambda^{\bigcirc\triangleright}$ | $\lambda^{\bigcirc\triangleright}_{\text{pat}}$ |
|---|---|---|---|
| Data.StagedList/StagedTree | Intrinsically well-staged lists and rose trees. | 314 | |
| Context | Defines types and typing contexts. | 22 | 22 |
| Term | Defines intrinsically typed terms using de Bruijn indices. | 142 | 262 |
| Depth | Defines the depth of contexts and substitutions. | 114 | 114 |
| Substitution | Defines substitution. | 34 | 77 |
| Reduction | Defines operational semantics and proves safety properties. | 95 | 87 |
| Denotational | Defines the Kripke-style denotational semantics. | 92 | 136 |
| Examples | Examples of well-typed expressions and their evaluation. | 66 | 67 |
| **Pattern Matching** | | | |
| Matching | Defines the match function. | | 165 |
| Rewriting | Defines the rewrite function. | | 77 |
| Context.Equality | Proves decidable equality for $\lambda^{\bigcirc\triangleright}_{\text{pat}}$ contexts. | | 119 |
| Term.Equality | Proves decidable equality for $\lambda^{\bigcirc\triangleright}_{\text{pat}}$ terms. | | 362 |
| **Adequacy** | | | |
| Term.Properties | Proves properties of renaming operations. | 175 | |
| Substitution.Properties | Proves properties of substitution. | 341 | |
| Denotational.Adequacy | Proves the adequacy of the denotational semantics. | 207 | |
| **Translation** | | | |
| Context | Defines types and typing contexts in $\lambda^{\bigcirc}$. | 30 | |
| Term | Defines intrinsically typed terms in $\lambda^{\bigcirc}$. | 68 | |
| Translation | Formalizes the translation. | 98 | |
| **Total** | | 3286 | |

Table 1. Formalization in Agda

## 9 Related Work

This section compares our calculi with related work. Table 2 provides an overview of the syntax, type systems, and key features of these calculi.

*Let-splice bindings.* Let-splice bindings are inspired by $F^{[\![]\!]}$, the core calculus of Typed Template Haskell [Xie et al. 2022], but with several key distinctions. In $F^{[\![]\!]}$, let-splice bindings (called *splice environments*) are bound to quotations, appearing as $[\![e]\!]_\phi$, where $e$ is a quoted expression and $\phi$ is a list of splice bindings. This effectively ties splice bindings to quotations (except for top-level splices). Our calculus decouples let-splice bindings from quotations, allowing for greater flexibility. Moreover, $F^{[\![]\!]}$ contexts are flat, supporting only a single level of nesting, and does not support dependencies ($\Delta \triangleright$) as first-class types. Our calculus allows arbitrary nesting of contexts, enabling more complex variable dependencies. This nesting necessitates delayed substitutions to ensure type soundness, as discussed in §4.2. Finally, $F^{[\![]\!]}$ lacks support for code pattern matching.

*Modal logic and staging.* While temporal logic based staging [Davies 1996] has been used widely, S4 modal logic offers an alternative approach to staging [Pfenning and Davies 2001]. Specifically, the box modality $\Box A$ models *closed code expressions*, which cannot depend on the surrounding context. Thus, $\Box A$ can always be evaluated to yield a value of type $A$. This contrasts with the temporal modality $\bigcirc A$, which allows code to use variables in the surrounding context in a well-staged way. The relationship between $\lambda^{\bigcirc}$, $F^{[\![]\!]}$, and $\lambda^{\bigcirc\triangleright}$ mirrors the different derivation systems of the intuitionistic S4 logic. $\lambda^{\bigcirc}$ corresponds to Pfenning and Davies [2001]'s implicit system which uses quote and unquote operators similar to quasi-quotes, $F^{[\![]\!]}$ corresponds to the style of Bierman and de Paiva [2000], pairing an explicit substitution with the quote constructor, and $\lambda^{\bigcirc\triangleright}$ corresponds to Pfenning and Davies [2001]'s implicit system, using let-bindings for unquoting. In literature,

| | $\lambda^{\bigcirc\triangleright}, \lambda^{\bigcirc\triangleright}_{\text{pat}}$ | $\lambda^{\bigcirc}$ [Davies 1996] | $F^{[\![]\!]}$ [Xie et al. 2022] | Mœbius [Jang et al. 2022] | $\lambda^{\blacktriangle}$ [Stucki et al. 2021] | $\lambda^{\{\}}$ [Parreaux et al. 2017] |
|---|---|---|---|---|---|---|
| Quoting<br>Unquoting | $\langle\cdot\rangle$<br>$\mathbf{let}_{\Delta}\ \langle\cdot\rangle$ | **next**<br>**prev** | $[\![\cdot]\!]_{\phi}$ | box<br>let box | $\lceil\cdot\rceil$<br>$\lfloor\cdot\rfloor$ | $\lceil\cdot\rceil$<br>$\lfloor\cdot\rfloor$ |
| Code Type<br>Contextual Type | $\bigcirc$<br>$\Delta \triangleright$ | $\bigcirc$<br>– | *Code*<br>– | $\lceil\Phi \vdash^{k} \cdot\rceil$ | $\lceil\cdot\rceil$<br>– | **Code** $T$ $C$ |
| Nested Contexts | Yes | – | 1-level | Yes | – | No |
| Polymorphism | No | No | Yes | Yes | No | Subtyping |
| Pattern Matching | Yes | – | – | Yes | Yes | Yes |
| Rewrite | Yes | – | – | – | – | Yes |

Table 2. Comparison of our calculus with related work

implicit systems are sometimes referred to as Kripke-style or Fitch-style [Clouston 2018; Murase 2017; Murase et al. 2023], while explicit systems are sometimes called dual-context style [Kavvos 2020; Nanevski et al. 2008]. *Contextual modal type theory* (CMTT) [Nanevski et al. 2008] extends the S4 approach with *contextual modalities*, generalizing the $\Box A$ type to $\lceil\Phi \vdash A\rceil$, which explicitly declares a context $\Psi$ that the code may reference. Code of type $\lceil\cdot \vdash A\rceil$ is guaranteed to be closed (containing no free variables), and can thus be evaluated. Boespflug and Pientka [2011] further extend CMTT with multiple levels, modeling meta$^n$-variables. Mœbius [Jang et al. 2022] additionally supports code pattern matching. Our calculus shares similarities with Mœbius, but with important differences. First, we have different logical foundations: ours is based on temporal logic, while Mœbius generalizes from S4. Thus, our $\bigcirc A$ can still access variables from its surrounding context. Within a quotation, our system treats both meta-variables and program variables as variables at the next level, while in Mœbius, meta-variables are treated separately from program variables. Also, we separate the code modality $\bigcirc$ and the contextual modality $(\Delta \triangleright)$, while Mœbius combines them into a single modality $\lceil\Phi \vdash^{k} \cdot\rceil$. Furthermore, we use different approaches to context tracking: a value of type $(\Delta \triangleright A)$ can use variables in $\Delta$ *in addition to* those in the surrounding context. This allows dependencies that do not follow lexical scoping, which is essential for expressing unhygienic functions. It also allows context specifications to be combined with other type constructors. For instance, $(x : \mathbf{bool}^1) \triangleright (A \times \bigcirc B)$ could represent an unhygienic value of type $A$ paired with code of type $B$, where both use a variable $x$. This different logical foundation and design also allow us to use different types for pattern variables. Lastly, Mœbius supports type polymorphism; as future work, we are interested in extending our design with type polymorphism.

*Code pattern matching.* Stucki et al. [2021] present $\lambda^{\blacktriangle}$, which also supports code pattern matching. In $\lambda^{\blacktriangle}$, as mentioned in §2.5, to match a term under a lambda, the pattern variable must first be η-expanded into a function that takes the code of the bindings as arguments. As noted by Stucki et al., this only works for a simpler two-stage setting and does not yet support matching on quotes, splices, and matches themselves. Compared to $\lambda^{\blacktriangle}$'s approach, our type system avoids the need for η-expansion and allows for matching on, for example, code that itself uses quotations and splices. Squid [Parreaux et al. 2017] (and their calculus $\lambda^{\{\}}$) uses CMTT-like types and is thus closely related to Mœbius [Jang et al. 2022]. Our code rewriting construct is inspired by Squid. Hu and Pientka [2024, 2025] propose *layered modal type theory*, where the language at layer $i + 1$ can inspect and analyze code from the one at layer $i$.

## References

Andreas Abel, Guillaume Allais, Aliya Hameer, Brigitte Pientka, Alberto Momigliano, Steven Schäfer, and Kathrin Stark. 2019. POPLMark reloaded: Mechanizing proofs by logical relations. *Journal of Functional Programming* 29 (Jan. 2019),

e19. https://doi.org/10.1017/S0956796819000170

Emmanuel Suárez Acevedo and Stephanie Weirich. 2023. Making Logical Relations More Relatable (Proof Pearl). *arXiv preprint arXiv:2309.15724* (2023).

Thorsten Altenkirch, Martin Hofmann, and Thomas Streicher. 2005. Categorical reconstruction of a reduction free normalization proof. In *Category Theory and Computer Science*. Springer, Berlin, Germany, 182–199.

Kenichi Asai, Luminous Fennell, Peter Thiemann, and Yang Zhang. 2014. A type theoretic specification of partial evaluation. In *Proceedings of the 16th International Symposium on Principles and Practice of Declarative Programming*. 57–68.

Eli Barzilay, Ryan Culpepper, and Matthew Flatt. 2011. Keeping it clean with syntax parameters. *Proc. Wksp. Scheme and Functional Programming* (2011).

G. M. Bierman and V. C. V. de Paiva. 2000. On an Intuitionistic Modal Logic. *Studia Logica* 65, 3 (Aug. 2000), 383–416.

Mathieu Boespflug and Brigitte Pientka. 2011. Multi-level Contextual Type Theory. In *Proceedings Sixth International Workshop on Logical Frameworks and Meta-languages: Theory and Practice, LFMTP 2011, Nijmegen, The Netherlands, August 26, 2011 (EPTCS, Vol. 71)*, Herman Geuvers and Gopalan Nadathur (Eds.). 29–43.

Cristiano Calcagno, Walid Taha, Liwen Huang, and Xavier Leroy. 2003. Implementing multi-stage languages using ASTs, gensym, and reflection. In *International Conference on Generative Programming and Component Engineering*. Springer, 57–76.

William Clinger. 1991. Hygienic macros through explicit renaming. *ACM SIGPLAN Lisp Pointers* 4, 4 (1991), 25–28.

Ranald Clouston. 2018. Fitch-style modal lambda calculi. In *Foundations of Software Science and Computation Structures: 21st International Conference, FOSSACS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14–20, 2018. Proceedings 21*. Springer, 258–275.

Rowan Davies. 1996. A temporal-logic approach to binding-time analysis. In *Proceedings 11th Annual IEEE Symposium on Logic in Computer Science*. IEEE, 184–195.

Jason ZS Hu and Brigitte Pientka. 2024. Layered modal type theory: Where meta-programming meets intensional analysis. In *European Symposium on Programming*. Springer, 52–82.

Jason ZS Hu and Brigitte Pientka. 2025. A Dependent Type Theory for Meta-programming with Intensional Analysis. *Proceedings of the ACM on Programming Languages* 9, POPL (2025), 416–445.

Junyoung Jang, Samuel Gélineau, Stefan Monnier, and Brigitte Pientka. 2022. Mœbius: metaprogramming using contextual types: the stage where system f can pattern match on itself. *Proceedings of the ACM on Programming Languages* 6, POPL (Jan. 2022), 1–27.

Manohar Jonnalagedda, Thierry Coppey, Sandro Stucki, Tiark Rompf, and Martin Odersky. 2014. Staged Parser Combinators for Efficient Data Processing. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications* (Portland, Oregon, USA) *(OOPSLA '14)*. Association for Computing Machinery, New York, NY, USA, 637–653. https://doi.org/10.1145/2660193.2660241

Georgios Alexandros Kavvos. 2020. Dual-context calculi for modal logic. *Logical Methods in Computer Science* 16 (2020).

G. A. Kavvos. 2024. Two-Dimensional Kripke Semantics I: Presheaves. *Schloss Dagstuhl – Leibniz-Zentrum für Informatik* (2024), 14:1–14:23. https://doi.org/10.4230/LIPIcs.FSCD.2024.14

Oleg Kiselyov. 2014. The Design and Implementation of BER MetaOCaml. In *Functional and Logic Programming*, Michael Codish and Eijiro Sumii (Eds.). Springer International Publishing, Cham, 86–102. https://doi.org/10.1007/978-3-319-07151-0_6

Oleg Kiselyov, Aggelos Biboudis, Nick Palladinos, and Yannis Smaragdakis. 2017. Stream fusion, to completeness. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. 285–299.

Eugene Kohlbecker, Daniel P Friedman, Matthias Felleisen, and Bruce Duba. 1986. Hygienic macro expansion. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*. 151–161.

Kensuke Kojima and Atsushi Igarashi. 2011. Constructive linear-time temporal logic: Proof systems and Kripke semantics. *Information and Computation* 209, 12 (Dec. 2011), 1491–1503.

John C. Mitchell and Eugenio Moggi. 1991. Kripke-style models for typed lambda calculus. *Annals of Pure and Applied Logic* 51, 1 (March 1991), 99–124.

Yuito Murase. 2017. Kripke-style contextual modal type theory. *Work-in-progress report at Logical Frameworks and Meta-Languages* (2017).

Yuito Murase, Yuichi Nishiwaki, and Atsushi Igarashi. 2023. Contextual Modal Type Theory with Polymorphic Contexts. In *Programming Languages and Systems*. Springer, Cham, Switzerland, 281–308. https://doi.org/10.1007/978-3-031-30044-8_11

Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. 2008. Contextual modal type theory. *ACM Trans. Comput. Logic* 9, 3, Article 23 (June 2008), 49 pages. https://doi.org/10.1145/1352582.1352591

Lionel Parreaux, Antoine Voizard, Amir Shaikhha, and Christoph E. Koch. 2017. Unifying analytic and statically-typed quasiquotes. *Proceedings of the ACM on Programming Languages* 2, POPL (Dec. 2017), 1–33.

Frank Pfenning and Rowan Davies. 2001. A judgmental reconstruction of modal logic. *Mathematical Structures in Computer Science* 11, 4 (Aug. 2001), 511–540.

Tim Sheard and Simon Peyton Jones. 2002. Template Meta-Programming for Haskell. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell* (Pittsburgh, Pennsylvania) *(Haskell '02)*. Association for Computing Machinery, New York, NY, USA, 1–16. https://doi.org/10.1145/581690.581691

Nicolas Stucki, Aggelos Biboudis, and Martin Odersky. 2018. A practical unification of multi-stage programming and macros. In *Proceedings of the 17th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*. 14–27.

Nicolas Stucki, Jonathan Immanuel Brachthäuser, and Martin Odersky. 2021. Multi-stage programming with generative and analytical macros. In *ACM Conferences*. Association for Computing Machinery, New York, NY, USA, 110–122.

Walid Taha and Tim Sheard. 2000. MetaML and multi-stage programming with explicit annotations. *Theoretical computer science* 248, 1-2 (2000), 211–242.

The Agda Community. 2024. *Agda Standard Library*. https://github.com/agda/agda-stdlib

Philip Wadler, Wen Kokke, and Jeremy G. Siek. 2022. *Programming Language Foundations in Agda*.

Fei Wang, Daniel Zheng, James Decker, Xilun Wu, Grégory M Essertel, and Tiark Rompf. 2019. Demystifying differentiable programming: Shift/reset the penultimate backpropagator. *Proceedings of the ACM on Programming Languages* 3, ICFP (2019), 1–31.

Jamie Willis, Nicolas Wu, and Matthew Pickering. 2020. Staged selective parser combinators. *Proc. ACM Program. Lang.* 4, ICFP (2020), 120:1–120:30. https://doi.org/10.1145/3409002

Ningning Xie, Matthew Pickering, Andres Löh, Nicolas Wu, Jeremy Yallop, and Meng Wang. 2022. Staging with class: a specification for typed template Haskell. *Proceedings of the ACM on Programming Languages* 6, POPL (Jan. 2022), 1–30.

Ningning Xie, Leo White, Olivier Nicole, and Jeremy Yallop. 2023. MacoCaml: Staging Composable and Compilable Macros. *MacoCaml: Staging Composable and Compilable Macros (Artifact)* 7, ICFP (Aug. 2023), 604–648. https://doi.org/10.1145/3607851

Jeremy Yallop. 2017. Staged Generic Programming. *Proc. ACM Program. Lang.* 1, ICFP, Article 29 (Aug. 2017), 29 pages. https://doi.org/10.1145/3110273