Typed Meta-Programming with Splice Variables

TSUNG-JU CHIANG

Typed meta-programming catches meta-programming errors early by checking them at definition time. This paper introduces $\lambda^{\bigcirc\triangleright}$, a typed meta-programming language that uses nested context design and temporal-style staging to track binding times and variable dependencies. The system supports a range of meta-programming idioms, including explicit splice definitions, unhygienic macros and analytic macros. We formalize the language in Agda, prove its safety propertes, define a denotational semantics to clarify the meaning of its types, and show its soundness and completeness with respect to constructive linear-time temporal logic through type-preserving translations. We compare our approach to contextual modal type theory-based systems, providing insights into their similarities and differences.

ACM Reference Format:
Tsung-Ju Chiang. 2025. Typed Meta-Programming with Splice Variables. 1, 1 (January 2025), 34 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Meta-programming allows programs to analyze and generate code at compile time, enabling flexible abstractions while reducing runtime overhead. Typed meta-programming integrates type and scope checking of code expressions into the type system, allowing meta programs to be specified with precise types and checked at definition time. This makes meta-programming more predictable, helping catch errors early and improving the overall programming experience.

A popular approach to typed meta-programming is based on *temporal logic* [Davies 1996], which has been used in various languages including OCaml [Kiselyov 2014; Xie et al. 2023], Scala [Stucki et al. 2018, 2021], and Haskell [Sheard and Jones 2002]. The temporal "next" operator $\bigcirc$ acts as a type constructor for typed code expressions, accompanied by *quoting* and *splicing* operators similar to Lisp's quasi-quote mechanism. This allows meta-programs to be written in the same language as the programs they generate, making them more intuitive and easier to reason about.

While the quote-and-splice syntax offers a powerful mechanism for meta-programming, it can be restrictive in certain cases. For example, precisely controlling the evaluation order of splice expressions can be challenging. Recently, Typed Template Haskell [Xie et al. 2022] addressed this issue by translating splices into a sequence of definitions within a core calculus, allowing the evaluation order of splice expressions to be explicitly specified. However, the core calculus is intended as an intermediate compilation target, not for direct use by the programmers.

In this paper, we introduce *let-splice bindings*, a language construct that explicitly defines splice expressions within a surface language. Unlike the quote-and-splice mechanisms, let-splice bindings offer precise control over splice evaluation order. Compared to Xie et al. [2022], let-splice bindings are more flexible and enable the sharing and reuse of splice computations across different contexts. Our design incorporates a novel type system that tracks *variable dependencies* of splice definitions, allowing splice expressions to be defined in a context where certain variables are not yet available.

Author's Contact Information: Tsung-Ju Chiang.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM XXXX-XXXX/2025/1-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Inspired by *Contextual Model Type Theory* (CMTT) [Jang et al. 2022; Nanevski et al. 2008], the type system associates let-splice bindings with a typing context to capture these variable dependencies, ensuring well-typedness of splice definitions. When a splice variable is used, the corresponding dependencies must be provided. Those contexts can also be nested to specify more complex dependencies. While the type system shares similarities with CMTT, it diverges in its logical foundation (i.e. temporal logic) as well as its treatment of variable dependency tracking; a detailed comparison with related work is provided in Section 10. Furthermore, as we will show, our design serves as a basis for more advanced meta-programming features, such as *unhygienic macros* [Barzilay et al. 2011] and *code pattern matching* [Stucki et al. 2021], both of which require similar mechanisms for managing variable dependencies. Our system naturally supports these features, demonstrating its expressiveness and potential for future language extensions.

More specifically, we present two calculi: $\lambda^{\bigcirc\triangleright}$, a temporal-style multi-stage calculus supporting let-splice bindings, featuring a novel contextual modality ($\Delta \triangleright$) for managing variable dependencies; and $\lambda^{\bigcirc\triangleright}_{\text{pat}}$, an extension of $\lambda^{\bigcirc\triangleright}$ which seamlessly integrates code pattern matching and code rewriting. For both calculi, we define a small-step operational semantics and a denotational semantics based on a Kripke-style model. We prove soundness and completeness of our type system with respect to *constructive linear-time temporal logic* [Kojima and Igarashi 2011]. Both calculi are fully formalized in the Agda proof assistant, along with all the proofs. Each formalized definition and property is marked with a clickable icon, linking to the corresponding Agda definition.

We offer the following contributions:

(1) Section 3 and 4 present a novel calculus $\lambda^{\bigcirc\triangleright}$ with let-splice bindings. It features dependency tracking with nested typing context, a temporal-style code type for code expressions, and a separate contextual modality for managing variable dependencies.
(2) Section 5 provides a type-preserving translation from $\lambda^{\bigcirc\triangleright}$ to constructive linear-time temporal logic [Kojima and Igarashi 2011] and then to $\lambda^{\bigcirc}$ [Davies 1996], offering insight into their relationship.
(3) Section 6 introduces $\lambda^{\bigcirc\triangleright}_{\text{pat}}$, an extension of $\lambda^{\bigcirc\triangleright}$ that allows for pattern matching on code, allowing for inspection and rewriting of code fragments.
(4) Section 7 defines a denotational semantics for $\lambda^{\bigcirc\triangleright}$ and $\lambda^{\bigcirc\triangleright}_{\text{pat}}$ using a Kripke-style model.
(5) We formalize $\lambda^{\bigcirc\triangleright}$ and $\lambda^{\bigcirc\triangleright}_{\text{pat}}$ in the Agda proof assistant, and establish key properties and theorems including progress and preservation.

Lastly, section 10 compares our approach to related work, including CMTT-based calculi [Jang et al. 2022], highlighting the differences in logical foundations and variable dependency tracking.

2 Motivation and Examples

In this section we outline the design of our calculus, and then demonstrate its expressiveness through three examples: reuse of splice variables, unhygienic macros for anaphoric conditionals, and pattern matching on code.

2.1 Staged Power Function

A classic example of code generation is the staged power function. Given a quoted expression `<e>` and a known integer `n`, this function generates the expression `<e * ... * e * 1>` with n repeated multiplications, avoiding recursion and thus reducing the overhead for any specific `e`. An implementation using the traditional quote-and-splice syntax can be written as follows:

```
let power : int^1 code → int^0 → int^1 code
let rec power e n =
```

```
    if n == 0 then <1>
    else <$(e) * $(power e (n - 1))>
  let power5 x = $(power <x> 5)    -- generates (x * x * x * x * x * 1)
```

where a quotation `<expr>` represents the code fragment of the expression, and a splice `$(expr)` extracts out the expression from the code fragment. Following Typed Template Haskell [Xie et al. 2022], `power5` uses *top-level splices* (i.e. splices without surrounding quotations) for compile-time code generation. For clarity, we annotate base types with superscripts to indicate their evaluation stages, where 0 represents compile-time and 1 represents runtime. For example, `int⁰` denotes a compile-time integer and `int¹` denotes a runtime integer. Code expressions have a `code` type; therefore, `int¹ code` represents a quoted expression of a runtime integer.

While the quote-and-splice syntax is useful, it can also introduce complexities. Specifically, the evaluation order of splice expressions can be unclear. For example, evaluating the expression `(e1 <e2 $(e3)>)` will first evaluate `e1` and then `e3`, but not `e2`. This requires a *level-indexed* reduction relation [] that keeps track of the relative number of quotations and splices during evaluation, adding complexity to both the implementation and the meta-theory. Moreover, in the context of compile-time code generation, it raises the question of how to evaluate nested splices, e.g. `$(e1) $($(e2))`, where `e1` appears first, but `e2` has more splices. Typed Template Haskell will evaluate `e2` before `e1`, while both Scala [Stucki et al. 2018] and OCaml [Xie et al. 2023] disallow nested splices.

Our design introduces novel *let-splice bindings* that make splice definitions explicitly. In particular, an implementation of the staged power function using our syntax can be written as:

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

In our calculus, the splicing operation is replaced instead by let-splice bindings (`let$`), which bind a code expression to a *splice variable*. In this case, the splice variables `s1` and `s2` represent the splice of `e` and of `power e (n - 1)`, respectively. Since both variables represent splice expressions, they can be directly used as `s1 * s2` within the quotation. Formally, quotations, let-splices, and splice variables are all managed by *levels*. As shown in this example, splice variables with explicit dependencies clarify the order in which splices are computed.

In this particular case, the two splice definitions do not capture any free variables. More interestingly, definitions can be annotated with a list of *variable dependencies*. This provides flexibility since splice expressions can depend on values that are only available when the splice variable is used. For example, we have:

```
  let$ s3 : (x : int¹ ⊢ int¹) = power <x> 5  -- lifted, with x as dependency
  let power5 x = s3 with x = x
```

As the original top-level splice `$(power <x> 5)` refers to the variable `x`, the splice variable `s3` is given type `(x : int¹ ⊢ int¹)`, allowing `x:int¹` to be used within its definition. When using a splice variable like `s3`, the syntax `(s3 with x = x)` provides a *delayed substitution*. This allows us to replace the variable dependencies with concrete values. For clarity, we explicitly write out all substitutions in the examples. In practice, a compiler could simply capture dependencies from the context, so entries like `x = x` can be omitted. More generally, we can write any expression `e` in `(s3 with x = e)`.

Notably, types like `(x : int¹ ⊢ int¹)` are first-class. Therefore, we can have dependencies in normal let definitions:

```
let w : (x : string¹; y : int¹ ⊢ int⁰) = e
```

This binds `w` to expression `e`, which depends on `x` and `y` and produces a value of type `int⁰`.

Moreover, function arguments can also be declared with dependencies:

```
val f : (x : string¹; y : int¹ ⊢ int⁰) ⟶ int⁰
let f z = (z with x = "hello"; y = 42)
```

Here, `f` takes an argument with dependencies `x` and `y`, and uses it with `x` bound to `"hello"` and `y` bound to `42`. We can then write, for example, `f w`.

Furthermore, dependencies can be nested, allowing splices to depend on other splices and effectively enabling nested splices:

```
let z : (s : (x : int¹ ⊢ string¹) ⊢ string¹ code) = <s with x = 42>
```

2.2 Reuse of Splice Variables

Consider the following meta-program, where `f : int¹ code ⟶ int¹ code`:

```
<fun x ⟶ $(f <x>) + $(f <x>)>
```

This program generates a function that applies `f` to its argument `x` twice and adds the results. For example, given `f y = <$(y) + 1>`, the program generates:

```
<fun x ⟶ (x + 1) + (x + 1)>
```

However, in this case, the two splices in the original computation are evaluated sequentially, leading to duplicated computations of `$(f <x>)`.

To eliminate duplicated computations, we can pre-compute the result of the splice expression:

```
let s = <fun z ⟶ $(f <z>)> in
<fun x ⟶ $(s) x + $(s) x>
```

Unfortunately, while this avoids redundant computations, it introduces two unnecessary beta-redexes in the generated code:

```
<fun x ⟶ ((fun z ⟶ z + 1) x) + ((fun z ⟶ z + 1) x)>
```

In our calculus, we can easily reuse splice variables without introducing unnecessary abstractions. Specifically, we can express the original computation as:

```
let$ s : (z : int¹ ⊢ int¹) = f <z> in
<fun x ⟶ (s with z = x) + (s with z = x)>
```

Here, `let$` declares a splice variable `s` with a dependency on `z: int¹`. The expression `f <z>` is evaluated symbolically, which can refer to variable `z`. The `(s with z = x)` syntax then directly substitutes `z` with `x`. In this case, the splice expression is only evaluated once, and the generated code is the desired `<fun x ⟶ (x + 1) + (x + 1)>`. In other words, the program achieves both computational efficiency and clean generated code. Moreover, we can also reuse the same splice variable and provide different substitutions, e.g. `(s with z = x) + (s with z = (x + 2))`.

2.3 Unhygienic Macros

Hygienic macros, whose expansion is guaranteed to not accidentally capture variables, are well established, but can sometimes be insufficient. Barzilay et al. [2011] observed that there are common kinds of unhygienic macros that are practically useful. One common kind of them that implicitly introduce bindings are *"notoriously difficult to deal with"*. Two such well-known example are 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.

In this work, we use *unhygienic macros* to mean functions whose arguments may depend on additional later-stage variables that are to be supplied when the function is used, and *unhygienic values* as its first-class counterpart, i.e. values that may depend on additional later-stage variables.

To demonstrate how unhygienic macros work in our calculus, we consider anaphoric conditionals as an example. Concretely, we would like to create a "macro" `aif`, with which we can write the following program:

```
aif <big-long-calculation> <foo it> <bar it>
```

Here, both then- and else-branches can refer to the variable `it` to stand for the result of the `big-long-calculation`. Specifically, the program will expand to:

```
<let it = big-long-calculation in
 if it then (foo it) else (bar it)>
```

In a statically typed language, it is obvious that `it` will stand for `True` in the then-branch and `False` in the else-branch, so the macro is less useful. In languages like Scheme, however, the value of `it` is not necessarily `False` in the else-branch.

In our calculus, we can define `aif` with the following function type signature, where the second and third arguments are declared with an additional dependency on variable `it`:

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

When applied, the type signature of `aif` informs the type checker to introduce a new variable `it` into the scope of the second and third arguments (e.g. `foo` and `bar`), allowing them to directly refer to it. Given this 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 = it in
  let$ s3 : (it : bool¹ ⊢ 'a¹) = bar with it = it in
  <let it = s1 in
   if it then (s2 with it = it)
   else (s3 with it = it)>
```

The function takes three code arguments, `cond`, `foo`, and `bar`, with the latter two depending on an additional variable `it`. First, the arguments are unwrapped using `let$`, binding them to splice variables `s1`, `s2`, and `s3` for use inside the quotation. The dependencies of `foo` and `bar` are explicitly rebound as dependencies of their corresponding splice variables. Then, the output code expression is constructed using a quotation. The splice variables indicate where each piece of code should be inserted, while the `with` syntax specifies the desired binding structure. Notably, while the code expressions for both branches will get generated, only the selected branch will be evaluated depending on the value of `it`.

By supporting unhygienic macros, our calculus can express a wider range of meta-programming patterns, including those that intentionally "break" lexical scoping in a well-typed way.

2.4 Pattern Matching on Code

So far we have focused on *generative* meta-programming, where smaller code fragments are combined to create larger ones, as seen in the `power` and `aif` examples. In contrast, *analytic macros* [Ganz et al. 2001; Stucki et al. 2021] can inspect the content of or take apart code fragments, and enable useful techniques like code rewriting for optimization.

In staging calculi, this is often realized through pattern matching on code [Jang et al. 2022; Parreaux et al. 2017]. However, typing code patterns is much more complicated, especially since matching under a binder can yield a code expression that contains the bound variable inaccessible outside of its scope.

We extend our calculus with support for pattern matching on code, which allows us to inspect the structure of code fragments. Interestingly, we show that pattern matching can be naturally supported with variable dependencies.

As an example, consider a program that computes the partial derivative of an arithmetic expression as a code fragment. Specifically, the following function `partial` recursively matches the input argument `e`, generating code for its partial derivative with respect to an variable `var`:

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

The function uses `match$` to perform pattern matching on code. Code patterns distinguish two kinds of variables: *pattern variables* like `g` and `h` match any code expression, and variables like `` `var``, `` `+`` and `` `*`` match those specific identifiers. This illustrates how our calculus supports analytic macros naturally by combining pattern matching and unhygienic variable bindings.

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

```
let$ df : (x y : int^1 ⊢ int^1) = (partial with var = x) <x * y + 1>
```

generates `<(1 * y + x * 0) + 0>` for any given `x` and `y`. We can use `df` by providing specific `x` and `y`, e.g. `df with x = 1, y = 2`.

Dependency tracking becomes crucial when matching under a binder. For example, consider computing the partial derivative of a let expression `let (y : int) = f in g`. Using the chain rule, the derivative can be expressed as:

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

This can be implemented as follows:

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

Here, `g` is matched as a splice variable with an additional dependency on `y`. `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.

2.5 Code Rewriting

Another useful analytic feature is *code rewriting* [Parreaux et al. 2017], which replaces all occurrences of a pattern in a target expression with a replacement expression. In our extended calculus, code rewriting can be expressed as:

```
rewrite p as e_replacement in e_target
```

where `p` is a code pattern and `e_replacement` and `e_target` are code expressions. This feature is especially useful for optimizing code that are programmatically generated, which often contain redundant code that can be simplified. For example, consider the code generated by the `partial` example above:

```
<(1 * y + x * 0) + 0>
```

The `1 *`, `* 0`, and `+ 0` are redundant. We can use code rewriting to simplify the expression:

```
let$ df_opt : (x y : int¹ ⊢ int¹) =
   rewrite (`1 `+ z) as <z> in
   rewrite (z `+ `0) as <z> in
   rewrite (z `+ `0) as <z> in
   rewrite (z `* `0) as <0> in
   <df with x = x; y = y>
```

which simplifies the expression to `<y>`.

3 Core Syntax and Typing

We introduce $\lambda^{\bigcirc\triangleright}$, a typed lambda calculus with quotations, let-quote bindings, and unhygienic functions. The full syntax of $\lambda^{\bigcirc\triangleright}$ is summarized in fig. 1.

3.1 Types and Typing Contexts

A key design of $\lambda^{\bigcirc\triangleright}$ is the use of *nested typing contexts* to track variable dependencies and stage levels. They enable unhygienic macros and serves as the foundation for supporting code pattern matching, which will be introduced in section 6.

3.1.1 Contexts. Contexts are defined by the grammar:

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

Each variable x in a context is associated with:

- A *context* Δ, which tracks the additional variable dependencies of x. When Δ is empty, we write $x : A^n$ as shorthand for $x : [\cdot \vdash^n A]$.
- A *stage level* n which specifies the stage of computation at which x can be accessed. The stage levels carry the same meaning as in Davies's $\lambda^{\bigcirc}$: higher values correspond to later stages, such as runtime, while lower values correspond to earlier stages, such as compile time.
- A *type* A, which describes the kind of value x represents.

3.1.2 Types. Types are defined by the 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 Δ. When Δ is empty, these are just normal functions, and 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, as in $\lambda^{\bigcirc}$.
- $\Delta \triangleright A$ represents unhygienic values of type A with dependencies Δ. This type is dual to the unhygienic function type, in the sense that $(\Delta \triangleright A) \to B$ is equivalent to $[\Delta \vdash A] \to B$. We keep $[\Delta \vdash A] \to B$ in the syntax as it allows us to express unhygienic macros more naturally.

3.1.3 Well-stagedness. We consider only *well-staged* contexts and types in our typing rules. A context Γ is well-staged at level n, if every entry $x : [\Delta \vdash^m A]$ in Γ meets two conditions:

- $m \geq n$, and
- Δ and A are well-staged at level m.

In other words, stage levels can only stay the same or increase as the nesting of [] becomes deeper.

For types, well-stagedness is defined as follows:

- **bool** is well-staged at any level.
- $[\Delta \vdash A] \to B$ is well-staged at level n, if
  - A and B are well-staged at level n, and
  - Δ is well-staged at level $n + 1$.
- $\bigcirc A$ is well-staged at level n if A is well-staged at level $n + 1$.
- $\Delta \triangleright A$ is well-staged at level n if A is well-staged at level n and Δ is well-staged at level $n + 1$.

Staging of $\bigcirc A$ reflects that quotations contain expressions belonging to the next stage. Staging of $[\Delta \vdash A] \to B$ and $\Delta \triangleright A$ captures the concept of *unhygienic values*: values that depend on later-stage variables and compute with them symbolically.

Note that in $\lambda^{\bigcirc}$ staging of types is implicit and relative to the context, while in $\lambda^{\bigcirc\triangleright}$ staging is explicit and absolute. This is more of a matter of presentation than a fundamental difference: we could have staged the Δ's in our types relatively to achieve relative staging, but we chose to make staging explicit to simplify the presentation of our rules. We discuss the trade-off between the two approaches in section 8.1.

3.1.4 Stage Annotation. When the staging level isn't clear from the context, we use superscripts Γ^n and A^n to indicate their levels. This notation binds more tightly than type constructors and the comma "," in contexts. Using this notation, we can annotate the grammar as follows:

$$\begin{array}{rcl} \Gamma^n, \Delta^n & ::= & \cdot \mid \Gamma^n, x : [\Delta^m \vdash^m A^m] \quad (m \geq n) \\ A^n, B^n & ::= & \mathbf{bool} \mid [\Delta^{n+1} \vdash A^n] \to B^n \mid \bigcirc A^{n+1} \mid \Delta^{n+1} \rhd A^n \end{array}$$

3.1.5 Restriction. The *restriction* of a context Γ to level n, written $\Gamma{\restriction}_n$, removes all variables in Γ with levels less than n. Restriction preserves well-stagedness: if Γ is well-staged at some level n, then $\Gamma{\restriction}_m$ is well-staged at level m for any m.

3.2 Expressions

Next, we define the syntax of expressions in $\lambda^{\bigcirc\rhd}$. The grammar is as follows:

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

x_σ represents a variable x paired with a *delayed substitution* σ. The delayed substitution maps dependencies of x to variables or expressions in the current context, and is applied when x is replaced with a concrete expression; the formal definition of substitution is given in Section 4.1. $\lambda_\Delta x : A.\ e$ defines an unhygienic function whose argument x depends on variables in Δ; $e_1\ e_2$ applies a function e_1 to an argument e_2. $\langle e \rangle$ quotes an expression e into a code expression; $\mathbf{let}_\Delta \langle x : A \rangle = e_1\ \mathbf{in}\ e_2$ unquotes a code expression e_1 that can depend on variables in Δ, introducing a next-stage variable x with dependencies Δ, which can be used inside quotations in e_2. $\mathbf{wrap}_\Delta e$ wraps an expression e with dependencies Δ, allowing it to symbolically compute with the variables; $\mathbf{let\ wrap}_\Delta\ x : A = e_1\ \mathbf{in}\ e_2$ unwraps a wrapped expression e_1, introducing a current-stage variable x with dependencies Δ, directly usable in e_2. In all cases, the subscripted Δ is staged one level higher than the current context, and can be arbitrarily nested.

Substitutions 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.

Table 1 summarizes the mapping between concrete and abstract syntax.

3.3 Typing Rules

The typing judgment $\Gamma \vdash^n e : A$ assigns a type A to an expression e under the context Γ, at stage level n. The following assumptions apply:

(1) All contexts contain distinct variables.
(2) Both the context Γ and the type A are well-staged at level n.

The rules are defined as follows:

Concrete Syntax	Abstract Syntax
`x with y = e1; z = e2`	$x_{y\mapsto e_1, z\mapsto e_2}$
`fun x : (Δ ⊢ A) ⟶ e`	$\lambda_\Delta x : A.\ e$
`let x : (Δ ⊢ A) = e1 in e2`	$(\lambda_\Delta x : A.\ e_2)\ e_1$
`let$ x : (Δ ⊢ A) = e1 in e2`	$\mathbf{let}_\Delta \langle x : A\rangle = e_1\ \mathbf{in}\ e_2$
`f x`	$f\ x$
`<e>`	$\langle e\rangle$
`true` `false`	$\mathbf{true}\ \mathbf{false}$
`if e1 then e2 else e3`	$\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3$

Table 1. Mapping between concrete and abstract syntax

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

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

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

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

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

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

Judgment $\Gamma \vdash \sigma : \Gamma'$ types a substitution σ that maps variables in Γ' to variables or expressions in Γ.

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

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

Rule VarSubst defines how variables may be used in expressions. A variable $x : [\Delta \vdash^n A]$ can only be used at level n, and must be accompanied by a substitution σ that maps each dependency in Δ to an expression with the corresponding type under Γ. If Δ is empty, as is the case for normal variables, then σ is also empty.

The rules True, False, and If are standard.

For the unhygienic function type $[\Delta \vdash A] \to B$, rule CtxAbs creates an unhygienic function, allowing its argument to refer to variables in a context Δ. Rule CtxApp applies an unhygienic function, extending the context with variables from Δ to type-check the argument.

Rule Quote quotes an expression into a code expression. The rule increases the level to $n+1$ and updates the context to $\Gamma\upharpoonright_{n+1}$ to type-check the quoted expression e. If e has type A, $\langle e \rangle$ has the code type $\bigcirc A$. Rule LetQuote unquotes a code expression and binds it to a variable x at the next level. This variable represents an open code fragment that may additionally depend on variables in a context Δ.

Rule Wrap wraps an expression with dependencies Δ, producing an unhygienic value type $\Delta \rhd A$, Note that unlike the $\bigcirc$ type, $\Delta \rhd$ does not change the stage level of the expression. Rule LetWrap unwraps a wrapped expression and binds it to a contextual variable $x : [\Delta \vdash^n A]$.

Typing rules for substitutions ensure that the substitution σ provides mappings for corresponding variables in Γ'. We call Γ' the domain of σ and Γ the codomain. 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 Γ is restricted to level m and is then appended with Δ to type-check e.

4 Dynamics

We describe a small-step, call-by-value operational semantics for $\lambda^{\bigcirc\rhd}$, based on term substitution.

4.1 Substitution

Substitution is mutually defined on typed expressions and substitutions. Given a substitution $\Gamma_2 \vdash \sigma : \Gamma_1$, $e[\sigma]$ applies σ to a typed expression $\Gamma_1 \vdash^n e : A$, while $\sigma_1[\sigma]$ applies σ to all entries of a type substitution $\Gamma_1 \vdash \sigma_1 : \Delta$, computing their composition. For $e[\sigma]$, The only non-trivial case is the variable case, which will be discussed in detail. All the other cases only involve weakening or restricting the substitution and recursing into the sub-expressions. For $\sigma_1[\sigma]$, the function recursively processes all entries of σ_1.

We introduce the following notations for substitutions: $\sigma\upharpoonright_n$ restricts the domain of σ by removing entries with levels smaller than n, similar to context restriction. id_Γ denotes the identity substitution on Γ, i.e. $x_1 \mapsto x_1, x_2 \mapsto x_2 \ldots$ for $x_i \in \Gamma$. They have the following types:

- If $\Gamma_2 \vdash \sigma : \Gamma_1$ then $\Gamma_2\upharpoonright_n \vdash \sigma\upharpoonright_n : \Gamma_1\upharpoonright_n$.
- $\Gamma \vdash \mathrm{id}_\Gamma : \Gamma$.

Given a typed substitution, we write $x^m_\Delta \mapsto e$ if the substitution entry is typed $x : [\Delta \vdash^m A]$ in the domain of the substitution and maps to e.

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

$$
\begin{aligned}
(x_{\sigma_1})[\sigma] &\coloneqq \begin{cases} y_{(\sigma_1[\sigma])} & \text{if } \sigma(x) = y, \\ e[\mathrm{id}_{\Gamma_2\upharpoonright_m}, \sigma_1[\sigma]] & \text{if } \sigma(x) = e. \end{cases} \\
(\mathbf{true})[\sigma] &\coloneqq \mathbf{true} \\
(\mathbf{false})[\sigma] &\coloneqq \mathbf{false} \\
(\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3)[\sigma] &\coloneqq \mathbf{if}\ e_1[\sigma]\ \mathbf{then}\ e_2[\sigma]\ \mathbf{else}\ e_3[\sigma] \\
(\lambda x : A.\ e)[\sigma] &\coloneqq \lambda x : A.\ e[\sigma, x \mapsto x] \\
(e_1\ e_2)[\sigma] &\coloneqq e_1[\sigma]\ e_2[\sigma] \\
(\langle e \rangle)[\sigma] &\coloneqq \langle e[\sigma\upharpoonright] \rangle \\
(\mathbf{let}_\Delta \langle x : A \rangle = e_1\ \mathbf{in}\ e_2)[\sigma] &\coloneqq \mathbf{let}_\Delta \langle x : A \rangle = (e_1[\sigma, \mathrm{id}_\Delta])\ \mathbf{in}\ (e_2[\sigma, x \mapsto x])
\end{aligned}
$$

$$(\mathbf{wrap}_\Delta\, e)[\sigma] \coloneqq \mathbf{wrap}_\Delta(e[\sigma, \mathrm{id}_\Delta])$$
$$(\mathbf{let\,wrap}_\Delta\; x : A = e_1 \;\mathbf{in}\; e_2)[\sigma] \coloneqq \mathbf{let\,wrap}_\Delta\; x : A = (e_1[\sigma]) \;\mathbf{in}\; (e_2[\sigma, x \mapsto x])$$

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

$$(\cdot)[\sigma] \coloneqq \cdot$$
$$(\sigma_1, x \mapsto y)[\sigma] \coloneqq \sigma_1[\sigma], x \mapsto \sigma(y)$$
$$(\sigma_1, x^m_\Delta \mapsto e)[\sigma] \coloneqq \sigma_1[\sigma], x \mapsto e[\sigma\restriction_m, \mathrm{id}_\Delta]$$

4.1.1 Termination. The substitution functions defined above is not structurally recursive on e by its definition, so it's not immediately obvious whether the function is total. The problematic case is the second case of $(x_{\sigma_1})[\sigma]$:

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

Here, the term e is not a subterm of x_{σ_1} but rather an element of the substitution σ. Therefore, we cannot argue for termination based solely on the size of the input expression. To prove that substitution terminates and is thus well-defined, we define a depth measure on typed substitutions and use it in additon to the size of the expression to show termination. From the definition of substitution, we observe that the mesure must decrease in the problematic case and be preserved under restriction and weakening. These observations motivate the following definitions:

$\boxed{\mathrm{depth}(\Gamma)}$ *(Context Depth)*

$$\mathrm{depth}(\cdot) \coloneqq 0$$
$$\mathrm{depth}(\Gamma, x : [\Delta \vdash^m A]) \coloneqq \mathrm{depth}(\Gamma) \sqcup (\mathrm{depth}(\Delta) + 1)$$

$\boxed{\mathrm{depth}(\sigma)}$ *(Substitution Depth)*

$$\mathrm{depth}(\cdot) \coloneqq 0$$
$$\mathrm{depth}(\sigma, x \mapsto y) \coloneqq \mathrm{depth}(\sigma)$$
$$\mathrm{depth}(\sigma, x^m_\Delta \mapsto e) \coloneqq \mathrm{depth}(\sigma) \sqcup (\mathrm{depth}(\Delta) + 1)$$

Preservation of depth is trivial, because renamings are simply not counted. For decrement, we have the following lemma:

Lemma 4.1 (Substitution depth decreases). *Let $\Gamma_1 \vdash \sigma_1 : \Delta$ and $\Gamma_2 \vdash \sigma : \Gamma_1$. If $x^m_\Delta \mapsto e \in \sigma$ then*

$$\mathrm{depth}(\sigma_1[\sigma]) \leq \mathrm{depth}(\Delta) < \mathrm{depth}(\sigma).$$

These together show that substitution is well-defined. In additon, substitution preserves typing, as stated in the following lemma:

Lemma 4.2 (Substitution). *Given $\Gamma_2 \vdash \sigma : \Gamma_1$,*

- *if $\Gamma_1 \vdash^n e : A$ then $\Gamma_2 \vdash^n e[\sigma] : A$,*
- *if $\Gamma_1 \vdash \sigma_1 : \Delta$ then $\Gamma_2 \vdash \sigma_1[\sigma] : \Delta$.*

4.2 Reduction

We first define values and evaluation contexts:

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

Evaluation contexts E are essentially an expression with a hole $[]$, and we write $E[e]$ for the expression obtained by pulling e into the hole of E.

The call-by-value reduction is defined as follows. We write $\longrightarrow_\beta$ for a step of beta reduction, and $\longrightarrow$ for evaluation under an evaluation context.

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

$$\textsc{CtxAppAbs}\quad \frac{}{\Gamma \vdash^n (\lambda_\Delta x : A.\ e_1)\ v_2 \longrightarrow_\beta e_1[\mathrm{id}_\Gamma, x \mapsto v_2]}$$

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

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

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

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

$$\textsc{Cong}\quad \frac{\Gamma \vdash^n e_1 \longrightarrow_\beta e_2}{\Gamma' \vdash^n E[e_1] \longrightarrow E[e_2]}$$

Preservation is a corollary of the substitution lemma (4.2).

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

Progress holds for expressions that don't 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.

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

Notably, since we allow arbitrary nesting of dependencies, having delayed substitutions in our calculus is crucial for progress to hold. For example, consider the following code:

$$\begin{aligned}
&\mathbf{let}_{x:[z:\mathbf{bool} \vdash^1 \mathbf{bool}]} \langle y : \mathbf{bool} \rangle = \\
&\quad \mathbf{let}\ \langle z : \mathbf{bool} \rangle = \langle \mathbf{true} \rangle\ \mathbf{in}\ \langle x_{z \mapsto z} \rangle \\
&\mathbf{in}\ \langle \mathbf{true} \rangle
\end{aligned}$$

Here, y is declared with a dependency x, and x is in turn declared with dependency z. To evaluate the inner let binding, we need a way to substitute z with **true** in the with clause. Without allowing delayed substitutions to contain arbitrary expressions (e.g. $x_{z \mapsto \mathbf{true}}$), the substitution would not be possible, and the evaluation would get stuck. In contrast, the core calculus of Xie et al. [2022] does not allow nested dependencies. As a result, in such a system, variables can simply capture their dependencies from the context without breaking progress.

4.3 Example

We demonstrate the reduction steps of the calculus with a larger example. Since the code fragments are longer, we present them in the concrete syntax for better readability. The mapping between the concrete syntax and the abstract syntax is provided in table 1.

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

Which definition of `z` is supplied to `x`?

First, line 2 is reduced to `<x with z = true>`, as we discussed in the previous subsection.

```
let$ y : (x : (z : bool^1 ⊢ bool^1) ⊢ bool^1) =
  <x with z = true>
in
  let$ x : (z : bool^1 ⊢ bool^1) = <not z> in
  let$ z = <false> in
  <(y with x = x) and z>
```

Then, definition of `y` is substituted with the content of `<x with z = true>`, which triggers the delayed substitution `with x = x`, which has no visible effect.

```
let$ x : (z : bool^1 ⊢ bool^1) = <not z> in
let$ z = <false> in
<(x with z = true) and z>
```

Then, `x` is substituted with the context of `<not z>`, which triggers the delayed substitution `with z = true` and results in `<not true>`.

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

Finally, `z` is substituted with the content of `<false>`.

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

This is the final result, as quoted expressions are values and cannot be reduced further.

4.3.1 Changing Dependencies. Say we want `x` to capture the `z = false` instead, we either have to change the definition of `y` to explicitly capture `z`,

```
let$ y : (x : (z : bool^1 ⊢ bool^1); z : bool^1 ⊢ bool^1) =
  <x with z = z>
in
  let$ x : (z : bool^1 ⊢ bool^1) = <not z> in
  let$ z = <false> in
  <(y with x = x; z = z) and z>
```

or change the definition of `y` to capture a non-capturing version of `x`,

```
let$ y : (x : bool^1 ⊢ bool^1) =
  <x>
in
  let$ x : (z : bool^1 ⊢ bool^1) = <not z> in
  let$ z = <false> in
  <(y with x = (x with z = z)) and z>
```

These examples demonstrate the capability of our type system to express and enforce different kinds of variable dependencies in unhygienic programs.

5 Translating between $\lambda^{\bigcirc\triangleright}$, $\lambda^{\bigcirc}$, and CLTL

We show that $\lambda^{\bigcirc\triangleright}$ is sound and complete with respect to $\lambda^{\bigcirc}$ using its Hilbert-style counterpart, *Constructive Linear-time Temporal Logic* (CLTL).

Davies introduced $\lambda^{\bigcirc}$ [Davies 1996], the first multi-stage language inspired by temporal logic. Kojima and Igarashi developed a Hilbert-style axiomatization of $\lambda^{\bigcirc}$ called *Constructive Linear-time Temporal Logic* (CLTL) [Kojima and Igarashi 2011], which is characterized by the following axioms and rules:

Axioms

- any intuitionistic tautology instance
- **K** : $\bigcirc(A \to B) \to \bigcirc A \to \bigcirc B$
- **CK** : $(\bigcirc A \to \bigcirc B) \to \bigcirc(A \to B)$

Rules

- If $A \to B$ and A, then B.
- If A, then $\bigcirc A$.

To show soundness, we translate $\lambda^{\bigcirc\triangleright}$ types into CLTL formulas and $\lambda^{\bigcirc\triangleright}$ expressions into $\lambda^{\bigcirc}$ expressions. For completeness, we show that CLTL formulas are provable in $\lambda^{\bigcirc\triangleright}$. A direct translation from $\lambda^{\bigcirc}$ to $\lambda^{\bigcirc\triangleright}$, similar to the translation from $\lambda^{\bigcirc}$ to $F^{[\![\,]\!]}$ in [Xie et al. 2022], is also possible but is not covered here.

5.1 $\lambda^{\bigcirc\triangleright}$ to CLTL

We convert types and judgments in $\lambda^{\bigcirc\triangleright}$ to CLTL formulas. Intuitively, the translation involves adding correct number of circles to match the level of staging. 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 way circles are introduced is not important if provability is the main concern. The formal translation for types is defined as follows:

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

$$
\begin{aligned}
[\![\mathbf{bool}]\!] &:= \mathbf{bool} \\
[\![\bigcirc A]\!] &:= \bigcirc[\![A]\!] \\
[\![[\Delta^{n+1} \vdash A] \to B]\!] &:= (\Delta \searrow^{n} [\![A]\!]) \to [\![B]\!] \\
[\![\Delta^{n+1} \triangleright A]\!] &:= \Delta \searrow^{n} [\![A]\!]
\end{aligned}
$$

The notation $\Gamma \searrow^{n} A$ recursively flattens Γ into a nested chain of implications pointing to A, adding $\bigcirc$ constructors 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

$$\Gamma \searrow^{n} A = \bigcirc^{m_1 - n}(\Delta_1 \searrow^{m_1} [\![A_1]\!]) \to \cdots \to \bigcirc^{m_k - n}(\Delta_k \searrow^{m_k} [\![A_k]\!]) \to A.$$

Formally, it is defined as follows:

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

$$
\begin{aligned}
\cdot \searrow^{n} A &:= A \\
(\Gamma, x : [\Delta \vdash^{m} A]) \searrow^{n} B &:= \Gamma \searrow^{n} \left(\bigcirc^{m-n}(\Delta \searrow^{m} [\![A]\!]) \to B\right)
\end{aligned}
$$

Then, a $\lambda^{\bigcirc\triangleright}$ typing judgment $\Gamma \vdash^n e : A$ corresponds to the CLTL formula $\Gamma \searrow^n [\![A]\!]$. 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 $\vdash \Gamma \searrow^n [\![A]\!]$ in CLTL.*

Translation from $\lambda^{\bigcirc\triangleright}$ to $\lambda^{\bigcirc}$. We now define a translation from $\lambda^{\bigcirc\triangleright}$ to $\lambda^{\bigcirc}$, where $\mathbf{let}_\Delta \langle y : A\rangle = e_1 \;\mathbf{in}\; e_2$ is translated into $\mathbf{let}\; y = \langle \lambda\Delta.\ \$(e_1)\rangle \;\mathbf{in}\; e_2[\$(y)/y]$. The translation preserves types but introduces addition beta redexes in quotations, similar to the example shown in Section 2.2. The translation from $\lambda^{\bigcirc\triangleright}$ contexts to $\lambda^{\bigcirc}$ contexts is given below, where each context entry is flattened using the CLTL translation.

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

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

We then define the term translation 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 Δ using lambda abstractions, and $x \bullet \sigma$ applies an variables x to each translated element in σ.

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

$$
\begin{aligned}
[\![x_\sigma]\!] &\coloneqq x \bullet \sigma \\
[\![\mathbf{true}]\!] &\coloneqq \mathbf{true} \\
[\![\mathbf{false}]\!] &\coloneqq \mathbf{false} \\
[\![\mathbf{if}\; e_1 \;\mathbf{then}\; e_2 \;\mathbf{else}\; e_3]\!] &\coloneqq \mathbf{if}\; [\![e_1]\!] \;\mathbf{then}\; [\![e_2]\!] \;\mathbf{else}\; [\![e_3]\!] \\
[\![\lambda_\Delta x : A.\ e]\!] &\coloneqq \lambda x.\ [\![e]\!] \\
[\![e_1\ e_2]\!] &\coloneqq [\![e_1]\!]\ (\boldsymbol{\lambda}\Delta.\ [\![e_2]\!]) \\
[\![\langle e\rangle]\!] &\coloneqq \langle [\![e]\!]\rangle \\
[\![\mathbf{let}_\Delta \langle x : A\rangle = e_1 \;\mathbf{in}\; e_2]\!] &\coloneqq \mathbf{let}\; x = \langle \boldsymbol{\lambda}\Delta.\ [\![e_1]\!]\rangle \;\mathbf{in}\; ([\![e_2]\!][\$(x)/x]) \\
[\![\mathbf{wrap}_\Delta e]\!] &\coloneqq \boldsymbol{\lambda}\Delta.\ [\![e]\!] \\
[\![\mathbf{let\;wrap}_\Delta\; x : A = e_1 \;\mathbf{in}\; e_2]\!] &\coloneqq \mathbf{let}\; x = [\![e_1]\!] \;\mathbf{in}\; [\![e_2]\!]
\end{aligned}
$$

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

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

The translation preserves typing, as stated in the following lemma:

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

5.2 CLTL to $\lambda^{\bigcirc\triangleright}$

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

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

where A and B are types staged at level $n+1$. The ability to introduce dependencies in rule LetQuote is crucial in the the proof of **CK**. This shows that the $\bigcirc$ fragment of our language is complete with respect to CLTL and thus $\lambda^{\bigcirc}$.

Translation from $\lambda^{\bigcirc}$ to $\lambda^{\bigcirc\triangleright}$. A direct translation from $\lambda^{\bigcirc}$ to $\lambda^{\bigcirc\triangleright}$ can be done through a lifting transformation, similar to the translation from $\lambda^{\bigcirc}$ to $F^{[\![]\!]}$ described in [Xie et al. 2022]. The ability to introduce dependencies similarly plays a crucial role in splice lifting.

6 Analytic Macros

We describe $\lambda^{\bigcirc\triangleright}_{\text{pat}}$, an extension of $\lambda^{\bigcirc\triangleright}$ with code pattern matching and code rewriting, enabling analytic macros. The full syntax, typing rules, and operational semantics are summarized in section B.

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.

The $\mathbf{if\ let}_{\Delta}\langle p\rangle = e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3$ expression matches the content of the code expression e_1 against pattern p. It 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. If the match succeeds, e_2 is evaluated with the pattern variables in p bound to the match results. Otherwise, e_3 is evaluated, where the pattern variables are not available.

The $\mathbf{rewrite}\ \langle p_1\rangle\ \mathbf{as}\ e_1\ \mathbf{in}\ e_2$ takes two code expressions e_1 and e_2, replacing occurrences of p_1 with e_1 in e_2. p may contain pattern variables, which matches sub-expressions in e_2 and are made available in e_1.

$$e \ ::= \ \dots \mid \mathbf{if\ let}_{\Delta}\langle p\rangle = e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 \mid \mathbf{rewrite}\ \langle p\rangle\ \mathbf{as}\ e_1\ \mathbf{in}\ e_2$$

The if-let expression differs from the multi-branch expression (`match$`) used in our code examples, as a multi-branch expression can be desugared into nested if-let expressions, and, moreover, if-let expressions are more convenient for formalization and ensure that the language is total.

Code patterns p are expressions with pattern variables that match sub-expressions. To distinguish between pattern variables and regular code variables, we use $\hat{x}$ to denote pattern variables and x to denote regular variables. All expression forms are allowed in patterns, including if-let and rewrite expressions. Substitution patterns π are used to match on substitutions, whose entries are either variables or patterns.

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

We use the notation Π to denote contexts of pattern variables, which is defined as a synonym for the regular contexts Γ and Δ.

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

6.2 Typing Rules

We extend expression typing with rule IfLet that type-checks if-let expressions and rule Rewrite that type-checks rewrite expressions. Rule IfLet generalizes rule LetQuote by replacing the single variable $x : [\Delta \vdash^{n+1} A]$ with a pattern variable context Π^{n+1}, which is made available in the then-branch e_2. Rule Rewrite ensures that both e_1 and e_2 are code expressions. The replacement expression e_1 must have the same type as the pattern p and may use pattern variables from p. The target expression e_2 may have any type, but only sub-expressions that have the same type as the pattern p are considered for rewriting.

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

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

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

The pattern typing judgement $\Gamma; \Delta \vdash^n p : A \rightsquigarrow \Pi$ checks the pattern p under Γ and Δ, producing a type A and a context of pattern variables Π. The typing context is split into Γ and Δ: Γ contains variables from the surrounding context of the if-let expression, allowing patterns to refer to existing variables, while Δ contains local variables introduced either by the $\textbf{let}_\Delta$ or within the pattern p. Separating local variables from the surrounding context ensures that each pattern variable captures the correct dependencies. For example, in $\langle(\lambda x.\ \hat{y})\ \hat{z}\rangle$, the pattern variable $\hat{y}$ should capture x since it matches on a sub-expression that may contain x, while $\hat{z}$ should capture no additional dependencies. In general, pattern variables capture exactly the variables specified in Δ.

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

P-PVar
$$\frac{}{\Gamma; \Delta \vdash^n (\hat{x} : A) : A \rightsquigarrow x : [\Delta \vdash^n A]}$$

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

P-VarSubst2
$$\frac{\Delta \ni x : [\Delta' \vdash^n A] \qquad \Gamma; \Delta \vdash \pi : \Delta' \rightsquigarrow \Pi}{\Gamma; \Delta \vdash^n x_\pi : A \rightsquigarrow \Pi}$$

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

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 handles the typing of pattern variables, producing a single pattern variable that captures the local context Δ. Rules P-VarSubst1 and P-VarSubst2 handle the typing of regular variables in Γ and Δ respectively. When matching on variables in Δ (rule P-VarSubst2), we are allowed to further match on the substitution used with it using a substitution pattern π. For variables in Γ, we can only match on a constant substitution σ (rule P-VarSubst1). This is needed to ensure linearity of pattern variables under substitution, since variables in Γ may be substituted with arbitrary terms. For example, consider the pattern $\langle x_{y \mapsto \hat{z}} \rangle$ where $x \in \Gamma$ and $\hat{z}$ is a pattern variable. When x is substituted with a term where y is not used linearly, such as 0 or $y + y$, linearity of $\hat{z}$ breaks and the pattern no longer type check.

The remaining rules are generalized from the expression typing rules, adapted to handle patterns:

- The typing context Γ is split into Γ and Δ.
- Local variables introduced in the pattern are added to Δ, while Γ remains unchanged.
- Pattern variables produced by each sub-pattern are combined into Π. Since we require contexts to contain distinct variables, linearity is ensured.

We present rule P-CtxAbs and rule P-CtxApp as examples, with the full set of typing rules available in section B.2. In rule P-CtxAbs, the local variable $x : [\Delta' \vdash^n A]$ is added to Δ to check the pattern p. In rule P-CtxApp, the pattern variables produced by p_1 and p_2 are combined into the result.

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

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

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

Typing rules of substitution patterns are generalized from the substitution typing rules. When the entry is a regular variable (rule P-S-Var), we ensure that the variable exists in either Γ or Δ and produce no pattern variables. For entries that are patterns (rule P-S-Pattern), we type-check the pattern and collect the pattern variables it produces.

6.3 Pattern Matching

Matching is defined by the following rules as partial functions. Note that $\mathrm{match}(p; e)$ is defined up to α-equivalence on e: we allow renaming of bound variables in e to match the pattern p. For contexts introduced by λ_Δ, $\mathbf{let}_\Delta$, or $\mathbf{if\ let}_\Delta$, only renaming is allowed but not reordering. These align with the De Bruijn representation used in the formalization. We present a selection of rules, with the complete definition available in section B.3. Notably, we support matching on the full expression syntax, including quotations, if-let and rewrite.

$\boxed{\mathrm{match}(p; e)}$ *(Expression Matching (excerpt))*

$$\begin{aligned}
\mathrm{match}(\hat{x} : A; e) &\coloneqq x \mapsto e \\
\mathrm{match}(x_\sigma; x_\sigma) &\coloneqq \cdot \\
\mathrm{match}(x_\pi; x_\sigma) &\coloneqq \mathrm{match}(\pi; \sigma) \\
\mathrm{match}((\lambda_\Delta x : A.\ p); (\lambda_\Delta x : A.\ e)) &\coloneqq \mathrm{match}(p; e) \\
\mathrm{match}(p_1\ p_2; e_1\ e_2) &\coloneqq \mathrm{match}(p_1; e_1), \mathrm{match}(p_2; e_2)
\end{aligned}$$

$\boxed{\mathrm{match}(\pi; \sigma)}$ *(Substitution Matching)*

$$\begin{aligned}
\mathrm{match}(\cdot; \cdot) &\coloneqq \cdot \\
\mathrm{match}(\pi, x \mapsto y; \sigma, x \mapsto y) &\coloneqq \mathrm{match}(\pi; \sigma) \\
\mathrm{match}(\pi, x \mapsto p; \sigma, x \mapsto e) &\coloneqq \mathrm{match}(\pi; \sigma), \mathrm{match}(p; e)
\end{aligned}$$

The match functions preserves typing in the following way:

- If $\Gamma; \Delta \vdash^n p : A \rightsquigarrow \Pi$ and $\Gamma, \Delta \vdash^n e : A$, and $\mathsf{match}(p; e)$ is defined, then $\Gamma \vdash \mathsf{match}(p; e) : \Pi$.
- If $\Gamma; \Delta \vdash \pi : \Gamma' \rightsquigarrow \Pi$ and $\Gamma, \Delta \vdash \sigma : \Gamma'$, and $\mathsf{match}(\pi; \sigma)$ is defined, then $\Gamma \vdash \mathsf{match}(\pi; \sigma) : \Pi$.

6.4 Rewriting

Rewriting builds on the matching function by applying it to sub-expressions in the target expression, replacing those that match the given pattern with a specified replacement expression. Given a pattern $\Gamma; \cdot \vdash^n p : A \rightsquigarrow \Pi$, replacement expression $\Gamma, \Pi \vdash^n e_1 : A$, and target expression $\Gamma \vdash^n e_2 : B$, The meta-level function $\mathsf{rewrite}(p; e_1; e_2)$ is defined as follows, producing an expression with the same type as e_2:

$\boxed{\mathsf{rewrite}(p; e_1; e_2)}$ *(Rewriting)*

$$\mathsf{rewrite}(p; e_1; e_2) = \begin{cases} e_1[\mathsf{id}_\Gamma, \sigma] & \text{if } A = B \text{ and } \mathsf{match}(p; e_2) = \sigma, \\ \mathsf{rewriteSubterms}(p; e_1; e_2) & \text{otherwise.} \end{cases}$$

where $\mathsf{rewriteSubterms}(p; e_1; e_2)$ applies rewrite to immediate sub-expressions of e_2.

The above definition rewrites all top-most occurrences of p in e_2 with e_1. Other strategies, such as rewriting all occurrences from bottom to top, can also be defined:

$$\begin{aligned} \mathsf{rewrite}_{\mathsf{BottomUp}}(p; e_1; e_2) = {} & \text{let } e_2' = \mathsf{rewriteSubterms}_{\mathsf{BottomUp}}(p; e_1; e_2) \\ & \text{in } \begin{cases} e_1[\mathsf{id}_\Gamma, \sigma] & \text{if } A = B \text{ and } \mathsf{match}(p; e_2') = \sigma, \\ e_2' & \text{otherwise.} \end{cases} \end{aligned}$$

6.5 Substitution and Reduction

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

$$\begin{aligned} \text{Evaluation contexts (excerpt)} \quad E \ ::= \ & \ldots \mid \textbf{if let}_\Delta \langle p \rangle = E \textbf{ then } e_2 \textbf{ else } e_3 \\ & \mid \textbf{rewrite } \langle p_1 \rangle \textbf{ as } E \textbf{ in } e_2 \mid \textbf{rewrite } \langle p_1 \rangle \textbf{ as } v_1 \textbf{ in } E \end{aligned}$$

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

$\boxed{\Gamma \vdash^n e_1 \longrightarrow e_2}$ *(Reduction (excerpt))*

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

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

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

The progress and preservation theorems extends to $\lambda^{\bigcirc\triangleright}_{\mathsf{pat}}$ as well.

7 Denotational Semantics

We define a Kripke-style model [Asai et al. 2014; Mitchell and Moggi 1991] for $\lambda^{\bigcirc\triangleright}$ and $\lambda^{\bigcirc\triangleright}_{\text{pat}}$, where level-$n$ types are interpreted as sets indexed by later-stage contexts Γ^{n+1}, and level-n function types are interpreted as functions indexed by later-stage substitutions $\Gamma' \vdash \sigma : \Gamma$.

We write $\Gamma \vdash^n A$ for the set of typed expressions, and $\Gamma' \vdash \Gamma$ for the set of typed substitutions.

$$(\Gamma \vdash^n A) \coloneqq \{e \mid \Gamma \vdash^n e : A\} \qquad\qquad (\Gamma' \vdash \Gamma) \coloneqq \{\sigma \mid \Gamma' \vdash \sigma : \Gamma\}$$

7.1 Type Interpretation

Types at level n are interpreted as sets indexed by later-stage contexts Γ^{n+1}.

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

$$\begin{aligned}
(\!| [\Delta \vdash A] \to B |\!)_\Gamma &\coloneqq \forall \Gamma'^{n+1}.\ (\Gamma' \vdash \Gamma \to (\!| A |\!)_{\Gamma',\Delta} \to (\!| B |\!)_{\Gamma'}) \\
(\!| \Delta \triangleright A |\!)_\Gamma &\coloneqq (\!| A |\!)_{\Gamma,\Delta} \\
(\!| \textbf{bool} |\!)_\Gamma &\coloneqq \{\mathsf{True}, \mathsf{False}\} \\
(\!| \bigcirc A |\!)_\Gamma &\coloneqq \Gamma \vdash^{n+1} A
\end{aligned}$$

Function types are interpreted as dependent functions, which take a later-stage substitution from Γ to Γ', an element in $(\!| A |\!)_{\Gamma',\Delta}$, and return an element in $(\!| B |\!)_{\Gamma'}$. This definition ensures that we can apply a later-stage substitution $\Gamma' \vdash \sigma : \Gamma$ to the interpretation of a function type. $\Delta \triangleright A$ is interpreted as the interpretation of A under the extended context Γ, Δ. **bool** is interpreted as the set of booleans. $\bigcirc A$ is interpreted as level-$n+1$ expressions of type A under the context Γ.

Given a type A^n, an element $d \in (\!| A |\!)_\Gamma$, and a later-stage substitution $\Gamma' \vdash \sigma : \Gamma$, $d^A[\sigma] \in (\!| A |\!)_{\Gamma'}$ is the result of applying σ to d, which is defined recursively on the type A as follows:

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

$$\begin{aligned}
f^{A \to B}[\sigma] &\coloneqq \lambda \sigma'\ d.\ f\ (\sigma[\sigma'])\ d \\
d^{\Delta \triangleright A}[\sigma] &\coloneqq d^A[\sigma, \mathsf{id}_\Delta] \\
b^{\textbf{bool}}[\sigma] &\coloneqq b \\
e^{\bigcirc A}[\sigma] &\coloneqq e[\sigma]
\end{aligned}$$

For brevity, we write $d[\sigma]$ when the type A is clear from the context.

7.2 Context Interpretation

Typing contexts at level n are interpreted as the product of the interpretations of their entries, where each entry is interpreted differently depending on whether it's at the current stage n. Current-stage entries $\Gamma \ni x : [\Delta \vdash^n A]$ are interpreted as substitution-indexed functions from $(\!| \Delta |\!)$ to $(\!| A |\!)$, while later-stage entries $\Gamma \ni x : [\Delta \vdash^m A]$ with $m > n$ are interpreted as syntactic substitution entries $\Gamma' \vdash x : [\Delta \vdash^m A]$, which can either be a variable $x \mapsto y$ or an expression $x \mapsto e$.

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

$$(\!| \Gamma^n |\!)_{\Gamma'} \coloneqq \prod_{\Gamma \ni x : [\Delta \vdash^m A]} \begin{cases} \forall \Gamma''^{n+1}.\ (\Gamma'' \vdash \Gamma' \to (\!| \Delta |\!)_{\Gamma''} \to (\!| A |\!)_{\Gamma''}) & \text{if } m = n, \\ \Gamma' \vdash x : [\Delta \vdash^m A] & \text{if } m > n. \end{cases}$$

We write ρ to denote an element in $(\!| \Gamma |\!)_{\Gamma'}$ which we call an *environment*. We write $\rho(x)$ to denote the entry corresponding to x in ρ. Entries with level $m > n$ can in an environment ρ can be combined into a later-stage substitution, which we denote as $\rho\restriction_{n+1}$. Applying a later-stage substitution

$\Gamma'' \vdash \sigma : \Gamma'$ to an environment $\rho \in \llparenthesis \Gamma \rrparenthesis_{\Gamma'}$ is defined as follows, where the case for $m = n$ is defined similarly to functions, and the case for $m > n$ is handled by substituting the substitution entry.

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

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

An element $d \in \llparenthesis A \rrparenthesis_{\Gamma,\Delta}$ can be lifted to a singleton environment $\{x^n \mapsto d\} \in \llparenthesis x : [\Delta \vdash^n A] \rrparenthesis_{\Gamma}$, which is defined as:

$\boxed{\{x^n \mapsto d\}}$ *(Singleton Environments)*

$$\{x^n \mapsto d\} \coloneqq \lambda\sigma'.\, d[\sigma', \rho\restriction_{n+1}]$$

We write $\rho \cup \rho'$ to add entries to an environment, where ρ' can either be an environment or a later-stage substitution.

7.3 Expression Interpretation

Given any later-stage context Γ', expressions $\Gamma \vdash^n e : A$ are interpreted as functions $\llparenthesis \Gamma \rrparenthesis_{\Gamma'} \to \llparenthesis A \rrparenthesis_{\Gamma'}$, and substitutions $\Gamma \vdash \sigma : \Delta$ are interpreted as functions $\llparenthesis \Gamma \rrparenthesis_{\Gamma'} \to \llparenthesis \Delta \rrparenthesis_{\Gamma'}$.

$\boxed{\llparenthesis e \rrparenthesis_{\Gamma'}}$ *(Expression Interpretation)*

$$\begin{aligned}
\llparenthesis x_{\sigma_1} \rrparenthesis_{\Gamma'}\ \rho &\coloneqq \rho(x)\ \mathrm{id}_{\Gamma'}\ \llparenthesis \sigma_1 \rrparenthesis_{\Gamma'} \\
\llparenthesis \mathbf{true} \rrparenthesis_{\Gamma'}\ \rho &\coloneqq \mathsf{True} \\
\llparenthesis \mathbf{false} \rrparenthesis_{\Gamma'}\ \rho &\coloneqq \mathsf{False} \\
\llparenthesis \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 \rrparenthesis_{\Gamma'}\ \rho &\coloneqq \begin{cases} \llparenthesis e_2 \rrparenthesis_{\Gamma'}\ \rho & \text{if } \llparenthesis e_1 \rrparenthesis_{\Gamma'}\ \rho = \mathsf{True} \\ \llparenthesis e_3 \rrparenthesis_{\Gamma'}\ \rho & \text{if } \llparenthesis e_1 \rrparenthesis_{\Gamma'}\ \rho = \mathsf{False} \end{cases} \\
\llparenthesis \lambda_\Delta x : A.\ e \rrparenthesis_{\Gamma'}\ \rho &\coloneqq \lambda\sigma'\ d.\ \llparenthesis e \rrparenthesis_{\Gamma'}\ (\rho[\sigma'] \cup \{x^n \mapsto d\}) \\
\llparenthesis e_1\ e_2 \rrparenthesis_{\Gamma'}\ \rho &\coloneqq \llparenthesis e_1 \rrparenthesis_{\Gamma'}\ \rho\ \mathrm{id}_{\Gamma'}\ (\llparenthesis e_2 \rrparenthesis_{\Gamma'}\ (\rho \cup \mathrm{id}_\Delta)) \\
\llparenthesis \langle e \rangle \rrparenthesis_{\Gamma'}\ \rho &\coloneqq e[\rho\restriction_{n+1}] \\
\llparenthesis \mathbf{let}\ \langle \Delta.\ x \rangle = e_1\ \mathbf{in}\ e_2 \rrparenthesis_{\Gamma'}\ \rho &\coloneqq \text{let } e = \llparenthesis e_1 \rrparenthesis_{\Gamma'}\ (\rho \cup \mathrm{id}_\Delta) \text{ in } \llparenthesis e_2 \rrparenthesis_{\Gamma'}\ (\rho \cup (x \mapsto e)) \\
\llparenthesis \mathbf{wrap}_\Delta e \rrparenthesis_{\Gamma'}\ \rho &\coloneqq \llparenthesis e \rrparenthesis_{\Gamma'}\ (\rho \cup \mathrm{id}_\Delta) \\
\llparenthesis \mathbf{let}\ \mathbf{wrap}_\Delta\ x : A = e_1\ \mathbf{in}\ e_2 \rrparenthesis_{\Gamma'}\ \rho &\coloneqq \text{let } d = \llparenthesis e_1 \rrparenthesis_{\Gamma'}\ \rho \text{ in } \llparenthesis e_2 \rrparenthesis_{\Gamma'}\ (\rho \cup \{x^n \mapsto d\})
\end{aligned}$$

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

$$\begin{aligned}
&\llparenthesis \mathbf{if}\ \mathbf{let}_\Delta \langle p \rangle = e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 \rrparenthesis_{\Gamma'}\ \rho \coloneqq \\
&\qquad \text{let } e = \llparenthesis e_1 \rrparenthesis_{\Gamma'}\ (\rho \cup \mathrm{id}_\Delta) \text{ in } \begin{cases} \llparenthesis e_2 \rrparenthesis_{\Gamma'}\ (\rho \cup \sigma) & \text{if } \mathsf{match}(p; e) = \sigma, \\ \llparenthesis e_3 \rrparenthesis_{\Gamma'}\ \rho & \text{otherwise.} \end{cases} \\
&\llparenthesis \mathbf{rewrite}\ \langle p_1 \rangle\ \mathbf{as}\ e_1\ \mathbf{in}\ e_2 \rrparenthesis_{\Gamma'}\ \rho \coloneqq \\
&\qquad \mathsf{rewrite}(p_1[\rho\restriction_{n+1}];\ \llparenthesis e_1 \rrparenthesis_{\Gamma'}\ (\rho \cup \mathrm{id}_\Pi);\ \llparenthesis e_2 \rrparenthesis_{\Gamma'}\ \rho)
\end{aligned}$$

Interpretation of substitutions is defined as follows, where $\Gamma \vdash \sigma : \Delta$ and $\rho \in \llparenthesis \Gamma \rrparenthesis_{\Gamma'}$:

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

$$((\!|\,\sigma\,|\!)_{\Gamma'}\ \rho)\ (x) := \begin{cases} \lambda\sigma'\ \rho'.\ (\!|\,e\,|\!)_{\Gamma'}\ (\rho[\sigma'] \cup \rho') & \text{if } m = n \text{ and } \sigma(x) = e, \\ \rho(y) & \text{if } m = n \text{ and } \sigma(x) = y, \\ \sigma_1(x)[\rho\restriction_{n+1}] & \text{if } m > n, \end{cases}$$
$$\text{for each } \Gamma \ni x : [\Delta' \vdash^m A].$$

For the current-stage entries $\Gamma \ni x : [\Delta' \vdash^n A]$, we want to interpret $\sigma(x)$ under ρ and ρ', which interprets Γ and Δ' respectively. If $\sigma_1(x)$ is an expression $\Gamma, \Delta' \vdash^n e : A$, we interpret e using the concatenation of the two environments. Otherwise, if $\sigma(x)$ is a variable y, then its interpretation already exists in the environment ρ, so we simply look it up. For the later-stage entries $\Gamma \ni x : [\Delta' \vdash^m A]$ with $m > n$, we apply the later-stage part of the environment ρ to the substitution entry, which ensures $((\!|\,\sigma\,|\!)_{\Gamma'}\ \rho)\restriction_{n+1} = (\sigma\restriction_{n+1})[\rho\restriction_{n+1}]$.

7.4 Relation to Operational Semantics

The denotational semantics, compared to the operational semantics described in section 4, is more compositional and guarantees termination by construction. It provides an alternative way to evaluate expressions that is reduction-free and always terminates, by running e under the identity environment id_Γ when Γ is at level $n + 1$. We expect the two semantics to be equivalent, but this has not been formally proven. Proving adequacy of the denotational semantics with respect to the operational semantics involves a logical relation argument, which would also establish termination for the operational semantics.

7.5 Categorification

Categorically, the model is close to a presheaf model [Kavvos 2024] over the category of later-stage substitutions. Refining it into a presheaf model would require proving that all operations commute with substitution, such as $d[\sigma][\sigma'] = d[\sigma[\sigma']]$ for elements. We expect this to be true for the core calculus $\lambda^{\bigcirc\triangleright}$, though it has not been formally proven. For $\lambda^{\bigcirc\triangleright}_{\text{pat}}$, this depends on the definition of match and rewrite. These refinements are left for future work.

8 Discussion

We discuss some of the design choices of our calculi and their implications.

8.1 Explicit Staging of Types

In our calculus, every type A has a fixed stage level, This has the advantage of making staging explicit and allows different stages to have different set of types. However, it makes types such as $A \to \bigcirc A$ impossible to express. One way to address this is to introduce a lifting operator on types and contexts, which converts a type or context from stage n to stage $n + 1$, such as follows:

$$\begin{aligned} (\mathbf{bool})^+ &= \mathbf{bool} \\ ([\Delta \vdash A] \to B)^+ &= [\Delta^+ \vdash A^+] \to B^+ \\ (\bigcirc A)^+ &= \bigcirc(A^+) \\ (\Delta \triangleright A)^+ &= (\Delta^+) \triangleright (A^+) \\ (\cdot)^+ &= \cdot \\ (\Gamma, x : [\Delta \vdash^m A])^+ &= \Gamma^+, x : [\Delta^+ \vdash^{m+1} A^+] \end{aligned}$$

Then, we can express types such as $A \to \bigcirc(A^+)$. Alternatively, we can make staging of every Δ in a type relative, then we recover the ability to use A at different stages, but at the cost of making staging implicit and assuming uniformity of types across stages.

8.2 Multistage Dependencies

Multistage dependencies correspond to nested splices in the quasi-quoting syntax. For example, consider the following expression:

$$\langle \lambda x_1.\ \langle \lambda x_2.\ \$(\$(e_0)) \rangle \rangle$$

where x_1 is a stage-1 variable, x_2 is a stage-2 variable, and e_0 is a stage-0 expression that depends on x_1 and x_2. The expression is equivalent to

$$\mathbf{let}_\Delta \langle y_1 : C \rangle = e_0 \ \mathbf{in}\ \langle \lambda x_1.\ \mathbf{let}_{x_2 : B^2} \langle y_2 : C \rangle = {y_1}_{\mathrm{id}_\Delta} \ \mathbf{in}\ \langle \lambda x_2.\ {y_2}_{x_2 \mapsto x_2} \rangle \rangle$$

in our calculus, where $\Delta = (x_1 : A^1, x_2 : B^2)$ is a multistage dependency context.

8.3 Let-splice vs. Splice

Our calculi use the $\mathbf{let}\ \langle x : A \rangle = e_1\ \mathbf{in}\ e_2$ syntax instead of the traditional in-place splicing syntax $\$(e)$. As discussed in section 1, the let-splice syntax makes the evaluation order explicit and allows finer control. It also naturally extends to the pattern matching syntax $\mathbf{if\ let}\ \langle p \rangle = \langle e_1 \rangle\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3$. However, let-splice syntax can be more verbose in simple cases compared to the traditional splice syntax. We believe the traditional splice syntax could be added to our calculi, at least as syntactic sugar translated into let-splice through a lifting transformation similar to the one in [Xie et al. 2022]. Extending the type system to support both syntaxes is left for future work.

8.4 Unhygienic Function and Value Types

In our core calculus, we included an unhygienic function type $[\Delta \vdash A] \to B$ and an unhygienic value type $\Delta \triangleright A$. The two types are interconvertible via the following functions:

$$\begin{aligned}
&\mathit{wrapToArr} : (\Delta \triangleright A \to B) \to ([\Delta \vdash A] \to B)\\
&\mathit{wrapToArr} := \lambda f.\ \lambda_\Delta x : A.\ f\ (\mathbf{wrap}_\Delta x_{\mathrm{id}_\Delta})\\
&\mathit{arrToWrap} : ([\Delta \vdash A] \to B) \to (\Delta \triangleright A \to B)\\
&\mathit{arrToWrap} := \lambda f.\ \lambda x : (\Delta \triangleright A).\ f\ (\mathbf{let\ wrap}_\Delta\ y : A = x\ \mathbf{in}\ y_{\mathrm{id}_\Delta})
\end{aligned}$$

The unhygienic function type is useful for expressing unhygienic macros, since it does not require explicit wrapping and unwrapping. On the other hand, the unhygienic value type allows unhygienic values to be used as a first-class citizen in the language and be stored in data structures. Without it, we can only annotate unhygienic dependencies on variables and definitions, but not on values. For example, $(\Delta \triangleright A) \times (\Delta' \triangleright B)$ would not be possible without the unhygienic value type.

8.5 Code Pattern Matching and Confluence

We note that $\lambda^{\bigcirc\triangleright}_{\mathrm{pat}}$ is not confluent if we were to allow reducing under let-bindings. For example, consider the following expression:

$$\mathbf{let}\ \langle x : \mathbf{bool} \rangle = \langle \mathbf{true} \rangle\ \mathbf{in}\ (\mathbf{if\ let}\ \langle \mathbf{true} \rangle = \langle x \rangle\ \mathbf{then}\ 1\ \mathbf{else}\ 0)$$

If the outer let-binding reduces first, we get $\mathbf{if\ let}\ \langle \mathbf{true} \rangle = \langle \mathbf{true} \rangle\ \mathbf{then}\ 1\ \mathbf{else}\ 0$ which reduces to 1. If the inner if-let reduces first, the pattern match fails, and we get $\mathbf{let}\ \langle x : \mathbf{bool} \rangle = \langle \mathbf{true} \rangle\ \mathbf{in}\ 0$, which reduces to 0. This is partly due to our mixed treatment of meta-variables and quoted variables, so we cannot distinguish between the two in the pattern match.

8.6 Substitution Patterns

As mentioned in section 6.2, we only allow using substitution patterns with variables that are introduced locally to avoid breaking linearity of patterns variables under substitution. Allowing substitution patterns with non-local variables however, seems to allow patterns to be programmed using substitution. For example, consider the pattern $\langle x_{y \mapsto \hat{z}:\mathbf{int}} \rangle$. Substituting x with $y + 2$ or $y + y$ would produce $\langle \hat{z} : \mathbf{int} + 2 \rangle$ or $\langle \hat{z} : \mathbf{int} + \hat{z} : \mathbf{int} \rangle$ respectively, which seems to be a useful feature as long as we can ensure y is used at least once in the pattern. This would involve integrating linearity into our type system, which could be a possible direction for future work.

9 Formalization

We formalize the syntax, typing rules, operational semantics, safety properties, and translation of our calculi in Agda. Our formalization relies on the `agda-stdlib` library [The Agda Community 2024] and follows the style of *Programming Language Foundations in Agda* [Wadler et al. 2022]. It is structured in the following way:

- `Everything`: Imports all modules and serves as an index.
- `Data.StagedList` and `Data.StagedTree`: Define intrinsically well-staged lists and rose trees, respectively. They are developed in a self-contained and reusable manner, so they can be used in other projects that require well-staged data structures. In our formalization, they are used to represent the nested structure of our typing context.
- `Core.*`, `CtxArr2.*`, `CtxTyp.*`, and `Pat.*`: Formalizes different variants of our calculi. The `Core.*` modules define a minimal calculus with only the $\bigcirc$ modality; the `CtxArr2.*` modules define a calculus with the unhygienic function type but without the unhygienic value type; the `CtxTyp.*` modules formalize $\lambda^{\bigcirc\triangleright}$ in full; the `Pat.*` modules formalize $\lambda^{\bigcirc\triangleright}_{\text{pat}}$. Each of them contains the following submodules:
  - `Context`: Defines types and typing contexts.
  - `Term`: Defines intrinsically typed terms using de Bruijn indices.
  - `Depth`: Defines the depth of contexts and substitutions.
  - `Substitution`: Defines substitution.
  - `Reduction`: Defines the operational semantics and proves safety properties.
  - `Examples`: Contains examples of typable terms in the calculus and their evaluation results.
  - `Denotational` : Defines the Kripke-style denotational semantics.

  Additionally, the `Pat` modules contain the following submodules:
  - `Context.Equality`, `Term.Equality`: Defines decidable equality for contexts and terms.
  - `Matching`: Defines the pattern matching function.
  - `Rewrite`: Defines the rewrite function.
- `Splice.*`: Formalizes the translation from $\lambda^{\bigcirc\triangleright}$ to $\lambda^{\bigcirc}$. It contains the following submodules:
  - `Context`, `Term`: Defines types and terms in $\lambda^{\bigcirc}$.
  - `Translation`: Defines the translation function.

All modules are checked with the `safe` flag to ensure soundness. Most are also checked with `without-K`, except for the `Pat` modules where we use K to simplify the proofs of decidable equality.

There are a few differences between the formalization and the presentation in this paper: in the formalization, all contexts and types are intrinsically well-staged, and all expressions are intrinsically typed. Variables are represented namelessly using de Bruijn indices. These simplifications make the formalization more concise and ensure that pattern matching respects α-equivalence.

10 Related Work

We compare our calculus with related work. Table 2 compares the syntax, type system, and features of our calculus with similar calculi.

	$\lambda^{\bigcirc\triangleright}, \lambda^{\bigcirc\triangleright}_{\text{pat}}$	$\lambda^{\bigcirc}$	$F^{[\![]\!]}$ (Haskell)	Mœbius	$\lambda^{\blacktriangle}$ (Scala 3)	$\lambda^{\{\}}$ (Squid)
Quoting	$\langle\cdot\rangle$	**next**	$[\![\cdot]\!]$	box	$\lceil\cdot\rceil$	$\lceil\cdot\rceil$
Unquoting	**let** $\langle\cdot\rangle$	**prev**	$\$(\cdot)$	let box	$\lfloor\cdot\rfloor$	$\lfloor\cdot\rfloor$
Code Type	$\bigcirc$	$\bigcirc$	*Code*	$\lceil\Phi \vdash^k \cdot\rceil$	$\lceil\cdot\rceil$	$\mathsf{\textbf{Code}}\ T\ C$
Contextual	Yes	–	–	Yes	–	Yes
Nested	Yes	No	1 level	Yes	No	No
Polymorphism	No	No	Yes	Yes	No	Subtyping
Analytic Macros	Yes	–	–	Yes	Yes	Yes
Rewrite	Yes	–	–	–	–	Yes

Table 2. Comparison of our calculus with related work

10.1 Typed Template Haskell

Our calculus is directly inspired by the $F^{[\![]\!]}$ core calculus of Typed Template Haskell [Xie et al. 2022]. Below, we discuss the relationship between $F^{[\![]\!]}$ and our calculus.

In $F^{[\![]\!]}$, let-splice bindings appear in the form of $[\![e]\!]_{\phi}$, where e is a quoted expression and ϕ is a list of let-splice bindings. This is similar to tying the let-splice bindings to the quote construct in our calculus. Since $F^{[\![]\!]}$ is intended as a translation target for a quote-and-splice language and all let-splice bindings are lifted during translation, this design choice is natural. In our calculus, we allow let-splice bindings to appear separately from the quote construct, allowing them to be used more flexibly.

Another difference is that $F^{[\![]\!]}$ context only allows a single level of nesting. Again, this is a natural choice for a translation target for a quote-and-splice language, since the context only needs to track the variable dependencies that are captured by splices. In our calculus, we allow arbitrary nested contexts to support more complex macro signatures and dependency relations. This makes our calculus more expressive but also more complex to reason about. Also, as discussed in section 4.2, it also requires us to introduce delayed substitutions to ensure progress.

We expect the convenience of simply capturing dependencies from the context can be recovered in the surface syntax by automatically generating identity substitutions for the unspecified dependencies. This way, the user can write the code in a more concise way while still having the full power of the calculus.

10.2 S4

In addition to the temporal logic approach, another logic that has been used in the context of meta-programming is the S4 modal logic [Pfenning and Davies 2001], which can be axiomatized as follows:

Axioms

- any intuitionistic tautology instance
- $\mathbf{K} : \Box(A \to B) \to \Box A \to \Box B$
- $\mathbf{T} : \Box A \to A$
- $\mathbf{4} : \Box A \to \Box(\Box A)$

Rules

- If $A \to B$ and A, then B.

- If A, then $\Box A$.

When interpreted as a type system, the box modality $\Box A$ models *closed code expressions* that do not depend on the surrounding context, in contrast to the temporal $\bigcirc A$ which allows code to reference variables in the surrounding context. The **T** axiom corresponds to evaluation of closed expressions, and **4** corresponds to self-quoting.

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's implicit system which uses quote and unquote operators similar to quasi-quotes, $F^{[\![]\!]}$ corresponds to the style of [Bierman and de Paiva 2000] which pairs an explicit substitution with the quote constructor, and $\lambda^{\bigcirc\triangleright}$ corresponds to Pfenning and Davies's implicit system which uses let-bindings for unquoting. In literature, the implicit system is sometimes called Kripke-style or Fitch-style [Clouston 2018; Murase 2017; Murase et al. 2023], while the explicit system is sometimes called the dual-context style [Kavvos 2020; Nanevski et al. 2008].

10.3 Contextual Modal Type Theory and Mœbius

Contextual modal type theory (CMTT) [Nanevski et al. 2008] extends the S4 approach with *contextual modalities*, which generalizes the $\Box$ type to allow code to depend on a specified context, representing open code expressions. *Mœbius* [Boespflug and Pientka 2011; Jang et al. 2022] further extends CMTT into multiple levels, modeling meta^n-variables in multi-stage programming. Our type system is highly inspired by Mœbius. While the two systems are based on different logical foundations and have different approaches to context tracking, some aspects, such as typing rules for delayed substitutions, are strikingly similar. Here, we outline the key differences between our system and Mœbius.

Logical foundation Our system is based on temporal logic, while Mœbius is gereralized from S4.

Separation of modalities We separate the code modality $\bigcirc$ and the contextual modality $\Delta \triangleright$, while Mœbius combines them into a single modality $\lceil \Phi \vdash^k \cdot \rceil$.

Treatment of meta-variables In our system, meta-variables and program variables are both treated as variables at the next level. In Mœbius, meta-variables are treated separately from program variables.

In Mœbius and CMTT, the code type $\lceil \Phi \vdash^k A \rceil$ explicitly declares all variables that the code may refer to. This design makes code evaluation possible because a code of type $\lceil \cdot \vdash^k A \rceil$ is guaranteed to contain no free variables.

In contrast, the temporal code type $\bigcirc A$ allows code to reference any later-stage variables in the surrounding context without explicit declaring them. For instance, a macro $f : \bigcirc\mathbf{int} \to \bigcirc\mathbf{int}$ can be used as $\lambda x : \mathbf{int}.\ \$(f\ \langle x + 1 \rangle)$, where the variable x is introduced at the use site and not known to the macro's definition.

Our calculus build on this by taking an additive approach to context tracking, where a value of type $\Delta \triangleright A$ can use variables in Δ in addition to those in the surrounding context. This allows depdencies that does not follow lexical scoping to be specified, which is essential for expressing unhygienic macros. Moreover, it enhances the expressiveness of the type system by allowing context specifications to be mixed 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 a code of type B that both use a variable x.

10.3.1 Extending $\lambda^{\bigcirc\triangleright}$ with S4-style code types. To extend our calculus with the ability to restrict contexts, we can add a third modal type $\Box A$ which restricts the context to be empty. Semantically, it can be interpreted as

$$\llparenthesis \Box A \rrparenthesis_{\Gamma} \coloneqq \llparenthesis A \rrparenthesis_{\cdot \leftarrow \text{the empty context}}$$

Using this, the S4 code type can be expressed as $\Box\bigcirc A$, which prcisely represents closed code expressions that does not depend on the surrounding context Γ.

$$\llparenthesis \Box\bigcirc A \rrparenthesis_{\Gamma} = \cdot \vdash^{n+1} A$$

Code expressions which do not depend on the surrounding context can be *shifted* [Xie et al. 2023] across levels by adjusting the level annotaions. Therefore, $\Box\bigcirc A^{+} \to A$ can be implemented by shifting the input expression down by one level and then evaluating it, where A^{+} adds 1 to all level annotations in A as defined in section 8.1. Similarly, $\Box\bigcirc A \to \Box\bigcirc(\Box\bigcirc A^{+})$ can be implemented by shifting the input expression up by one level and then quoting it. These properties suggests that $\Box\bigcirc A$ indeed satisfies the S4 axioms. Developing a full λ-calculus with this extension would require a more sophisticated type system to handle the interaction between $\Box$ and $\bigcirc$, which is left for future work. The Mœbius contextual type $\lceil \Phi \vdash^{k} A \rceil$ is similar to $\Box(\Phi \rhd \bigcirc A)$ in this setting. However, there are some differences in how the levels are managed, since they carry different meanings in the two systems. In Mœbius, Φ contains variables with levels smaller than k, while in our system the context contains variables with levels greater than n.

10.4 Polymorphic Contexts

Murase et al. observed that $\lambda^{\bigcirc}$ types can be embedded into a contextual modal type theory extended with *polymorphic contexts* [Murase et al. 2023]. This is similar to viewing the type interpretation function $\llparenthesis A \rrparenthesis_{\Gamma}$ from section 7 as a syntactic translation into CMTT types, where the $\forall\Gamma$ quantification is replaced by $\forall\gamma$, an abstraction over context variables, and the $\bigcirc$ type is translated into CMTT code type under the given context. That is:

$$\begin{aligned} \llparenthesis [\Delta \vdash A] \to B \rrparenthesis_{\Gamma} &\coloneqq \forall\gamma.\ (\gamma \vdash \Gamma) \to \llparenthesis A \rrparenthesis_{\gamma,\Delta} \to \llparenthesis B \rrparenthesis_{\gamma} \quad (\gamma \text{ fresh}) \\ \llparenthesis \bigcirc A \rrparenthesis_{\Gamma} &\coloneqq \lceil \Gamma \vdash A \rceil \end{aligned}$$

where γ is a polymorphic context variable, and Γ may include such variables. This is another promising direction for integrating our calculus with contextual modal type theory.

10.5 Nested Sequents

The nested context design in our calculus is similar to *nested sequents* [Guenot 2013], which has been studied in the context of explicit substitutions and deep inference. Our type system extends this idea by adding stage levels for bind-time tracking, while using a shallow inference system to keep the expression syntax close to the λ-calculus.

10.6 Multimodal Type Theory

Multimodal Type Theory [Gratzer et al. 2020; Kavvos and Gratzer 2023] provides a general framework for combining multiple modal types in a single type system. $\lambda^{\bigcirc\rhd}$ can be seen as a multimodal type system with modalities $\bigcirc$ and $\Delta \rhd$ for each context Δ. Several aspects of our type system, such as having a modal function type and using let-bindings to integrate multiple modalities, also appear in multimodal type theory. The main difference is that multimodal type theory uses Fitch-style syntactical locks 🔒 to control variable usage, while our calculus modifies the context directly using the restriction operator $(\Gamma\restriction_{n+1})$ and extension (Γ, Δ). Specifying our calculus as a multimodal system would be an interesting direction for future work.

10.7 $\lambda^{\blacktriangle}$

Our treatment of analytic macros is similar to that of $\lambda^{\blacktriangle}$ [Stucki et al. 2021]. In $\lambda^{\blacktriangle}$, the typing context is not nested, so when matching a term under a lambda, the result must be first "η-expanded" into a function that takes the code of the dependencies as arguments. For example, in our language, the pattern variable $\hat{x}$ in $\langle \lambda y : A.\ \hat{x} : B \rangle$ has type $x : [y : A^1 \vdash^1 B]$, whereas in $\lambda^{\blacktriangle}$ it would have type $\bigcirc A \to \bigcirc B$. This design simplifies the type system, but as noted by Stucki et al., it only works for a simpler two-stage settings and does not support matching on multi-staged meta-programs.

We extend $\lambda^{\blacktriangle}$'s approach in two ways: First, the nested structure of our type system allows us to directly type the match result as $\Gamma \vdash \sigma : \Pi$, avoiding the need for η-expansion. Second, the translation to $\lambda^{\bigcirc}$ developed in section 5.1 generalizes $\lambda^{\blacktriangle}$'s η-expansion technique to multi-stage programs: For a match result with type $\Gamma \vdash \sigma : \Pi$ in our calculus, one can translate each item in σ using the expression translation function, resulting in a list of items with purely temporal type $[\![\Pi]\!]$ which can be directly typed in $\lambda^{\blacktriangle}$.

10.8 $\lambda^{\{\}}$ and Squid

The rewriting feature in our calculus is inspired by Squid [Parreaux et al. 2017], which is another macro system for Scala with a different type system and feature set. While our type system is quite different from Squid's, the expression syntax for analytic macros is similar. Our **if let** $\langle p \rangle = e_1$ **then** e_2 **else** e_3 is similar to writing e_1 **match** $\lceil p \rceil \Rightarrow e_2$ **else** e_3 in Squid, and **rewrite** $\langle p \rangle$ **as** e_1 **in** e_2 is similar to writing e_2 **rewrite** $\lceil p \rceil \Rightarrow e_1$ in Squid.

11 Conclusion

Correctly tracking binding-time and variable dependencies is essential for the expressiveness of a typed meta-programming language. We introduced a novel approach to this problem using a nested context design combined with temporal-style staging. The approach flexibly supports multiple meta-programming idioms, including explicit splice definition, unhygienic macros, and code pattern matching. We also compared our approach with contextual modal type theory-based systems in section 10, highlighting several potential directions for future work on integrating these frameworks.

References

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.

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.

Steven E Ganz, Amr Sabry, and Walid Taha. 2001. Macros as multi-stage computations: Type-safe, generative, binding macros in MacroML. *ACM SIGPLAN Notices* 36, 10 (2001), 74–85.

Daniel Gratzer, G. A. Kavvos, Andreas Nuyts, and Lars Birkedal. 2020. Multimodal Dependent Type Theory. In *ACM Conferences*. Association for Computing Machinery, New York, NY, USA, 492–506.

Nicolas Guenot. 2013. *Nested Deduction in Logical Foundations for Computation*. Ph. D. Dissertation. Ecole Polytechnique X.

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.

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. doi:10.4230/LIPIcs.FSCD.2024.14

G. A. Kavvos and Daniel Gratzer. 2023. UNDER LOCK AND KEY: A PROOF SYSTEM FOR A MULTIMODAL LOGIC. *Bulletin of Symbolic Logic* 29, 2 (June 2023), 264–293.

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. doi:10.1007/978-3-319-07151-0_6

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. doi: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. doi: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. doi: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.

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*.

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. doi:10.1145/3607851

A $\lambda^{\bigcirc\triangleright}$ Details

A.1 Syntax

$$
\begin{array}{llll}
\text{Variables} & x, y, z & & \\
\text{Levels} & m, n \in \mathbb{N} & & \\
\text{Types} & A, B & ::= & \mathbf{bool} \mid [\Delta \vdash A] \to B \mid \bigcirc A \mid \Delta \triangleright A \\
\text{Contexts} & \Gamma, \Delta & ::= & \cdot \mid \Gamma, x : [\Delta \vdash^{n} A] \\
\text{Expressions} & e & ::= & x_{\sigma} \mid \mathbf{true} \mid \mathbf{false} \mid \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 \\
 & & \mid & \lambda_{\Delta} x : A.\ e \mid e_1\ e_2 \mid \langle e \rangle \mid \mathbf{let}_{\Delta} \langle x : A \rangle = e_1\ \mathbf{in}\ e_2 \\
 & & \mid & \mathbf{wrap}_{\Delta}\, e \mid \mathbf{let\ wrap}_{\Delta}\ x : A = e_1\ \mathbf{in}\ e_2 \\
\text{Substitutions} & \sigma & ::= & \cdot \mid \sigma, x \mapsto y \mid \sigma, x \mapsto e
\end{array}
$$

Fig. 1. Syntax of $\lambda^{\bigcirc\triangleright}$

B $\lambda^{\bigcirc\triangleright}_{\text{pat}}$ Details

B.1 Syntax

$$
\begin{array}{llll}
\text{Variables} & x, y, z & & \\
\text{Levels} & m, n \in \mathbb{N} & & \\
\text{Types} & A, B & ::= & \mathbf{bool} \mid [\Delta \vdash A] \to B \mid \bigcirc A \mid \Delta \triangleright A \\
\text{Contexts} & \Gamma, \Delta, \Pi & ::= & \cdot \mid \Gamma, x : [\Delta \vdash^n A] \\
\text{Expressions} & e & ::= & x_\sigma \mid \mathbf{true} \mid \mathbf{false} \mid \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 \\
 & & \mid & \lambda_\Delta x : A.\ e \mid e_1\ e_2 \mid \langle e \rangle \mid \mathbf{let}_\Delta \langle x : A \rangle = e_1\ \mathbf{in}\ e_2 \\
 & & \mid & \mathbf{wrap}_\Delta\, e \mid \mathbf{let\,wrap}_\Delta\ x : A = e_1\ \mathbf{in}\ e_2 \\
 & & \mid & \mathbf{if\ let}_\Delta \langle p \rangle = e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 \mid \mathbf{rewrite}\ \langle p \rangle\ \mathbf{as}\ e_1\ \mathbf{in}\ e_2 \\
\text{Patterns} & p & ::= & x : A \mid x_\pi \mid \mathbf{true} \mid \mathbf{false} \mid \mathbf{if}\ p_1\ \mathbf{then}\ p_2\ \mathbf{else}\ p_3 \\
 & & \mid & \lambda_\Delta x : A.\ p \mid p_1\ p_2 \mid \langle p \rangle \mid \mathbf{let}_\Delta \langle x : A \rangle = p_1\ \mathbf{in}\ p_2 \\
 & & \mid & \mathbf{wrap}_\Delta\, p \mid \mathbf{let\,wrap}_\Delta\ x : A = p_1\ \mathbf{in}\ p_2 \\
 & & \mid & \mathbf{if\ let}_\Delta \langle p \rangle = p_1\ \mathbf{then}\ p_2\ \mathbf{else}\ p_3 \mid \mathbf{rewrite}\ \langle p \rangle\ \mathbf{as}\ p_1\ \mathbf{in}\ p_2 \\
\text{Substitutions} & \sigma & ::= & \cdot \mid \sigma, x \mapsto y \mid \sigma, x \mapsto e \\
\text{Substitution Patterns} & \pi & ::= & \cdot \mid \pi, x \mapsto y \mid \pi, x \mapsto p
\end{array}
$$

Fig. 2. Syntax of $\lambda^{\bigcirc\triangleright}_{\text{pat}}$

B.2 Typing Rules

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

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

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

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

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

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

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

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

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

S-Empty
$$\frac{}{\Gamma \vdash \cdot : \cdot}$$

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

S-Subst
$$\frac{\Gamma \vdash \sigma : \Gamma' \qquad \Gamma\lceil_m, \Delta \vdash^m e : A}{\Gamma \vdash (\sigma, x \mapsto e) : \Gamma', x : [\Delta \vdash^m A]}$$

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

P-PVar
$$\frac{}{\Gamma;\Delta \vdash^n (\hat{x} : A) : A \rightsquigarrow x : [\Delta \vdash^n A]}$$

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

P-VarSubst2
$$\frac{\Delta \ni x : [\Delta' \vdash^n A] \qquad \Gamma;\Delta \vdash \pi : \Delta' \rightsquigarrow \Pi}{\Gamma;\Delta \vdash^n x_\pi : A \rightsquigarrow \Pi}$$

P-If
$$\frac{\Gamma;\Delta \vdash^n p_1 : \textbf{bool} \rightsquigarrow \Pi_1 \qquad \Gamma;\Delta \vdash^n p_2 : A \rightsquigarrow \Pi_2 \qquad \Gamma;\Delta \vdash^n p_3 : A \rightsquigarrow \Pi_3}{\Gamma;\Delta \vdash^n \textbf{if } p_1 \textbf{ then } p_2 \textbf{ else } p_3 : A \rightsquigarrow \Pi_1,\Pi_2,\Pi_3}$$

P-True
$$\frac{}{\Gamma;\Delta \vdash^n \textbf{true} : \textbf{bool} \rightsquigarrow \cdot}$$

P-False
$$\frac{}{\Gamma;\Delta \vdash^n \textbf{false} : \textbf{bool} \rightsquigarrow \cdot}$$

P-CtxAbs
$$\frac{\Gamma;\Delta, x : [\Delta'^{n+1} \vdash^n A] \vdash^n p : B \rightsquigarrow \Pi}{\Gamma;\Delta \vdash^n (\lambda_{\Delta'} x : A.\ p) : [\Delta' \vdash A] \longrightarrow B \rightsquigarrow \Pi}$$

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

P-Quote
$$\frac{\Gamma\lceil_{n+1};\Delta\lceil_{n+1} \vdash^{n+1} p : A \rightsquigarrow \Pi^{n+1}}{\Gamma;\Delta \vdash^n \langle p \rangle : \bigcirc A \rightsquigarrow \Pi}$$

P-LetQuote
$$\frac{\Gamma;\Delta,\Delta' \vdash^n p_1 : \bigcirc A \rightsquigarrow \Pi_1 \qquad \Gamma;\Delta, x : [\Delta' \vdash^{n+1} A] \vdash^n p_2 : B \rightsquigarrow \Pi_2}{\Gamma;\Delta \vdash^n \textbf{let}_{\Delta'} \langle x : A \rangle = p_1 \textbf{ in } p_2 : B \rightsquigarrow \Pi_1,\Pi_2}$$

$$\textsc{P-Wrap}\;\frac{\Gamma; \Delta, \Delta' \vdash^n p : A \rightsquigarrow \Pi}{\Gamma; \Delta \vdash^n \mathbf{wrap}_{\Delta'}\, p : \Delta' \rhd A \rightsquigarrow \Pi}$$

$$\textsc{P-LetWrap}\;\frac{\Gamma; \Delta \vdash^n p_1 : \Delta' \rhd A \rightsquigarrow \Pi_1 \qquad \Gamma; \Delta, x : [\Delta' \vdash^n A] \vdash^n p_2 : B \rightsquigarrow \Pi_2}{\Gamma; \Delta \vdash^n \mathbf{let\,wrap}_{\Delta'}\; x : A = p_1 \;\mathbf{in}\; p_2 : B \rightsquigarrow \Pi_1, \Pi_2}$$

$$\textsc{P-IfLet}\;\frac{\Gamma; \Delta, \Delta' \vdash^n p_1 : \bigcirc A \rightsquigarrow \Pi_1 \qquad \begin{array}{c}(\Gamma, \Delta)\lceil_{n+1}; \Delta' \vdash^{n+1} p : A \rightsquigarrow \Pi \\ \Gamma; \Delta, \Pi \vdash^n p_2 : B \rightsquigarrow \Pi_2\end{array} \qquad \Gamma; \Delta \vdash^n p_3 : B \rightsquigarrow \Pi_3}{\Gamma; \Delta \vdash^n \mathbf{if\;let}_{\Delta'}\langle p \rangle = p_1 \;\mathbf{then}\; p_2 \;\mathbf{else}\; p_3 : B \rightsquigarrow \Pi_1, \Pi_2, \Pi_3}$$

$$\textsc{P-Rewrite}\;\frac{(\Gamma, \Delta)\lceil_{n+1}; \cdot \vdash^{n+1} p : A \rightsquigarrow \Pi \qquad \Gamma; \Delta, \Pi \vdash^n p_1 : \bigcirc A \rightsquigarrow \Pi_1 \qquad \Gamma; \Delta \vdash^n p_2 : B \rightsquigarrow \Pi_2}{\Gamma; \Delta \vdash^n \mathbf{rewrite}\;\langle p \rangle \;\mathbf{as}\; p_1 \;\mathbf{in}\; p_2 : B \rightsquigarrow \Pi_1, \Pi_2}$$

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

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

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

B.3 Pattern Matching

$\boxed{\mathsf{match}(p; e)}$ *(Expression Matching)*

$$\begin{aligned}
\mathsf{match}(\hat{x} : A; e) &\coloneqq x \mapsto e \\
\mathsf{match}(x_\sigma; x_\sigma) &\coloneqq \cdot \\
\mathsf{match}(x_\pi; x_\sigma) &\coloneqq \mathsf{match}(\pi; \sigma) \\
\mathsf{match}(\mathbf{true}; \mathbf{true}) &\coloneqq \cdot \\
\mathsf{match}(\mathbf{false}; \mathbf{false}) &\coloneqq \cdot \\
\mathsf{match}(\mathbf{if}\; p_1 \;\mathbf{then}\; p_2 \;\mathbf{else}\; p_3; & \\
\mathbf{if}\; e_1 \;\mathbf{then}\; e_2 \;\mathbf{else}\; e_3) &\coloneqq \mathsf{match}(p_1; e_1), \mathsf{match}(p_2; e_2), \mathsf{match}(p_3; e_3) \\
\mathsf{match}((\lambda_\Delta x : A.\; p); (\lambda_\Delta x : A.\; e)) &\coloneqq \mathsf{match}(p; e) \\
\mathsf{match}(p_1\; p_2; e_1\; e_2) &\coloneqq \mathsf{match}(p_1; e_1), \mathsf{match}(p_2; e_2) \\
\mathsf{match}(\langle p \rangle; \langle e \rangle) &\coloneqq \mathsf{match}(p; e) \\
\mathsf{match}(\mathbf{let}_\Delta \langle x : A \rangle = p_1 \;\mathbf{in}\; p_2; & \\
\mathbf{let}_\Delta \langle x : A \rangle = e_1 \;\mathbf{in}\; e_2) &\coloneqq \mathsf{match}(p_1; e_1), \mathsf{match}(p_2; e_2) \\
\mathsf{match}(\mathbf{wrap}_\Delta p; \mathbf{wrap}_\Delta e) &\coloneqq \mathsf{match}(p; e) \\
\mathsf{match}(\mathbf{let\,wrap}_\Delta\; x : A = p_1 \;\mathbf{in}\; p_2; & \\
\mathbf{let\,wrap}_\Delta\; x : A = e_1 \;\mathbf{in}\; e_2) &\coloneqq \mathsf{match}(p_1; e_1), \mathsf{match}(p_2; e_2)
\end{aligned}$$

$$\begin{aligned}
&\mathrm{match}(\textbf{if let}_{\Delta}\langle p\rangle = p_1 \textbf{ then } p_2 \textbf{ else } p_3;\\
&\qquad \textbf{if let}_{\Delta}\langle p\rangle = e_1 \textbf{ then } e_2 \textbf{ else } e_3) \coloneqq \mathrm{match}(p_1; e_1), \mathrm{match}(p_2; e_2), \mathrm{match}(p_3; e_3)\\
&\qquad \mathrm{match}(\textbf{rewrite } \langle p\rangle \textbf{ as } p_1 \textbf{ in } p_2;\\
&\qquad\qquad \textbf{rewrite } \langle p\rangle \textbf{ as } e_1 \textbf{ in } e_2) \coloneqq \mathrm{match}(p_1; e_1), \mathrm{match}(p_2; e_2)
\end{aligned}$$

$\boxed{\mathrm{match}(\pi; \sigma)}$ *(Substitution Matching)*

$$\begin{aligned}
\mathrm{match}(\cdot; \cdot) &\coloneqq \cdot\\
\mathrm{match}(\pi, x \mapsto y; \sigma, x \mapsto y) &\coloneqq \mathrm{match}(\pi; \sigma)\\
\mathrm{match}(\pi, x \mapsto p; \sigma, x \mapsto e) &\coloneqq \mathrm{match}(\pi; \sigma), \mathrm{match}(p; e)
\end{aligned}$$