Refined^2 Environment Classifiers

YUITO MURASE, Kyoto University, Japan
ATSUSHI IGARASHI, Kyoto University, Japan

MetaML-style multi-stage programming (MSP) supports quasi-quotation-based code generation, runtime execution of generated code, and cross-stage persistence (CSP). However, its interaction with computational effects is subtle: mutable state can cause scope extrusion, where generated code escapes the scope of variables on which it depends.

This paper presents a type system for MetaML-style MSP with mutable state that statically rules out harmful scope extrusion while supporting multi-level code generation, runtime execution, and a variant of CSP. Our system builds on refined environment classifiers (RECs), a discipline that annotates code types with the variable scopes on which generated code depends. To scale RECs to the MetaML-style setting, we refine classifiers so that they track not only variable scopes, but also the scopes of classifiers themselves. Further, we integrated polymorphism over classifiers, enabling more general and reusable code generation patterns in a multi-level setting.

For the resulting system, we define an operational semantics via a definitional interpreter and prove type soundness and safety of offline code generation, showing that generated code can be extracted as standalone well-typed programs. We provide working implementations and mechanized proofs in Rocq.

ACM Reference Format:
Yuito Murase and Atsushi Igarashi. 2026. Refined^2 Environment Classifiers. 1, 1 (August 2026), 28 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Multi-stage programming (MSP) allows programmers to write programs that generate and execute code fragments as first-class values. In particular, MetaML-style MSP [20, 38, 39, 42] introduced quasi-quotation syntax for code generation and the `run` primitive for execution, together with cross-stage persistence (CSP), which allows variables to be used at later stages. MetaML-style MSP provides a simple and intuitive programming model and is employed in real programming languages such as MetaOCaml [17, 18] and Scala 3 [35].

However, designing type systems for MetaML-style MSP has long been a central challenge for this approach. In particular, typing disciplines in the presence of runtime execution are already nontrivial [17, 23, 38, 40]. Moreover, the interaction between code generation and computational effects is subtle because it can cause *scope extrusion*, in which generated code may contain variables that are not in scope at the point of execution. At present, even semantic reasoning about effectful code generation remains an active area of research, and there is no agreed-upon typing discipline that can statically detect harmful scope extrusion [4, 19, 20, 22, 31].

Recently, a novel typing discipline called *refined environment classifiers* (RECs) was proposed for effectful code generation while preventing harmful scope extrusion [14, 19, 20, 28]. RECs annotate code types with information about the variable scopes on which generated code depends, ensuring

Authors' Contact Information: Yuito Murase, murase@fos.kuis.kyoto-u.ac.jp, Kyoto University, Kyoto, Japan; Atsushi Igarashi, igrarashi@kyoto-u.ac.jp, Kyoto University, Kyoto, Japan.

ACM XXXX-XXXX/2026/8-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

that generated code fragments are well-scoped. While early proposals targeted combinator-based code generation, Lee et al. [20] designed a type system for MetaML-style MSP. However, their type system is limited to two-level staging and does not support runtime execution nor CSP.

Indeed, the original RECs discipline is inadequate for multi-level staging. As we show in Section 2.3, it does not account for how to track variables that occur under nested quotations. Supporting multi-level code generation therefore requires a further refinement of the typing discipline.

Our contribution. In this paper, we present a novel typing discipline for MetaML-style MSP with mutable state that statically detects harmful scope extrusion. We extend the RECs approach from earlier two-level settings to a MetaML-style setting with multi-level code generation, runtime execution, and scoped CSP.

The main technical refinement is that classifiers no longer track only variable scope dependencies. In a multi-level setting, the type system must also track the scopes of classifiers themselves. This additional scoping information enables the REC approach to scale to nested code generation, and leads to our typing discipline of *refined*^{2} *environment classifiers (R2ECs).*

The system also integrates *polymorphic classifiers*, which substantially increase its flexibility by enabling general and reusable code generators. Prior work on RECs discussed this idea only informally; we provide, to the best of our knowledge, the first formal treatment of polymorphic classifiers for REC-based typing and extend it to the multi-level setting of MetaML-style MSP.

For the full resulting calculus, we prove key metatheoretic properties, including type soundness and the safety of offline code generation. The proofs are mechanized in the Rocq proof assistant.

Structure of the paper. First, Section 2 presents an informal overview of our approach to safe, effectful multi-stage programming through a series of examples. Section 3 gives a formal definition of our surface language and type system, introducing the central ideas of our typing discipline with R2ECs. We present polymorphic classifiers separately in Section 4, providing the idea and formal definitions for polymorphic classifiers. Section 5 defines the operational semantics of our language via a definitional interpreter. Section 6 introduces runtime typing, which we use in Section 7 to prove metatheoretic properties such as type soundness and the safety of offline code generation. Finally, Section 8 discusses related work, and Section 9 concludes with future directions.

2 Informal Description

This section provides an informal overview of our approach to safe, effectful multi-stage programming. We begin with an operational account of untyped MetaML-style staging through a series of examples. These examples illustrate both well-behaved programs and ill-scoped ones that lead to scoping errors, such as scope extrusion, particularly in the presence of mutable state. They highlight the need for a static discipline that can distinguish harmless from harmful uses of free variables in generated code. We then outline how R2ECs provide such a discipline, enabling expressive multi-level staging while preventing the generation of ill-scoped code.

Implementation and Web Demo. We provide a full implementation of the surface type system and definitional interpreter as an interactive web demo for editing, type checking, and evaluating the paper's examples and additional programs. The source code and container images are distributed as a reproducible artifact. OpenAI Codex with GPT-5.5 and Google Antigravity with Gemini 3.1 Pro were used extensively to develop the core implementation and browser interface, respectively. The authors reviewed, corrected, and tested the generated code and take full responsibility for the result.

2.1 MetaML-style MSP by Example

We start with an informal presentation of MetaML-style multi-stage programming in an untyped setting. We first illustrate well-behaved programs that demonstrate the intended behavior of staging constructs. We then present examples that exhibit scoping errors, motivating the need for a typing discipline that enables their static detection.

Quasi-Quotation-Based Code Generation. As a standard example of multi-stage programming, we consider *specializing* the exponentiation function `pow`. The function `pow` takes two arguments `n` and `x` and computes the value x^n:

```
let rec pow n x = if n == 0 then 1 else x * pow (n-1) x
```

It is often beneficial to specialize `pow` with respect to `n` when we know `n` in advance. We therefore define `spow`, a specialized variant of `pow`: given an integer `n`, it generates code for a function that computes x^n without recursive function calls or branching.

```
let spow n = < fun x → $( let rec loop m = if m == 0 then < 1 >
                                           else if m == 1 then < x >
                                           else < x * $(loop (m-1)) > in
                          loop n ) > in
(* spow 10 generates < fun x → x * x * ... * x > *)
let pow10 = run (spow 10) in pow10 2 (* => 1024 *)
```

This definition illustrates quasi-quotation-based code generation. *Quotation* `< e >` constructs a code fragment representing the expression `e`, while *splicing* `$e` inserts a previously generated code fragment into a larger one. In `spow`, the auxiliary function `loop` builds code for the body of the power function by recursion on `n`. Thus, `spow 10` produces code equivalent to a function that multiplies its argument by itself ten times. We can execute the generated code at runtime using the primitive **`run`**. We apply **`run`** to the code generated by `spow 10` to obtain an executable function. We then execute `pow10 2` to compute 2^{10} with optimized code.

Thus, MetaML-style MSP performs specialization by constructing code values via quotation, composing them via splicing, and executing them via **`run`**. We can also emit the content of generated code for later execution: this is called *offline code generation.*

Cross-Stage Persistence. Let us optimize the code generated by `spow` by introducing an auxiliary function `sqr`, which squares its argument. Using `sqr`, the generated code can reuse common subcomputations and avoid repeated multiplications.

```
let sqr x = x * x in
let spow n = < fun x → $( let rec loop m =
                            if m == 0 then < 1 >
                            else if m == 1 then < x >
                            else if m % 2 == 0 then < sqr $( loop (m/2) ) >
                            else < x * $( loop (m-1) ) > in
                          loop n ) > in
(* spow 10 generates < fun x → sqr(x * sqr(sqr(x))) > *)
let pow10 = run (spow 10) in pow10 2 (* => 1024 *)
```

In this program, `sqr` is defined during code generation while being used inside a quotation. Such pattern is common in MetaML-style MSP and is called *cross-stage persistence* (CSP) [13, 38, 39]. The

gap in stages does not pose a problem because the code fragment containing `sqr` is executed by **`run`**: `sqr` is resolved in the runtime environment at the execution point, where `sqr` is defined.

Our language differs from the original MetaML in both the implementation and capabilities of CSP. In particular, CSP is restricted to the scope of persisting variables such as `sqr`, so we call it *scoped CSP*. We return to this point in Sections 3, 5 and 8.

Code Generation with Mutable State. We can also use mutable state during code generation. In the following example, `spow` constructs the body of the function by storing an intermediate code fragment in a reference and updating it iteratively.

```
let spow n = < fun x → $( let r = ref < 1 > in
                          let rec loop m = if m == 0 then ()
                                           else let c = deref r in
                                                r := < x * $c >; loop (m-1) in
                          loop n; deref r ) > in
let pow10 = run (spow 10) in pow10 2
```

Here, the reference `r` stores a code fragment that is repeatedly extended during generation. This program is well-behaved, but in general, storing open code in mutable state can lead to *scope extrusion*, as we will see shortly. While mutable state is not essential in this example, there are staged programming patterns where it is inherently valuable, such as assertion insertion [15, 19].

Scoping Errors. Allowing code fragments with free variables introduces the possibility of scoping errors, where generated code refers to variables that are not available at the point of execution. We present several examples illustrating how such errors arise.

The first example demonstrates *ill-staged runtime execution.*

```
< fun x → $( run < x > ) > (* error: undefined variable x *)
```

In this example, the code fragment `< x >` refers to the variable `x`, which is bound in a future stage inside the quotation. Hence, **`run`** `< x >` fails to resolve `x`, leading to a runtime error.

Another example is *scope extrusion*, which illustrates the problematic interaction between code generation and mutable state.

```
let r = ref < 1 > in
< fun x → $( r := < x >; < x > ) >; deref r (* returns < x > *)
```

Here, a code fragment with `x` is stored in the reference `r` during code generation. The reference itself is allocated outside the scope of `x`, and therefore outlives the binding of `x`. As a result, the code `< x >` escapes its lexical scope and can be retrieved later via **`deref`** `r`, even though no binding for `x` is available.

This example highlights the subtle interaction between code generation and mutable state: code fragments may be stored and later used outside the scope of the variables they depend on. The static detection of such interactions is known to be nontrivial and has long been studied [4, 16, 19, 22, 31].

2.2 Tracking Scopes via Refined Environment Classifiers

To prevent the scoping errors presented in the previous section, we introduce a typing discipline based on *refined environment classifiers (RECs)* [14, 19, 20, 28]. The purpose of RECs is to statically track the variable scopes on which generated code fragments depend. Here, we follow the presentation style of Lee et al. [20] to explain the basic idea of RECs. We illustrate the basic idea of RECs using the `spow` example. We annotate the program with types and *classifiers*.

```
let spow (n:int): <int→int@!> = <! fun x^γ1 → $( let rec loop (m:int): <int@γ1> =
                                                   if m == 0 then <^γ1 1 >
                                                   else if m == 1 then <^γ1 x >
                                                   else <^γ1 x * $(loop (m-1)) > in
                                                 loop n ) > in ...
```

The binding for x is annotated with the classifier γ_1, which represents the scope in which x is valid. The brown color indicates that the classifier is newly introduced. We also use a distinguished classifier ! for the top-level scope, where no local variables are available.

A code type records both the type of a code fragment and the classifier representing the scope on which the fragment may depend. For example, the return type of loop is the code type <int@γ_1>. This indicates that the generated code has type int and may depend on the scope represented by γ_1. Consequently, x may occur free in the generated code, whereas no other variables may occur free.

A quotation switches the current scope to an existing classifier; for instance, <^γ_1 x > switches to γ_1, allowing x to be used inside the quotation. A splice escapes back from the quotation and moves to the scope of the past stage.

Classifiers allow us to detect potential scope extrusion. For example, consider the following annotated version of the scope extrusion example:

```
1 let r^γ1:<int@!> ref = ref <! 1 > in
2 < fun x^δ1:int → $( r := <^δ1 x >; <^δ1 x > ) >;
3 deref r                error: <int@δ1> cannot be assigned to <int@!> ref
```

In this case, the type for mutable state is <int@!> ref, clarifying the scope of code fragments that can be safely assigned to it. When we try to assign <^δ_1 x > at line 2, we get a type error because its type <int@δ_1> is inconsistent with the type of the mutable state <int@!> ref.

We impose a partial order $\preceq$ on classifiers to reflect the nesting structure of scopes: $\delta_1 \preceq \delta_2$ means that the scope δ_2 is located within the scope δ_1. In the spow example, we have ! $\preceq \gamma_1$ because scope γ_1 is nested within the top-level scope !. We use classifiers to track variable scopes of quasi-quotation-based code generation.

2.3 Further Refining RECs for MetaML-style MSP

We apply the RECs discipline to full-fledged MetaML-style MSP, including multi-level staging, runtime execution, and cross-stage persistence. This requires refining the original RECs discipline by elaborating the notion of variable scope. We call the resulting discipline *refined^2 environment classifiers (R2ECs)*. In this section, we explain the motivation for these refinements and illustrate them through examples. We also show that they make it possible to type runtime execution with cross-stage persistence, which is not supported by the original RECs discipline.

Subtleties of Classifiers in Multi-Level Staging. Classifiers track variable scopes in the RECs discipline, but things become more subtle in multi-level settings. Consider the following multi-level program:

```
let r = ref < 1 > in
<< fun x → $$( r := < let y = < x > in 10 >; ① << x >> ) >>;
deref r
```

This program generates the code fragment < let y = < x > in 10 > by assigning it to mutable state at ① and retrieving it later. In this fragment, the variable x appears in a nested quotation and escapes its scope. We expect such programs to be rejected, but this is not easy to detect with

the discipline developed so far. Let us try annotating the program with classifiers. In multi-stage settings, all stages carry classifier annotations.

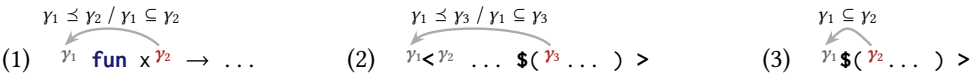
```
1 let r^γ1 :<int@!> ref = ref <! 1 > in
2 <!<! fun x^δ1 :int → $$( r := <! let y^δ2 :<int@δ1> = <^δ1 x >① in 10 >; <!<^δ1 x >> ) >>;
3 deref r
```

At first, this program appears well-typed. The code fragment $<^{\delta_1}$ x > at ① correctly records its dependence on scope δ_1, but this information is not reflected in the typing of the surrounding quotation. As a result, the whole quotation is assigned the type <int@!>, which no longer mentions δ_1. Consequently, when typing the whole `let` binding, the type system cannot detect that x may escape its scope. We argue that the problem lies in the classifier annotation on the quotation at ①. This quotation is nested under the top-level scope !, which should have no knowledge of any other scope. Hence, using δ_1 in the nested quotation appears illegal.

Scoping Structure Formed by Classifiers. The previous example illustrates that the type system must precisely track not only where variables may be used, but also where classifiers may be used. Thus, in addition to ordinary variable scopes, we consider the scopes of classifiers themselves.

Although classifiers represent both kinds of scopes, we distinguish the relations they induce. We call the ordinary variable-scope relation $\preceq$ *reachability*: $\gamma_1 \preceq \gamma_2$ (read "γ_1 is reachable from γ_2") means that variables bound under γ_1 may be used under γ_2. We call the classifier-scope relation $\subseteq$ *visibility*: $\gamma_1 \subseteq \gamma_2$ (read "γ_1 is visible to γ_2") means that the classifier γ_1 may be used under γ_2. Here, using a classifier γ_1 under a classifier γ_2 means introducing, in a context annotated with γ_2, a quotation annotated with γ_1, as in $^{\gamma_2}$ $<^{\gamma_1}$ ...>. Both relations behaves as partial orders.

We illustrate the scoping structure induced by variables and classifiers below.



(1) When a variable x is bound, a fresh classifier γ_2 is introduced to represent its scope. This scope is nested within the surrounding scope γ_1 as an ordinary variable scope, and hence $\gamma_1 \preceq \gamma_2$ holds. The same nesting also makes the surrounding classifier available inside the new scope; therefore, $\gamma_1 \subseteq \gamma_2$ holds as well.

(2) A splice introduces a fresh classifier γ_3 for the splice body. Since the splice escapes from the quotation back to the same stage as γ_1, variables available under γ_1 should also be available under γ_3. Thus, $\gamma_1 \preceq \gamma_3$ holds. For the same reason, the classifier γ_1 is available under γ_3, so $\gamma_1 \subseteq \gamma_3$ also holds.

(3) A splice induces another visibility relation. In the figure, this gives $\gamma_1 \subseteq \gamma_2$, meaning that γ_1 can be used under γ_2. This is necessary, for example, when typing <fun x → $(<x>)>: the splice body constructs the quotation <x>, whose classifier is the scope of x. However, this is only a visibility relation. It does not make variables from γ_1 available inside the splice body, so $\gamma_1 \preceq \gamma_2$ does not hold.

Tracking Variables in Nested Quotations via Classifier Scopes. We now revisit the previous example and show how R2ECs detect its scoping error.

```
1 let r^γ1 :<int@!> ref = ref <! 1 > in
2 <!<! fun x^δ1 :int → $^γ2$(^γ3 r := <!①let y^δ2 :<int@δ1>=<^δ1x > in 10 >; <!<^δ1x >>)>>;
3 deref r                                   error: δ1 is not visible to !
```

The program contains a nested quotation annotated with δ_1. However, this quotation appears inside the top-level quotation marked ①, where δ_1 is not visible to `!`. This use of δ_1 is therefore rejected by our scoping discipline. One might try to repair this visibility error by changing the annotation of the surrounding quotation:

```
1 let r^γ1 :<int@!> ref = ref <! 1 > in
2 <!<! fun x^δ1 :int → $^γ2 $(^γ3 r := <^γ2 ① let y^δ2 :<int@δ1>=<^δ1 ② x > in 10 >; <^γ2 <^δ1 x >>)>>;
3 deref r
                          error: <int@γ2> cannot be assigned to <int@!> ref
```

Here, the classifier of the surrounding quotation is changed from `!` to γ_2 in ①. This use of γ_2 is well-scoped because $\gamma_2 \subseteq \gamma_3$ holds. Then the use of δ_1 in ② is well-scoped because $\delta_1 \subseteq \gamma_2$ holds. However, the assignment is now rejected instead: the mutable cell has type `<int@!> ref`, whereas the assigned code fragment has type `<int@`γ_2`>`. The two types record incompatible scoping information. Thus, the type checker detects the potential scope extrusion even when the escaping variable occurs inside a nested quotation. The key point is that classifiers now track the scopes of nested quotations through the visibility relation $\subseteq$.

Tracking Scopes across Stages. We now show that the scoping discipline illustrated so far naturally extends to runtime execution with CSP. In such settings, classifiers introduced at one stage may be used at later stages, because `run` checks generated code against the current lexical scope. Interestingly, we do not need any additional machinery to support this cross-stage tracking of variable scopes: the notion of visibility already captures the structure needed to track scopes across stages. Let us consider an annotated version of the `spow` example with `sqr`:

```
1 let sqr^γ1 (x^γ2 :int) : int = x * x in
2 let spow^γ3 (n^γ4 :int) :<int→int@γ1> = <^γ1 ① fun x^δ1 :int → ... > in
3 let pow10^γ6 = run (spow 10) ② in pow10 2
```

At ①, the body of the function `spow` introduces a quote to switch the scope from γ_4 to γ_1, which is the scope for `sqr` at the code-generation stage. Such a scope transition is allowed because the visibility relation $\gamma_1 \subseteq \gamma_4$ arises from the reachability relation $\gamma_1 \preceq \gamma_4$. At ②, `run (spow 10)` occurs in scope γ_3, and the classifier ordering $\gamma_1 \preceq \gamma_3$ holds. Therefore, the generated code is well scoped, since all free variables of the code fragment remain available in the lexical environment at the execution point. Thus, the program type-checks, and `run` is safely applied to code with CSP.

We can also detect scoping errors in the ill-staged execution example:

```
1 <! fun x^γ1 :int → $(^γ2 run <^γ1 x > ) >  error: cannot run code with <int@γ1> at γ2
```

Here, `run` attempts to execute a code fragment that is valid only under classifier γ_1. However, `run` occurs at the scope γ_2, where $\gamma_1 \preceq \gamma_2$ does not hold. Consequently, the type system rejects this program due to inconsistency in scoping information.

Summary. R2ECs refine the scoping discipline of RECs by managing both variable scopes and classifier scopes. This refinement allows the type system to detect scope extrusion even when it occurs through nested quotations, thereby establishing static scope safety for MetaML-style multi-stage programming. Furthermore, R2ECs naturally capture the structure needed to track scopes across stages, enabling the type system to support cross-stage persistence without additional machinery.

3 Surface Language and Typing

In this section, we introduce the typing discipline based on R2ECs by defining a surface type system for our multi-stage language. As its name suggests, it types only surface expressions. We later define runtime objects in Section 5, and then introduce a runtime type system that accounts for them in Section 6.

The syntactic categories of the surface type system are defined as follows. We have a set of integers **Int**, ranged over by n, and a set of variables **Var**, ranged over by $x, y, \ldots$. We also have a set of classifiers, ranged over by $\gamma, \delta, \ldots$. We reserve a special classifier $\mathbf{!}$ to represent the top-level scope, which we call the *initial classifier*. We write variables and classifiers in brown when they are binders or newly introduced.

Surface Expression

Expression	e^k	$\in \mathbf{Exp}^k$	$::=$	$n \mid x \mid \lambda x.\, e^k \mid {e_1}^k {e_2}^k \mid \langle e^{k+1} \rangle \mid \$_{k'}\{e^{k-k'}\}$ (where $k \geq k'$)
			$\mid$	$\mathbf{rec}\, f(x).\, e^k \mid \mathbf{ref}\,(e^k) \mid {e_1}^k := {e_2}^k \mid \mathbf{deref}(e^k)$

Type-Level Object

Classifier Stack	$\overrightarrow{\gamma}$	$::=$	$\varepsilon \mid \overrightarrow{\gamma}, \delta$
Base Type	ι	$::=$	$\texttt{int} \mid \ldots$
Type	A, B	$::=$	$\iota \mid A \to B \mid \langle A \rangle^{\gamma} \mid A\ \mathbf{ref}$
Typing Context	Γ	$::=$	$\varepsilon \mid \Gamma, x{:}^{\gamma} A \mid \Gamma, \blacktriangleright^{\gamma} \mid \Gamma, \blacktriangleleft_k^{\gamma}$

For the surface expression, we follow Taha [37] in stratifying expressions into stages, where e^k denotes an expression at stage k. The stage level k is a non-negative integer that indicates the level of staging: $k = 0$ corresponds to the code-generating level, while higher values of k correspond to deeper levels inside quotations. We omit the stage annotation and simply write e when it does not matter for the discussion. It is easy to verify the following property of expressions:

Lemma 1. $\mathbf{Exp}^{k_1} \subseteq \mathbf{Exp}^{k_2}$ *if* $k_1 \leq k_2$.

The syntax of expressions includes integers, variables, functions, applications, quotations, and splices, as well as recursive functions and mutable-state operations. The intended behavior of these constructs is described informally in Section 2.1. Note that we unify the syntax of runtime execution and splicing: $\$_0\{e\}$ represents runtime execution of e, while $\$_1\{e\}$ represents a splice. For $k > 1$, $\$_k\{e\}$ represents a generalized form of splicing that embeds code fragments into deeper stages; we call this a *deep splice*. In the rest of the paper, we use the term *splice* for the construct $\$_k\{e\}$ at any k. We do not include primitive operations in the syntax, but they can be added without affecting the core ideas of our type system.

Note that expressions do not include type-level objects such as classifiers or types. As we will see in Section 5, the dynamic semantics of our language is defined independently of the typing discipline.

We next define the type-level objects. A classifier stack $\overrightarrow{\gamma}$ tracks classifiers at each stage, where the rightmost classifier corresponds to the current stage. In particular, we write $\overrightarrow{\gamma}^+$ to denote a non-empty classifier stack. Types include base types, function types, and code types. A code type $\langle A \rangle^{\gamma}$ represents a code fragment of type A that is valid under classifier γ. The structure of typing contexts is more complex than that of traditional typed lambda calculi. We provide details shortly below. $\mathbf{Dom}_C(\Gamma)$ returns the set of classifiers declared by the context Γ, and $\mathbf{FC}(A)$ returns the set of classifiers that occur freely in the type A.

3.1 Type-level Judgments

We have four type-level judgments, defined in a mutually recursive manner:

$\boxed{\Gamma \vdash \overrightarrow{\gamma}^+ \ \mathbf{wf}}$ $\boxed{\Gamma \vdash \overrightarrow{\gamma}^+ \ A \ \mathbf{wf}}$

$$\text{WF-Ctx-Emp}\ \frac{}{\varepsilon \vdash^! \mathbf{wf}} \qquad \text{WF-Ctx-Var}\ \frac{\Gamma \vdash \overrightarrow{\gamma_1},\gamma_2 \ \mathbf{wf} \qquad \Gamma \vdash \overrightarrow{\gamma_1},\gamma_2 \ A \ \mathbf{wf} \qquad \gamma_3 \notin \mathbf{Dom}_C(\Gamma)}{\Gamma, x{:}^{\gamma_3} A \vdash \overrightarrow{\gamma_1},\gamma_3 \ \mathbf{wf}}$$

$$\text{WF-Ctx-}\blacktriangleright\ \frac{\Gamma \vdash \overrightarrow{\gamma_1},\gamma_2 \ \mathbf{wf} \qquad \Gamma \vdash \gamma_3 \subseteq \gamma_2}{\Gamma, \blacktriangleright^{\gamma_3} \vdash \overrightarrow{\gamma_1},\gamma_2,\gamma_3 \ \mathbf{wf}}$$

$$\text{WF-Ctx-}\blacktriangleleft\ \frac{\Gamma \vdash \overrightarrow{\gamma_1},\delta_0,\ldots,\delta_k \ \mathbf{wf} \qquad \gamma_2 \notin \mathbf{Dom}_C(\Gamma)}{\Gamma, \blacktriangleleft_k^{\gamma_2} \vdash \overrightarrow{\gamma_1},\gamma_2 \ \mathbf{wf}} \qquad \text{WF-T-}\langle\rangle\ \frac{\Gamma, \blacktriangleright^{\gamma_1} \vdash \overrightarrow{\gamma_2}^+,\gamma_1 \ A \ \mathbf{wf}}{\Gamma \vdash \overrightarrow{\gamma_2}^+ \ \langle A \rangle^{\gamma_1} \ \mathbf{wf}}$$

$\boxed{\Gamma \vdash \gamma_1 \preceq \gamma_2}$ $\boxed{\Gamma \vdash \gamma_1 \subseteq \gamma_2}$ Assuming that Γ is well-formed.

$$\preceq\text{-Refl}\ \frac{\gamma \in \mathbf{Dom}_C(\Gamma) \cup \{!\}}{\Gamma \vdash \gamma \preceq \gamma} \qquad \preceq\text{-Trans}\ \frac{\Gamma \vdash \gamma_1 \preceq \gamma_2 \qquad \Gamma \vdash \gamma_2 \preceq \gamma_3}{\Gamma \vdash \gamma_1 \preceq \gamma_3}$$

$$\preceq\text{-Var}\ \frac{\Gamma_1 \vdash \overrightarrow{\gamma_1},\gamma_2 \ \mathbf{wf}}{\Gamma_1, x{:}^{\gamma_3} A, \Gamma_2 \vdash \gamma_2 \preceq \gamma_3} \qquad \preceq\text{-}\blacktriangleleft\ \frac{\Gamma_1 \vdash \overrightarrow{\gamma_1},\delta_0,\delta_1,\ldots,\delta_k \ \mathbf{wf} \qquad k \geq 0}{\Gamma_1, \blacktriangleleft_k^{\gamma_2}, \Gamma_2 \vdash \delta_0 \preceq \gamma_2}$$

$$\subseteq\text{-Lift}\ \frac{\Gamma \vdash \gamma_1 \preceq \gamma_2}{\Gamma \vdash \gamma_1 \subseteq \gamma_2} \qquad \subseteq\text{-Trans}\ \frac{\Gamma \vdash \gamma_1 \subseteq \gamma_2 \qquad \Gamma \vdash \gamma_2 \subseteq \gamma_3}{\Gamma \vdash \gamma_1 \subseteq \gamma_3} \qquad \subseteq\text{-}\blacktriangleleft\ \frac{\Gamma_1 \vdash \overrightarrow{\gamma_1},\gamma_2 \ \mathbf{wf}}{\Gamma_1, \blacktriangleleft_k^{\gamma_3}, \Gamma_2 \vdash \gamma_2 \subseteq \gamma_3}$$

Fig. 1. Curated rules for type-level judgments in surface typing.

Table 2. Context Elements and Their Induced Structure

Context Element	Corresponding Form	Effect on Classifier Stack (before)	→	(after)	Constraints Introduced	Constraints Required
$x{:}^{\gamma} A$	$\lambda x.\ \ldots$	$\overrightarrow{\delta_1}, \delta_2$	→	$\overrightarrow{\delta_1}, \gamma$	$\delta_2 \preceq \gamma$	
$\blacktriangleright^{\gamma}$	$\langle \ldots \rangle$	$\overrightarrow{\delta_1}, \delta_2$	→	$\overrightarrow{\delta_1}, \delta_2, \gamma$		$\gamma \subseteq \delta_2$
$\blacktriangleleft_k^{\gamma}$	$\$_k\{\ldots\}$	$\overrightarrow{\delta}, \delta_0, \ldots, \delta_k$	→	$\overrightarrow{\delta}, \gamma$	$\delta_0 \preceq \gamma, \delta_k \subseteq \gamma$	

Well-formed Contexts $\Gamma \vdash \overrightarrow{\gamma}^+ \ \mathbf{wf}$ **:** the context Γ is well-formed with $\overrightarrow{\gamma}^+$.
Well-formed Types $\Gamma \vdash \overrightarrow{\gamma}^+ \ A \ \mathbf{wf}$ **:** the type A is well-formed under Γ with $\overrightarrow{\gamma}^+$.
Reachability $\Gamma \vdash \gamma_1 \preceq \gamma_2$ **:** γ_1 is reachable from γ_2.
Visibility $\Gamma \vdash \gamma_1 \subseteq \gamma_2$ **:** γ_1 is visible to γ_2.

See Section 2.3 for the informal meaning of $\preceq$ and $\subseteq$. Derivation rules are given in Figure 1. We begin with a description of the well-formedness of typing contexts, along with the structure of contexts themselves. A typing context reflects the structure of the surrounding program: $x{:}^{\gamma} A$ corresponds to a variable binding, $\blacktriangleright^{\gamma}$ corresponds to a quotation, and $\blacktriangleleft_k^{\gamma}$ corresponds to splices (when $k > 1$) or runtime execution (when $k = 0$). A typing context provides three kinds of information:

(1) The mapping of variables, types and classifiers, which are provided by the item $x{:}^{\gamma} A$.
(2) The current classifiers at each stage, provided by a classifier stack.
(3) Birelational structure over classifiers with $\preceq$ and $\subseteq$.

The information in (1) is already provided in standard typed lambda-calculi. The remaining two points are less immediate. We discuss (2) next, and return to (3) when introducing the reachability/visibility judgments. Table 2 summarizes the structure induced by each context element; the corresponding constraints are formalized by the context well-formedness and reachability/visibility judgments.

$$\text{T-Int}\ \frac{\Gamma \vdash^{\overrightarrow{\gamma}} \mathbf{wf}}{\Gamma \vdash^{\overrightarrow{\gamma}} n : \mathtt{int}} \qquad \text{T-Var}\ \frac{\Gamma \vdash^{\overrightarrow{\gamma_1},\gamma_2} \mathbf{wf} \quad \mathbf{lookup}\,(\Gamma, x) = \mathbf{Done}\,(A, \gamma_3) \quad \Gamma \vdash \gamma_3 \preceq \gamma_2}{\Gamma \vdash^{\overrightarrow{\gamma_1},\gamma_2} x : A}$$

$$\text{T-Func}\ \frac{\Gamma, x{:}^{\gamma_1} A_1 \vdash^{\overrightarrow{\gamma_2},\gamma_1} e : A_2 \quad \gamma_1 \notin \mathbf{FC}(A_2) \quad \Gamma \vdash^{\overrightarrow{\gamma_2},\gamma_3} \mathbf{wf}}{\Gamma \vdash^{\overrightarrow{\gamma_2},\gamma_3} \lambda x.\, e : A_1 \to A_2} \qquad \text{T-Ref}\ \frac{\Gamma \vdash^{\overrightarrow{\gamma}^+} e : A}{\Gamma \vdash^{\overrightarrow{\gamma}^+} \mathbf{ref}\,(e) : A\ \mathbf{ref}}$$

$$\text{T-Quo}\ \frac{\Gamma, \blacktriangleright_{\gamma_1} \vdash^{\overrightarrow{\gamma_2}^+,\gamma_1} e : A}{\Gamma \vdash^{\overrightarrow{\gamma_2}^+} \langle e \rangle : \langle A \rangle^{\gamma_1}} \qquad \text{T-Splice}\ \frac{\Gamma, \blacktriangleleft_k^{\gamma_1} \vdash^{\overrightarrow{\gamma_3},\gamma_1} e : \langle A \rangle^{\gamma_2} \quad \Gamma \vdash^{\overrightarrow{\gamma_3},\delta_0,\ldots,\delta_k} \mathbf{wf} \quad \Gamma \vdash \gamma_2 \preceq \delta_k \quad \gamma_1 \neq \gamma_2}{\Gamma \vdash^{\overrightarrow{\gamma_3},\delta_0,\ldots,\delta_k} \$_k\{e\} : A}$$

Fig. 2. Curated rules for the expression typing judgment $\Gamma \vdash^{\overrightarrow{\gamma}^+} e : A$, from surface typing.

A well-formed context judgment $\Gamma \vdash^{\overrightarrow{\gamma}^+} \mathbf{wf}$ states that the context Γ is well-formed and classifiers at each stage are given by the classifier stack $\overrightarrow{\gamma}^+$. We can confirm that the classifier stack in a well-formed context is unique:

Lemma 2. *If* $\Gamma \vdash^{\overrightarrow{\gamma_1}^+} \mathbf{wf}$ *and* $\Gamma \vdash^{\overrightarrow{\gamma_2}^+} \mathbf{wf}$, *then* $\overrightarrow{\gamma_1}^+ = \overrightarrow{\gamma_2}^+$.

In this sense, the information in the classifier stack is already included in the context itself.

We review each rule for well-formed contexts in Figure 1. WF-Ctx-Emp states that the empty context is well-formed. It corresponds to a single stage with the initial classifier. WF-Ctx-Var states that adding an item $x{:}^{\gamma_3} A$ requires that γ_3 is fresh and that A is well-formed. This item also replaces the classifier of the current stage with γ_3, thereby introducing a reachability relation $\gamma_2 \preceq \gamma_3$ (corresponding to$\preceq$-Var). Here, x does not have to be fresh and can shadow another variable in Γ with the same name. WF-Ctx-$\blacktriangleright$ states that entering a quotation with the classifier γ_3 requires that γ_3 is visible from the current classifier γ_2. WF-Ctx-$\blacktriangleleft$ pops k elements from the classifier stack: this represents the effect of splicing that escapes k levels of quotations. It also introduces a new classifier γ_2 replacing δ_0, thereby introducing a reachability relation $\delta_0 \preceq \gamma_2$ (corresponding to $\preceq$-$\blacktriangleleft$) and a visibility relation $\delta_k \subseteq \gamma_2$ (corresponding to $\subseteq$-$\blacktriangleleft$).

The rules for well-formed types $\Gamma \vdash^{\overrightarrow{\gamma}^+} A\ \mathbf{wf}$ are designed so that Γ is well formed. These rules are mostly standard and are therefore omitted. An exception is the rule for code types, WF-T-$\langle\rangle$: it requires A to be well formed under $\Gamma, \blacktriangleright_{\gamma_1}$, meaning that code types introduce the same scoping structure as quotations. Here, the well-formedness of $\Gamma, \blacktriangleright_{\gamma_1}$ entails that γ_1 is visible to the current scope of Γ.

Judgments for the reachability and visibility relations derive these relations from a well-formed context, as described in table 2.

3.2 Expression Typing

We now turn to the typing rules for expressions, given by the judgment $\Gamma \vdash^{\overrightarrow{\gamma}^+} e : A$. This judgment states that the expression e has the type A under the context Γ and the classifier stack $\overrightarrow{\gamma}^+$. We present important rules in Figure 2. T-Var states that a variable x can be used if its scope γ_3 is reachable from the current scope γ_2. Note that we overload the function **lookup** so that it looks up the type and classifier of x in the context Γ; this function tries to find the closest binding for x in Γ, and otherwise returns **Fail**. Due to this behavior of **lookup**, shadowed variables cannot be used. T-Func gives the standard typing rule for functions, except for classifier annotations. The side condition $\gamma_1 \notin \mathbf{FC}(A_2)$ in T-Func states that the classifier γ_1 of the parameter cannot appear in the return type, which ensures that parameter variables in quoted code fragments do not escape

their scope. T-Quo establishes a correspondence between the context element $\blacktriangleright^{\gamma_1}$ and quotation syntax, introducing the code type $\langle A \rangle^{\gamma_1}$. Similarly, T-Splice establishes a correspondence between the context element $\blacktriangleleft_k^{\gamma_1}$ and splice syntax, eliminating the code type $\langle A \rangle^{\gamma_2}$. The side condition $\Gamma \vdash \gamma_2 \preceq \delta_k$ ensures that the scope of the spliced code is consistent with the current scope. The other side condition $\gamma_1 \neq \gamma_2$ ensures that use of γ_1 in typing e does not escape its scope. Note that $\gamma_1 \notin \mathbf{FC}(A)$ is not required because it is already guaranteed by the well-formedness of A. The rules for mutable state are standard.

3.3 Typing Scoped Cross-Stage Persistence

We now discuss the subtlety of typing CSP in our type system. As we discussed in Section 2.1, our type system allows scoped CSP, a form of CSP in which a code fragment containing runtime variables can be safely evaluated at runtime. For example, the expression $\lambda x.\ \$_0\{\langle x \rangle\}$ is well-typed in our type system:

$$\text{T-Func}\ \frac{\text{T-Splice}\ \dfrac{\vdots\ x{:}^{\gamma_1}\texttt{int}, \blacktriangleleft_0^{\gamma_2} \vdash^{\gamma_2} \langle x \rangle : \langle \texttt{int} \rangle^{\gamma_1} \qquad x{:}^{\gamma_1}\texttt{int} \vdash \gamma_1 \preceq \gamma_1 \qquad \gamma_2 \neq \gamma_1}{x{:}^{\gamma_1}\texttt{int} \vdash^{\gamma_1} \$_0\{\langle x \rangle\} : \texttt{int}} \qquad \gamma_1 \notin \mathbf{FC}(\texttt{int})}{\varepsilon \vdash^{!} \lambda x.\ \$_0\{\langle x \rangle\} : \texttt{int} \to \texttt{int}}$$

We next consider a slightly different expression, $\lambda x.\ \langle x \rangle$, which returns a code fragment containing an occurrence of x without executing it. Traditional CSP accepts such an expression by embedding the value of x into a code fragment. However, our type system rejects it because of the side condition of T-Func:

$$\text{T-Func cannot be applied}\ \frac{\vdots\ x{:}^{\gamma_1}\texttt{int} \vdash^{\gamma_1} \langle x \rangle : \langle \texttt{int} \rangle^{\gamma_1} \qquad \gamma_1 \notin \mathbf{FC}(\langle \texttt{int} \rangle^{\gamma_1})\ \text{does NOT hold}}{(\lambda x.\ \langle x \rangle\ \text{fails to type})}$$

In this type derivation, T-Func requires the side condition $\gamma_1 \notin \mathbf{FC}(\langle \texttt{int} \rangle^{\gamma_1})$, which does not hold. The question, then, is how this side condition is justified.

While traditional CSP is achieved by embedding values into code fragments, scoped CSP in our language is achieved by deferring variable resolution until execution. In the expression $\lambda x.\ \$_0\{\langle x \rangle\}$, the variable x is not resolved during code generation; instead, it is resolved when the code fragment is executed. If we do not perform runtime execution, as in $\lambda x.\ \langle x \rangle$, then the code fragment $\langle x \rangle$ is returned and escapes the scope of x. In Section 5, we formalize this intuition in the operational semantics.

Assuming this behavior of scoped CSP, the side condition in T-Func effectively rejects programs that may cause code fragments to escape their scope. In this sense, the typing discipline with R2ECs has strong synergy with the semantics of scoped CSP, which is why we include scoped CSP in our language design. We further discuss this design choice in Section 8.

4 Polymorphic Classifiers

In this section, we extend our type system with polymorphic classifiers, which allow us to type a wider range of staged programs in our language. In fact, this extension is essential for typing realistic staged programs. We first present motivating examples and then give the formal definition of the surface type system.

The idea of polymorphic classifiers already appears in the first paper on RECs [19], but that work did not provide a formal treatment. Polymorphic classifiers were not easy to accommodate in earlier RECs-based type systems and were not discussed in the subsequent work on RECs [14, 20, 28]. Our contributions on polymorphic classifiers are twofold. First, we redesigned the RECs-based type

system so that polymorphic classifiers can be integrated much more easily. Second, we show how to accommodate polymorphic classifiers in a multi-level setting, thereby enabling more flexible typing of multi-stage programs. In this section, we first explain the informal idea of polymorphic classifiers and then give the formal definition of the corresponding surface type system.

4.1 Informal Description

Let us refactor the spow example with sqr by lifting the loop function to the top level for readability.

```
let sqr z = z * z in
let rec loop n yc = if n == 0 then < 1 > else if n == 1 then yc
                    else if n%2 == 0 then < sqr $(loop (n/2) yc) >
                    else < $yc * $(loop (n-1) yc) > in
let spow n = < fun x → $(loop n < x >) > in ...
```

Since loop is defined outside the scope of x, we add a new parameter yc. Then spow passes the code fragment < x > as the argument yc, thereby injecting x into the code generated by loop. However, this program is not typable in our current type system: we expect yc to have the code type <int@δ_1>, where δ_1 is the classifier for x, but δ_1 is not in scope at the point where loop is defined. To type this program, we introduce *polymorphism over classifiers*. (We omit unnecessary classifier annotations for readability in the following examples)

```
let sqr^γ1 (z:int):int = z * z in
let rec loop [δ1 :≥ γ1]^γ2 (n:int) (yc:<int@δ1>②) : <int@δ1> =
  if n == 0 then <^δ1 1 > else if n == 1 then yc
  else if n%2 == 0 then <^δ1③ sqr④ $(loop [δ1] (n/2) yc) > else ... in
let spow (n:int): <int→int@γ1> = <^γ1 fun x^δ2 :int → $(^γ3 loop [δ2]⑤ n <^δ2 x>) > in ...
```

Here, $[\delta_1 :\geq \gamma_1]^{\gamma_2}$ at ① introduces a *polymorphic classifier* δ_1, which is declared to satisfy $\gamma_1 \preceq \delta_1$. It also introduces a new scope γ_2, where δ_1 is available. Hence, we can use the classifier δ_1 in a type such as <int@δ_1> at ②, and introduce a quotation with δ_1 at ③. At ④, we can use sqr because γ_1 is reachable from δ_1. As a result, the type of the loop function is $[\delta_1 :\geq \gamma_1]$(int → <int@$\delta_1$> → <int@$\delta_1$>), and we can instantiate δ_1 later.

At ⑤, loop $[\delta_2]$ instantiates the polymorphic classifier with δ_2. The type checker confirms that δ_2 satisfies two constraints. First, $\gamma_1 \preceq \delta_2$ is required by the declaration of δ_1. Second, δ_2 should be available at the current scope: namely, $\delta_2 \subseteq \gamma_3$ is required, where γ_3 is the classifier of the current scope. Since both constraints are satisfied, this instantiation is valid, and loop $[\delta_2]$ has type int → <int@δ_2> → <int@δ_2>, which can be used to generate code fragments that include x. Thus, polymorphic classifiers allow us to name classifiers that are not available at the point of definition and instantiate them at the point of use. This feature is particularly useful for defining reusable code generators, which are often defined outside the scope of the variables they manipulate.

Since our language is a multi-level calculus, we can also introduce polymorphic classifiers for multi-level code generation. For example, consider a function lift_code that takes a code fragment and returns its code representation. Note that $2 is a deep splice, as explained in Section 3.

```
let lift_code^γ1 [δ1 :≥ !, δ2 :≥ !]^δ0 (x: <int@δ2>): <<int@δ2>@δ1> = <^δ1 <^δ2 $2(x) > >
```

Its type is $[\delta_1 :\geq !,\ \delta_2 :\geq !]$ <int@δ_2> → <<int@δ_2>@δ_1>, where the classifiers δ_1 and δ_2 are introduced simultaneously. This declaration introduces reachability $! \preceq \delta_1$ and $! \preceq \delta_2$, and also introduces a new scope with classifier δ_0, where $\delta_1 \subseteq \delta_0$ and $\delta_2 \subseteq \delta_1$ hold. These visibility constraints allow us

to use a quotation with δ_2 under δ_1. In this sense, the polymorphic classifier declaration `[`δ_1`:≥!, `δ_2`:≥!]` differs from two separate declarations `[`δ_1`:≥!][`δ_2`:≥!]`, since the latter does not introduce the relation $\delta_2 \subseteq \delta_1$.

4.2 Introduction-Form Restriction

Polymorphic classifiers must be used carefully with mutable state: if they are used without restriction, they can cause scope extrusion and lead to unsoundness. The core issue is that a polymorphic classifier may be instantiated differently for the same mutable location, even though the location's contents are not themselves polymorphic. This mismatch between the instantiated classifier and the actual classifier of the stored state leads to unsoundness. For example, the following program exhibits this problem:

```
let r: [δ1:≥!](<int@δ1> ref) = [δ1:≥!]^δ2 ref <^δ1 1 > in
<! fun x^γ1:int → $( r [γ1] := <^γ1 x >; <^γ1 x > ) >
deref (r [!]) (* results in < x > *)
```

To avoid this unsoundness, one approach is to apply the usual value restriction for polymorphic types [41] to polymorphic classifiers, restricting their use to values that do not perform computation. However, this restriction can be relaxed in certain cases.

```
let gen_add_1 [γ1:≥!,γ2:≥!]^γ3 (x:<<int@γ2>@γ1>):<<int@γ2>@γ1> = <^γ1 <^γ2 $$x + 1 > > in
<! let f = [γ4:≥!] fun (y:<int@γ4>^γ5):<int@γ4> → $(gen_add_1 [γ5,γ4] <^γ5y>) ① in ... >
```

Here, the function at ① is typed with polymorphic classifiers, but it is not a value: there is a splice in the function body, yielding computation. Nevertheless, such computation in splices does not cause unsoundness, as we confirm later. In order to allow such cases, we restrict the use of polymorphic classifiers to *introduction forms*: polymorphic classifiers are only introduced to functions, recursive functions and quotations, while computation is permitted in their bodies. We give a formal definition of introduction forms in the next section.

4.3 Surface Typing

Based on the intuition from the examples above, we extend the syntax of the surface type system. We extend type syntax with polymorphic classifier types, and extend typing contexts with polymorphic classifier declarations. We also introduce a new syntactic category *introduction forms*. They are defined as functions, recursive functions, and quotations. These introduction forms may still contain computation via splices: for example, $\langle \$_1\{e_1 e_2\}\rangle$ is a valid introduction form at level 0, in which the splice evaluates the computation $e_1 e_2$.

$$
\begin{array}{ccccl}
\text{Type} & A, B & ::= & \cdots \mid \forall \gamma_1 :\succeq \delta_1, \ldots, \gamma_k :\succeq \delta_k. A \\
\text{Typing Context} & \Gamma & ::= & \cdots \mid \Gamma, [\gamma_1 :\succeq \delta_1, \ldots, \gamma_k :\succeq \delta_k]^{\gamma_0} \\
\text{Introduction Forms} & {e_I}^k & ::= & \lambda x.\ e^k \mid \mathbf{rec}\ f(x).\ e^k \mid \langle e^{k+1} \rangle
\end{array}
$$

Derivation rules for the new type form and context form are presented in Figure 3. Table 3 summarizes the induced structure of polymorphic classifiers, which is formally described in the derivation rules. WF-Ctx-Polycls, $\preceq$-Polycls, and $\subseteq$-Polycls describe relations between classifiers that are declared or used in a polymorphic classifier declaration. Given a declaration $[\gamma_1 :\succeq \gamma'_1, \ldots, \gamma_k :\succeq \gamma'_k]^{\gamma_0}$ at γ'_0, we have the following relations. $\longleftarrow$ and $\leftsquigarrow$ represent $\preceq$ and $\subseteq$, respectively.

WF-Ctx-Polycls

$$\frac{\Gamma \vdash \overrightarrow{\delta}, \gamma_0' \ \mathbf{wf} \qquad \gamma_i \notin \mathbf{Dom}_C(\Gamma) \text{ for } 0 \le i \le k \qquad \Gamma \vdash \gamma_{i+1}' \subseteq \gamma_i' \text{ for } 0 \le i < k}{\Gamma, [\gamma_1 :\succeq \gamma_1', \ldots, \gamma_k :\succeq \gamma_k']^{\gamma_0} \vdash \overrightarrow{\delta_1}, \gamma_0 \ \mathbf{wf}}$$

WF-T-Polycls

$$\frac{\Gamma, [\gamma_1 :\succeq \gamma_1', \ldots, \gamma_k :\succeq \gamma_k']^{\gamma_0} \vdash \overrightarrow{\delta_1}, \gamma_0 \ A \ \mathbf{wf} \qquad \gamma_0 \notin \mathbf{FC}(A) \qquad \Gamma \vdash \overrightarrow{\delta_1}, \gamma_0' \ \mathbf{wf}}{\Gamma \vdash \overrightarrow{\delta_1}, \gamma_0' \ \forall \gamma_1 :\succeq \gamma_1', \ldots, \gamma_k :\succeq \gamma_k'.A \ \mathbf{wf}}$$

$\preceq$-Polycls

$$\frac{\Gamma_1 \vdash \overrightarrow{\delta}, \gamma_0' \ \mathbf{wf} \qquad 0 \le i \le k}{\Gamma_1, [\gamma_1 :\succeq \gamma_1', \ldots, \gamma_k :\succeq \gamma_k']^{\gamma_0}, \Gamma_2 \vdash \gamma_i' \preceq \gamma_i}$$

$\subseteq$-Polycls

$$\frac{0 \le i < k}{\Gamma_1, [\gamma_1 :\succeq \gamma_1', \ldots, \gamma_k :\succeq \gamma_k']^{\gamma_0}, \Gamma_2 \vdash \gamma_{i+1} \subseteq \gamma_i}$$

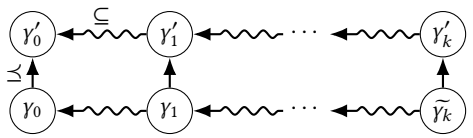
T-Polycls

$$\frac{\Gamma, [\gamma_1 :\succeq \gamma_1', \ldots, \gamma_k :\succeq \gamma_k']^{\gamma_0} \vdash \overrightarrow{\delta_1}, \gamma_0 \ e_I : A \qquad \gamma_0 \notin \mathbf{FC}(A) \qquad \Gamma \vdash \overrightarrow{\delta_1}, \gamma_0' \ \mathbf{wf}}{\Gamma \vdash \overrightarrow{\delta_1}, \gamma_0' \ e_I : \forall \gamma_1 :\succeq \gamma_1', \ldots, \gamma_k :\succeq \gamma_k'.A}$$

T-Appcls

$$\frac{\Gamma \vdash \overrightarrow{\delta'}, \delta_0 \ e : \forall \gamma_1 :\succeq \gamma_1', \ldots, \gamma_k :\succeq \gamma_k'.A \qquad \Gamma \vdash \gamma_i' \preceq \delta_i \text{ for } 1 \le i \le k \qquad \Gamma \vdash \delta_{i+1} \subseteq \delta_i \text{ for } 0 \le i < k}{\Gamma \vdash \overrightarrow{\delta'}, \delta_0 \ e : A[\gamma_1 := \delta_1, \ldots, \gamma_k := \delta_k]}$$

Fig. 3. Surface typing for polymorphic classifiers.



When introducing a polymorphic classifier type, WF-T-Polycls requires the side condition $\gamma_0 \notin \mathbf{FC}(A)$, which safely hides γ_0 from the type. The introduction rule, T-Polycls, applies only to introduction forms: functions, recursive functions, and quotations, which may contain splices of level-0 expressions. Eliminating polymorphic classifier types requires that the instantiated classifiers satisfy the constraints declared by the polymorphic classifier type.

5 Operational Semantics

This section presents the operational semantics of our MetaML-style multi-level calculus, given as a definitional interpreter. The runtime syntax and operational semantics are type-erased. Hence, the typing information presented in Section 3 is used purely for static checking and does not affect the runtime behavior of programs. Our operational semantics is characterized by three key design choices.

Global name generation: Our semantics generates fresh names and reserves them to manage variable names that appear in generated code. This approach is common in multi-stage languages to avoid name conflicts in generated code fragments [6, 19, 22]. It is useful to model the behavior of

Table 3. The Induced Structure of Polymorphic Classifiers

Context Element	Effect on Classifier Stack (before)	→	(after)	Constraints Introduced	Constraints Required
$[\gamma_1 :\succeq \gamma_1', \ldots, \gamma_k :\succeq \gamma_k']^{\gamma_0}$	$\overrightarrow{\delta_1}, \gamma_0'$	→	$\overrightarrow{\delta_1}, \gamma_0$	$\gamma_i' \preceq \gamma_i \ (0 \le i \le k)$ $\gamma_{i+1} \subseteq \gamma_i \ (0 \le i < k)$	$\gamma_{i+1}' \subseteq \gamma_i' \ (0 \le i < k)$

effectful code generation because it allows us to capture situations in which variables temporarily escape their lexical scope while open code fragments are stored in a global store.

Semantics via a definitional interpreter: We formalize the semantics using a definitional interpreter rather than a substitution-based reduction semantics. There are two main reasons for this choice. First, globally reserved names make α-equivalence awkward in a substitution-based reduction semantics. Our definitional interpreter instead deterministically renames binders, yielding a simpler formalization without the need for α-equivalence. Second, the definitional interpreter fits well with Siek's *three-easy-lemmas* methodology [1, 34], which we use to structure the type soundness proof in later sections.

Scoped CSP: Our semantics realizes scoped CSP, in which runtime execution of code fragments respects lexical scoping. This design choice is motivated by the synergy between the semantics of scoped CSP and our typing discipline with R2ECs, as discussed in Section 3.3. Section 5.3 provides evaluation examples to illustrate how our semantics ensures lexical scoping by variable renaming.

5.1 Runtime Objects

We start by defining syntactic categories of runtime objects. We have a set of locations **Loc**, ranged over by l. Locations represent pointers to mutable reference cells.

$$
\begin{array}{cccll}
\text{Value} & v & \in \mathbf{Val} & ::= & n \mid \mathbf{clos}\,(E, R, \lambda x.\ e^0) \mid \mathbf{clos}\,(E, R, \mathbf{rec}\ f(x).\ e^0) \mid \langle e^0 \rangle \mid l \\
\text{Reserved Names} & N & \in \mathbf{RN} & ::= & \varepsilon \mid N, x \\
\text{Value Env.} & E & \in \mathbf{VEnv} & ::= & \varepsilon \mid E, x \coloneqq v \\
\text{Renaming Env.} & R & \in \mathbf{REnv} & ::= & \mathbf{id} \mid R, x \coloneqq y \\
\text{Store} & S & \in \mathbf{Store} & ::= & \varepsilon \mid S, l \coloneqq v
\end{array}
$$

Values include integers, closures for both non-recursive and recursive functions, quoted code fragments, and locations. Reserved names track the names generated during execution, ensuring that fresh names can be generated without conflicts. Value environments map variables to values, and stores model mutable state by mapping locations to values. Renaming environments are specific to our multi-stage setting: they track variable renaming to avoid unintended name capture. **id** is the identity renaming, which maps each variable to itself. We explain how these components work through examples in Section 5.3.

5.2 Definitional Interpreter

We define the operational semantics of our language using a definitional interpreter, following the style of Siek [34]. We use a pseudo-functional language similar to ML to write the interpreter. First, we define an auxiliary type **Result** to represent the possible outcomes of evaluation.

$$\mathbf{type\ Result}(\tau) \ ::= \mathbf{Timeout} \mid \mathbf{Fail} \mid \mathbf{Done}(\tau)$$

Here, **Timeout** represents a timeout due to exceeding the step limit, **Fail** represents a runtime error (e.g., an undefined variable or application of a non-function), and $\mathbf{Done}(\tau)$ represents a successful outcome, yielding a value of type τ. We then introduce monadic notation to describe operations on **Result**:

$$
\begin{aligned}
\mathbf{return}(M) &\equiv \mathbf{Done}(M) \\
X \leftarrow M_1;\ M_2 &\equiv \mathbf{match}\ M_1\ \mathbf{with} \\
&\quad\ \mathbf{case\ Timeout} \Rightarrow \mathbf{Timeout} \\
&\quad\ \mathbf{case\ Fail} \Rightarrow \mathbf{Fail} \\
&\quad\ \mathbf{case\ Done}(R) \Rightarrow [X \coloneqq R]M_2
\end{aligned}
$$

We next introduce auxiliary functions. For brevity, we omit the definitions of these functions here, and provide their type signatures and brief descriptions:

lookup : **VEnv** → **Var** → **Result**(**Val**)	(look up variable in value environment)
gensym : **RN** → **Result**(**Var**)	(generate a fresh variable name)
rename : **REnv** → **Var** → **Var**	(apply renaming)
alloc : **Store** → **Result**(**Loc**)	(allocate a new location)
lookupStore : **Store** → **Loc** → **Result**(**Val**)	(look up value at location)
updateStore : **Store** → **Loc** → **Val** → **Result**(**Store**)	(update value at location)
toQuote : **Val** → **Result**($\mathbf{Exp}^0$)	(decompose value as quoted code)
toLocation : **Val** → **Result**(**Loc**)	(decompose value as a location)

We are now ready to define the main evaluation functions, $\mathcal{V}^{\mathrm{rt}}$ and $\mathcal{V}^{\mathrm{fut}}$, which evaluate expressions at runtime and within quotations, respectively. Their type signatures and curated definitions are given in Figure 4. Both functions take a value environment, a renaming environment, reserved names, a store, and a step limit to ensure termination. The results of these functions include reserved names and a store, modeling the global effects of name generation and mutable state. $\mathcal{V}^{\mathrm{rt}}$ evaluates expressions at stage 0 and returns a value. $\mathcal{V}^{\mathrm{fut}}$ is a dependent function that takes an integer k representing a staging level. It evaluates expressions at the future-stage level $k+1$ and returns an expression at the lower level k, indicating that evaluation for the top-level stage is complete. We highlight key aspects of these definitions:

Runtime evaluator ($\mathcal{V}^{\mathrm{rt}}$)

- A variable (x) is first renamed and then looked up in the value environment.
- Function application (${e_1}^0 {e_2}^0$) evaluates the function and argument, and then evaluates the function body under appropriately extended environments. Instead of adding a mapping directly from x to v_2, we generate a fresh name y, add the mapping from y to v_2 to the value environment, and add the mapping from x to y to the renaming environment. This is necessary to ensure lexical scoping during runtime execution of code fragments with scoped CSP, as illustrated by the examples in Section 5.3. For recursive functions, we also add a binding for the function name to the environments.
- A quotation ($\langle {e_1}^1 \rangle$) is evaluated by evaluating the inner expression ${e_1}^1$ at the future stage via $\mathcal{V}^{\mathrm{fut}}$, and then returning the resulting expression ${e_2}^0$ as a quoted code fragment.
- Runtime execution of generated code ($\$_0\{{e_1}^0\}$) involves evaluating ${e_1}^0$ at runtime to obtain a quoted code fragment, and then evaluating the expression inside it under the same runtime environment. This allows evaluation of open code fragments that may contain free variables: this is how we realize scoped CSP. Free variables in the code fragment have already been renamed; hence, evaluation is performed under the identity renaming environment **id** to avoid further renaming.

Future-stage evaluator ($\mathcal{V}^{\mathrm{fut}}$)

- A variable (x) is renamed and returned as an expression. This allows generated code fragments to be well-scoped under future-stage bindings or runtime environments.
- For generation of a function ($\lambda x.\ e^{k+}$), a fresh name y is generated for the parameter, and x in the body is renamed to y by extending the renaming environment. This ensures that the generated code fragment does not contain duplicate binder names. After evaluating e^{k+}, the reserved name y binds the occurrences of y in the generated code fragment ${e_2}^{k_1}$. We then construct the function $\lambda y.\ {e_2}^{k_1}$, where the parameter y binds those same occurrences in the body. At this point, the binding structure of ${e_2}^{k_1}$ is transferred from the reserved name to the

$$\boxed{\mathcal{V}^{\mathrm{rt}} : \mathbf{Exp}^0 \to \mathbf{VEnv} \to \mathbf{REnv} \to \mathbf{RN} \to \mathbf{Store} \to \mathbf{Int} \to \mathbf{Result}(\mathbf{Val} \times \mathbf{RN} \times \mathbf{Store})}$$

$$\begin{aligned}
\mathcal{V}^{\mathrm{rt}}(e^0, E, R, N, S, 0) &= \mathbf{Timeout} \\
\mathcal{V}^{\mathrm{rt}}(x, E, R, N, S, k^+) &= v \leftarrow \mathbf{lookup}\,(E, \mathbf{rename}\,(R, x));\ \mathbf{return}\,(v, N, S) \\
\mathcal{V}^{\mathrm{rt}}(\lambda x.\ e^0, E, R, N, S, k^+) &= \mathbf{return}\,(\mathbf{clos}\,(E, R, \lambda x.\ e^0), N, S) \\
\mathcal{V}^{\mathrm{rt}}({e_1}^0 {e_2}^0, E, R, N, S, k^+) &= (v_1, N_1, S_1) \leftarrow \mathcal{V}^{\mathrm{rt}}({e_1}^0, E, R, N, S, k);\ (v_2, N_2, S_2) \leftarrow \mathcal{V}^{\mathrm{rt}}({e_2}^0, E, R, N_1, S_1, k); \\
&\quad \mathbf{match}\ v_1\ \mathbf{with} \\
&\quad \mathbf{case}\ \mathbf{clos}\,(E', R', \lambda x.\ {e_3}^0)\ \Rightarrow \\
&\qquad y \leftarrow \mathbf{gensym}\,(N_2);\ \mathcal{V}^{\mathrm{rt}}({e_3}^0, (E', y \coloneqq v_2), (R', x \coloneqq y), (N_2, y), S_2, k) \\
&\quad \mathbf{case}\ \mathbf{clos}\,(E', R', \mathbf{rec}\, f\,(x).\ {e_3}^0)\ \Rightarrow \\
&\qquad g \leftarrow \mathbf{gensym}\,(N_2);\ y \leftarrow \mathbf{gensym}\,(N_2, g); \\
&\qquad \mathcal{V}^{\mathrm{rt}}({e_3}^0, (E', g \coloneqq v_1, y \coloneqq v_2), (R', f \coloneqq g, x \coloneqq y), (N_2, g, y), S_2, k) \\
&\quad \mathbf{case}\ _ \Rightarrow \mathbf{Fail} \\
\mathcal{V}^{\mathrm{rt}}(\langle {e_1}^1 \rangle, E, R, N, S, k^+) &= (e_2^0, N_1, S_1) \leftarrow \mathcal{V}^{\mathrm{fut}}(0, {e_1}^1, E, R, N, S, k);\ \mathbf{return}\,(\langle {e_2}^0 \rangle, N_1, S_1) \\
\mathcal{V}^{\mathrm{rt}}(\$_0\{{e_1}^0\}, E, R, N, S, k^+) &= (v_1, N_1, S_1) \leftarrow \mathcal{V}^{\mathrm{rt}}({e_1}^0, E, R, N, S, k); \\
&\quad {e_2}^0 \leftarrow \mathbf{toQuote}\,(v_1);\ \mathcal{V}^{\mathrm{rt}}({e_2}^0, E, \mathbf{id}, N_1, S_1, k) \\
\mathcal{V}^{\mathrm{rt}}(\mathbf{ref}\,(e^0), E, R, N, S, k^+) &= (v_1, N_1, S_1) \leftarrow \mathcal{V}^{\mathrm{rt}}(e^0, E, R, N, S, k);\ l \leftarrow \mathbf{alloc}\,(S_1);\ \mathbf{return}\,(l, N_1, (S_1, l \coloneqq v_1))
\end{aligned}$$

$$\boxed{\mathcal{V}^{\mathrm{fut}} : (k : \mathbf{Int}) \to \mathbf{Exp}^{k^+} \to \mathbf{VEnv} \to \mathbf{REnv} \to \mathbf{RN} \to \mathbf{Store} \to \mathbf{Int} \to \mathbf{Result}(\mathbf{Exp}^k \times \mathbf{RN} \times \mathbf{Store})}$$

$$\begin{aligned}
\mathcal{V}^{\mathrm{fut}}(k_1, {e^{k_1}}^+, E, R, N, S, 0) &= \mathbf{Timeout} \\
\mathcal{V}^{\mathrm{fut}}(k_1, x, E, R, N, S, {k_2}^+) &= \mathbf{return}\,(\mathbf{rename}\,(R, x), N, S) \\
\mathcal{V}^{\mathrm{fut}}(k_1, \lambda x.\ {e_1}^{k_1+}, E, R, N, S, {k_2}^+) &= y \leftarrow \mathbf{gensym}\,(N); \\
&\quad ({e_2}^{k_1}, N_1, S_1) \leftarrow \mathcal{V}^{\mathrm{fut}}(k_1, {e_1}^{k_1+}, E, (R, x \coloneqq y), (N, y), S, k_2); \\
&\quad \mathbf{return}\,(\lambda y.\ {e_2}^{k_1}, N_1, S_1) \\
\mathcal{V}^{\mathrm{fut}}(k_1, \langle {e_1}^{k_1+2} \rangle, E, R, N, S, {k_2}^+) &= ({e_2}^{k_1+}, N_1, S_1) \leftarrow \mathcal{V}^{\mathrm{fut}}({k_1}^+, {e_1}^{k_1+2}, E, R, N, S, k_2); \\
&\quad \mathbf{return}\,(\langle {e_2}^{k_1+} \rangle, N_1, S_1) \\
\mathcal{V}^{\mathrm{fut}}(k_1, \$_{k_1+}\{{e_1}^0\}, E, R, N, S, {k_2}^+) &= (v_1, N_1, S_1) \leftarrow \mathcal{V}^{\mathrm{rt}}({e_1}^0, E, R, N, S, k_2);\ {e_2}^0 \leftarrow \mathbf{toQuote}\,(v_1); \\
&\quad \mathbf{return}\,({e_2}^0, N_1, S_1) \quad (\text{Note that } {e_2}^0 \in \mathbf{Exp}^{k_1} \text{ by Lemma 1})
\end{aligned}$$

Fig. 4. Curated definitions of evaluator functions. k^+ is a shorthand for $k + 1$.

new binder. This transfer is in fact the most nontrivial part of the type soundness proof and is captured later by Lemma 9.

- When splicing generated code from runtime level ($\$_{k_1+}\{{e_1}^0\}$), the code is evaluated at runtime via $\mathcal{V}^{\mathrm{rt}}$, and the expression in the resulting quoted code fragment is returned. Here, the returned expression ${e_2}^0$ is a level-0 expression, but it can be safely treated as a level-k_1 expression thanks to Lemma 1.

5.3 Evaluation Examples

We show several examples to illustrate how our semantics ensures lexical scoping for generated code fragments by systematically renaming variables.

Example 1: Hygienic Code Generation. Let us consider the following program:

```
1 < fun x① → $( let y = < x >② in < fun x③ → $y④ > ) >
```

This example generates a code fragment `< x >` and splices it under another binding for x (`let` is syntactic sugar for a function and an application). Without renaming, this program would result in `< fun x → fun x → x >`, where the use of `x` is captured by the inner binding, which violates lexical scoping. Our semantics avoids this by systematic renaming with **gensym** and renaming environments. We highlight the evaluation steps for each underlined part of the code:

① Evaluating the outer `fun x` generates a fresh name, say x_1, for x, and adds $x \coloneqq x_1$ to the renaming environment.
② Evaluating `< x >` then renames x to x_1 and returns `< x1 >`.
③ Evaluating the inner `fun x` generates another fresh name, say x_2, and adds $x \coloneqq x_2$ to the renaming environment.
④ Splicing y embeds `< x1 >` directly. Since it is already a value, no further renaming is performed, and `x1` correctly refers to the outer binding.

As a result, the final outcome of evaluating this program is `< fun x1 → fun x2 → x1 >`, where the use of `x1` is correctly captured by the outer binding.

Example 2: Scoped CSP with Lexical Scoping. We consider another staged program that executes a code fragment with scoped CSP.

```
1 let x = 1① in
2 let run_under_x_10 code = let x = 10③ in run code④ in
3 run_under_x_10 < x >②
```

In this example, we generate a code fragment `< x >` that contains a free variable x and execute it under another binding of x. Without renaming, `run_under_x_10 < x >` would execute the code fragment under the inner binding $x = 10$, yielding 10 instead of 1. Our semantics preserves lexical scoping by systematic renaming and thereby prevents dynamic capture of free variables. Consequently, evaluating this program yields 1. We highlight the evaluation steps corresponding to each underlined part of the program:

① Evaluating `let x = 1 in ...` generates a fresh name, say x_1, for x, adds $x_1 \coloneqq 1$ to the value environment, and adds $x \coloneqq x_1$ to the renaming environment.
② Since the renaming environment contains $x \coloneqq x_1$, evaluating `< x >` yields `< x1 >`.
③ Evaluating `run_under_x_10` generates another fresh name, say x_2, for x, adds $x_2 \coloneqq 10$ to the value environment, and adds $x \coloneqq x_2$ to the renaming environment.
④ Finally, `run code` executes the code fragment `< x1 >`. Since this is already a value, no further renaming is needed, and `x1` is resolved to 1, correctly referring to the outer binding.

Thus, fresh name generation together with the renaming environment performs systematic renaming during evaluation, ensuring lexical scoping of generated code and preventing unintended variable capture. In particular, generated code always refers to globally fresh names, ensuring that binders introduced later cannot capture them.

6 Runtime Typing

The surface type system presented in Sections 3 and 4 is not sufficient to prove type soundness with respect to the operational semantics defined in Section 5. This is because the surface type system does not account for runtime objects such as values, reserved names, and stores. To bridge this gap, we introduce a runtime type system that reasons about these runtime objects.

First, we define syntactic categories for typing runtime objects.

$$\begin{array}{rrcl} \textbf{Reserved Names Typing} & \Psi & ::= & \varepsilon \mid \Psi, x : A \mathbin{@} \gamma_1 :\succeq \gamma_2 \mid \Psi, \gamma_1 (:\succeq \gamma_2 / :\supseteq \gamma_3) \\ & & \mid & \Psi, [\gamma_0 :\succeq \delta_0, \ldots, \gamma_k :\succeq \delta_k] \\ \textbf{Location Typing} & \Theta & ::= & \varepsilon \mid \Theta, l : A \end{array}$$

A reserved names typing Ψ assigns types to globally reserved names and maintains reachability and visibility constraints among their classifiers. An element $x : A \mathbin{@} \gamma_1 :\succeq \gamma_2$ states that the reserved name x has type A, is declared at classifier γ_1, and that γ_2 is reachable from γ_1. This element is generated during evaluation of variable binding forms. The element $\gamma_1 (:\succeq \gamma_2 / :\supseteq \gamma_3)$ declares a new classifier γ_1 such that $\gamma_2 \preceq \gamma_1$ and $\gamma_3 \subseteq \gamma_1$. This element is generated during evaluation of splices. The other element $[\gamma_0 :\succeq \delta_0, \ldots, \gamma_k :\succeq \delta_k]$ declares a sequence of classifiers $\gamma_0, \ldots, \gamma_k$ with their bases $\delta_0, \ldots, \delta_k$, introducing reachability constraints $\delta_i \preceq \gamma_i$ and visibility constraints $\gamma_{i+1} \subseteq \gamma_i$. This element is generated during evaluation of polymorphic classifiers. The structure of location typing Θ is standard, mapping locations to their types.

The runtime type system has eleven judgments:

(1)	Well-formed Reserved Names Typing	$\vdash \Psi$ **wf**
(2)	Well-formed Classifier Stack	$\Psi \vdash \overrightarrow{\gamma}^+$ **wf**
(3)	Well-formed Contexts	$\Psi \mid^{\overrightarrow{\gamma_1}^+} \Gamma \vdash^{\overrightarrow{\gamma_2}^+}$ **wf**
(4)	Well-formed Types	$\Psi \mid^{\overrightarrow{\gamma_1}^+} \Gamma \vdash^{\overrightarrow{\gamma_2}^+} A$ **wf** (or, $\Psi \vdash^{\overrightarrow{\gamma_1}^+} A$ **wf** if $\Gamma = \varepsilon$)
(5)	Reachability	$\Psi \mid^{\overrightarrow{\gamma_1}^+} \Gamma \vdash \gamma_2 \preceq \gamma_3$ (or, $\Psi \vdash \gamma_2 \preceq \gamma_3$ if $\Gamma = \varepsilon$)
(6)	Visibility	$\Psi \mid^{\overrightarrow{\gamma_1}^+} \Gamma \vdash \gamma_2 \subseteq \gamma_3$ (or, $\Psi \vdash \gamma_2 \subseteq \gamma_3$ if $\Gamma = \varepsilon$)
(7)	Expression Typing	$\Psi \mid^{\overrightarrow{\gamma_1}^+} \Gamma \vdash_R^{\overrightarrow{\gamma_2}^+} e : A$
(8)	Well-formed Location Typing	$\Psi \vdash \Theta$ **wf**
(9)	Store Typing	$\Psi \vdash S : \Theta$
(10)	Value Typing	$\Psi \mid \Theta \vdash^{\gamma} v : A$ **val**
(11)	Well-formed Value Environments	$\Psi \mid \Theta \vdash^{\gamma} E$ **wf**

$\vdash \Psi$ **wf** defines well-formed reserved names typing, and $\Psi \vdash \overrightarrow{\gamma}^+$ **wf** defines a well-formed classifier stack under Ψ. Their derivation rules are straightforward and omitted here. The other judgments largely divide into two groups: judgments extended from surface typing with reserved names typing (3–7), and new judgments for runtime objects (8–11).

For the first group, we present important rules in Figure 5 for brevity, and the other rules are mostly the same as those in surface typing. These judgments extend those from surface typing with a reserved names typing Ψ and a classifier stack $\overrightarrow{\gamma_1}^+$. Here, $\overrightarrow{\gamma_1}^+$ represents the classifier stack at the point where evaluation takes place; hence, the classifier stack for the empty context will be $\overrightarrow{\gamma_1}^+$ instead of $!$, as stated in R-WF-Ctx-Nil. We also introduce five additional rules that deduce reachability and visibility from reserved names typing: R-$\preceq$-NHT-Name, R-$\preceq$-NHT-$\blacktriangleleft$, R-$\subseteq$-NHT-$\blacktriangleleft$, R-$\preceq$-NHT-Polycls and R-$\subseteq$-NHT-Polycls. The judgment for expression typing $\Psi \mid^{\overrightarrow{\gamma_1}^+} \Gamma \vdash_R^{\overrightarrow{\gamma_2}^+} e : A$ is also extended with a renaming environment R. We use it to type reserved names in R-T-Name. This rule states that a reserved name y can be used as a variable x if (1) the scope of y is reachable from the current classifier, (2) x renames to y under the renaming environment, and (3) x is not shadowed in the context Γ.

For judgments on runtime objects, we present the relevant rules in Figure 6. These judgments type values, stores, and value environments. A value typing judgment $\Psi \mid \Theta \vdash^{\gamma} v : A$ **val** states that a value v has type A at classifier γ under reserved names typing Ψ and location typing Θ. Unlike standard typed lambda calculi, where values are typed only at the top-level scope, this judgment types values at a local scope γ, whose visibility information is needed to type open code fragments. VT-Clos-Func states that a closure value **clos** $(E, R, \lambda x.\ e^0)$ is well-typed if the

$\boxed{\Psi \mid^{\overrightarrow{\gamma_1}^+} \Gamma \vdash^{\overrightarrow{\gamma_2}^+} \mathbf{wf}}$ $\boxed{\Psi \mid^{\overrightarrow{\gamma_1}^+} \Gamma \vdash \gamma_2 \preceq \gamma_3}$ $\boxed{\Psi \mid^{\overrightarrow{\gamma_1}^+} \Gamma \vdash \gamma_2 \subseteq \gamma_3}$

$$\text{R-WF-Ctx-Nil}\ \frac{\Psi \vdash \overrightarrow{\gamma}^+\ \mathbf{wf}}{\Psi \mid^{\overrightarrow{\gamma}^+} \varepsilon \vdash^{\overrightarrow{\gamma}^+} \mathbf{wf}} \qquad \text{R-}\preceq\text{-NHT-Name}\ \frac{x : A\ @\ \gamma_1 :\succeq \gamma_2 \in \Psi}{\Psi \mid^{\overrightarrow{\gamma_3}^+} \Gamma \vdash \gamma_2 \preceq \gamma_1} \qquad \text{R-}\preceq\text{-NHT-}\blacktriangleleft\ \frac{\gamma_1(:\succeq \gamma_2/:\supseteq \gamma_3) \in \Psi}{\Psi \mid^{\overrightarrow{\gamma_4}^+} \Gamma \vdash \gamma_2 \preceq \gamma_1} \qquad \text{R-}\subseteq\text{-NHT-}\blacktriangleleft\ \frac{\gamma_1(:\succeq \gamma_2/:\supseteq \gamma_3) \in \Psi}{\Psi \mid^{\overrightarrow{\gamma_4}^+} \Gamma \vdash \gamma_3 \subseteq \gamma_1}$$

$$\text{R-}\preceq\text{-NHT-Polycls}\ \frac{[\gamma_0 :\succeq \gamma_0', \ldots, \gamma_k :\succeq \gamma_k'] \in \Psi \qquad 0 \le i \le k}{\Psi \mid^{\overrightarrow{\delta}^+} \Gamma \vdash \gamma_i' \preceq \gamma_i} \qquad \text{R-}\subseteq\text{-NHT-Polycls}\ \frac{[\gamma_0 :\succeq \gamma_0', \ldots, \gamma_k :\succeq \gamma_k'] \in \Psi \qquad 0 \le i < k}{\Psi \mid^{\overrightarrow{\delta}^+} \Gamma \vdash \gamma_{i+1} \subseteq \gamma_i}$$

$\boxed{\Psi \mid^{\overrightarrow{\gamma_1}^+} \Gamma \vdash_R^{\overrightarrow{\gamma_2}^+} e : A}$

$$\text{R-T-Name}\ \frac{\mathbf{lookup}\ (\Gamma, x) = \mathbf{Fail} \qquad y : A\ @\ \gamma_1 :\succeq \gamma_2 \in \Psi \qquad \Psi \mid^{\overrightarrow{\gamma_3}^+} \Gamma \vdash^{\overrightarrow{\gamma_4}, \gamma_5} \mathbf{wf} \qquad \Psi \mid^{\overrightarrow{\gamma_3}^+} \Gamma \vdash \gamma_1 \preceq \gamma_5 \qquad \mathbf{rename}\ (R, x) = y}{\Psi \mid^{\overrightarrow{\gamma_3}^+} \Gamma \vdash_R^{\overrightarrow{\gamma_4}, \gamma_5} x : A}$$

Fig. 5. Curated rules for judgments that are extended from surface typing.

$\boxed{\Psi \mid \Theta \vdash^{\gamma} v : A\ \mathbf{val}}$

$$\text{VT-Clos-Func}\ \frac{\Psi \mid^{\gamma_1} \varepsilon \vdash_R^{\gamma_1} \lambda x.\ e^0 : A \qquad \Psi \mid \Theta \vdash^{\gamma_1} E\ \mathbf{wf} \qquad \Psi \vdash^{\delta} A\ \mathbf{wf}}{\Psi \mid \Theta \vdash^{\delta} \mathbf{clos}\ (E, R, \lambda x.\ e^0) : A\ \mathbf{val}} \qquad \text{VT-Code}\ \frac{\Psi \mid^{\gamma_1} \varepsilon \vdash_{\mathbf{id}}^{\gamma_1} \langle e^0 \rangle : A \qquad \Psi \vdash \Theta\ \mathbf{wf}}{\Psi \mid \Theta \vdash^{\gamma_1} \langle e^0 \rangle : A\ \mathbf{val}}$$

$$\text{VT-Loc}\ \frac{\Psi \vdash \Theta\ \mathbf{wf} \qquad \mathbf{lookupStore}(\Theta, l) = \mathbf{Done}\ (A) \qquad \Psi \vdash^{\gamma} A\ \mathbf{wf}}{\Psi \mid \Theta \vdash^{\gamma} l : A\ \mathbf{ref}\ \mathbf{val}}$$

$\boxed{\Psi \mid \Theta \vdash^{\gamma} E\ \mathbf{wf}}$ $\boxed{\Psi \vdash S : \Theta}$

$$\text{WF-VEnv-Empty}\ \frac{\Psi \vdash \Theta\ \mathbf{wf}}{\Psi \mid \Theta \vdash^{!} \varepsilon\ \mathbf{wf}} \qquad \text{WF-VEnv-Var}\ \frac{\Psi \mid \Theta \vdash^{\gamma_1} E\ \mathbf{wf} \qquad \Psi \mid \Theta \vdash^{\gamma_1} v : A\ \mathbf{val} \qquad x : A\ @\ \gamma_2 :\succeq \gamma_1 \in \Psi}{\Psi \mid \Theta \vdash^{\gamma_2} E, x := v\ \mathbf{wf}}$$

$$\text{WF-VEnv-Shut}\ \frac{\Psi \mid \Theta \vdash^{\gamma_1} E\ \mathbf{wf} \qquad \gamma_2(:\succeq \gamma_1/:\supseteq \gamma_3) \in \Psi}{\Psi \mid \Theta \vdash^{\gamma_2} E\ \mathbf{wf}} \qquad \text{WF-VEnv-Polycls}\ \frac{\Psi \mid \Theta \vdash^{\delta_0} E\ \mathbf{wf} \qquad [\gamma_0 :\succeq \delta_0, \ldots, \gamma_k :\succeq \delta_k] \in \Psi}{\Psi \mid \Theta \vdash^{\gamma_0} E\ \mathbf{wf}}$$

$$\text{T-Store}\ \frac{S = (l_1 := v_1, \ldots, l_k := v_k) \qquad \Theta = (l_1 : A_1, \ldots, l_k : A_k) \qquad \Psi \vdash \Theta\ \mathbf{wf} \qquad \Psi \mid \Theta \vdash^{\gamma_i} v_i : A_i\ \mathbf{val}\ \text{with some}\ \gamma_i\ \text{for}\ 1 \le i \le k}{\Psi \vdash S : \Theta}$$

Fig. 6. Curated rules for judgments with regard to runtime objects.

underlying function is well-typed at some classifier under the given renaming environment, the captured value environment is well-formed at that classifier, and the closure type is well-formed at the classifier where the closure itself is typed. We have a similar rule for typing closures with recursive functions. VT-Code states that a code value $\langle e^0 \rangle$ is well-typed if the quotation itself is well-typed under the identity renaming environment and the location typing Θ is well-formed. VT-Loc states that a location value l has type A **ref** if $l : A$ appears in Θ and A is well-formed at the current classifier γ. Here, we overload **lookupStore** to look up the type assigned to l in Θ. One key

aspect of value typing is that a value can be typed at any classifier as long as its type is well-formed at that classifier, as stated in the following lemma:

Lemma 3 (Value Portability). *If* $\Psi \mid \Theta \vdash^{\gamma_1} v : A\ \mathbf{val}$ *and* $\Psi \vdash^{\gamma_2} A\ \mathbf{wf}$*, then* $\Psi \mid \Theta \vdash^{\gamma_2} v : A\ \mathbf{val}$.

We leverage this fact for typing stores later.

The value-environment well-formedness judgment $\Psi \mid \Theta \vdash^{\gamma} E\ \mathbf{wf}$ states that a value environment E is well-formed at the classifier γ. The empty environment is well-formed at the initial classifier, as specified by WF-VEnv-Empty. WF-VEnv-Var extends the environment with a new binding $x \coloneqq v$ while advancing the classifier along the reachability witnessed by an entry $x : A\ @\ \gamma_2 {:}\succeq \gamma_1 \in \Psi$. WF-VEnv-Shut and WF-VEnv-Polycls similarly advance the classifier along the reachability witnessed by the entries, leaving the value environment unchanged.

For stores, we have two judgments: well-formedness of location typing $\Psi \vdash \Theta\ \mathbf{wf}$ and store typing $\Psi \vdash S : \Theta$. In Θ, the type assigned to each location is assumed to be well-formed at some classifier, but that classifier is not recorded in Θ. We can observe a similar discipline in T-Store. This is justified by Lemma 3: since any classifier may be chosen as long as the relevant types are well-formed under it, we do not need to track a specific classifier for each location.

7 Metatheoretic Properties

We now have all the machinery needed to state and prove metatheoretic properties of our language. The goal of this section is to prove two properties: soundness of our surface and runtime type systems, and safety of offline code generation.

7.1 Type Soundness

To state type soundness with respect to the definitional interpreter in Section 5, we employ Siek's three-easy-lemmas style of type soundness proof [1, 34]. First, we prove the safety of auxiliary functions used in the definitional interpreter.

Lemma 4 (Safety of **lookup**). *If* $\Psi \mid^{\gamma} \varepsilon \vdash^{\gamma}_{R} x : A$ *and* $\Psi \mid \Theta \vdash^{\gamma} E\ \mathbf{wf}$*, then there exists* v *such that* $\mathbf{lookup}\,(E, \mathbf{rename}\,(R, x)) = \mathbf{Done}\,(v)$ *and* $\Psi \mid \Theta \vdash^{\gamma} v : A\ \mathbf{val}$.

Lemma 5 (Safety of **lookupStore**). *If* $\Psi \mid \Theta \vdash^{\gamma} l : A\ \mathbf{ref}\ \mathbf{val}$ *and* $\Psi \vdash S : \Theta$*, then there exists* v *such that* $\mathbf{lookupStore}(S, l) = \mathbf{Done}\,(v)$ *and* $\Psi \mid \Theta \vdash^{\gamma} v : A\ \mathbf{val}$.

Lemma 6 (Safety of **updateStore**). *If* $\Psi \mid \Theta \vdash^{\gamma} v : A\ \mathbf{val}$, $\Psi \vdash S : \Theta$ *and* $l : A \in \Theta$*, then* $\mathbf{updateStore}\,(S, l, v) = \mathbf{Done}\,(S_1)$ *for some* S_1 *where* $\Psi \vdash S_1 : \Theta$.

We then prove *expression persistency*, which ensures that an expression well-typed at the top-level stage remains well-typed at any future stage as long as classifiers are consistent. This property is used when a generated code fragment is spliced into larger code fragments.

Lemma 7 (Expression Persistency). *If* $\Psi \mid^{\gamma_1} \varepsilon \vdash^{\gamma_1}_{\mathbf{id}} e^0 : A$, $\Psi \vdash \overrightarrow{\gamma_2}, \gamma_3\ \mathbf{wf}$ *and* $\Psi \vdash \gamma_1 \preceq \gamma_3$*, then* $\Psi \mid^{\overrightarrow{\gamma_2}, \gamma_3} \varepsilon \vdash^{\overrightarrow{\gamma_2}, \gamma_3}_{\mathbf{id}} e^0 : A$.

The following promotion and demotion results move variables and classifiers between reserved names typing and contexts. The demotion result is a difficult and important part of the metatheory, because it requires careful handling of binding relations.

Lemma 8 (Promotion).

(1) *If* $\Psi \mid^{\overrightarrow{\gamma_1}^+ \blacktriangleright \gamma_2}\vdash_R^{\overrightarrow{\gamma_1}^+,\gamma_2} e : A$, *then* $\Psi \mid^{\overrightarrow{\gamma_1}^+,\gamma_2} \varepsilon \vdash_R^{\overrightarrow{\gamma_1}^+,\gamma_2} e : A$.

(2) *If* $\Psi \mid^{\overrightarrow{\gamma_1},\delta_0,\ldots,\delta_k \blacktriangleleft_k^{\gamma_2}}\vdash_R^{\overrightarrow{\gamma_1},\gamma_2} e : A$, *then* $\Psi, \gamma_2(:\succeq \delta_0/:\supseteq \delta_k) \mid^{\overrightarrow{\gamma_1},\gamma_2} \varepsilon \vdash_R^{\overrightarrow{\gamma_1},\gamma_2} e : A$.

(3) *If* $\Psi \mid^{\overrightarrow{\gamma_1},\gamma_2} x{:}^{\gamma_3} A_1 \vdash_R^{\overrightarrow{\gamma_1},\gamma_3} e : A_2$ *and* $y \notin \mathbf{Dom}_V(\Psi)$, *then* $\Psi, y : A_1 \mathbin{@} \gamma_3 :\succeq \gamma_2 \mid^{\overrightarrow{\gamma_1},\gamma_3} \varepsilon \vdash_{R,x:=y}^{\overrightarrow{\gamma_1},\gamma_3} e : A_2$.

(4) *If* $\Psi \mid^{\overrightarrow{\delta_1},\gamma_0'} [\gamma_1 :\succeq \gamma_1', \ldots, \gamma_k :\succeq \gamma_k']^{\gamma_0} \vdash_R^{\overrightarrow{\delta_1},\gamma_0} e : A$, *then* $\Psi, [\gamma_0 :\succeq \gamma_0', \ldots, \gamma_k :\succeq \gamma_k'] \mid^{\overrightarrow{\delta_1},\gamma_0} \varepsilon \vdash_R^{\overrightarrow{\delta_1},\gamma_0} e : A$.

Definition 1 (Disjointness of Classifiers). *We write* $\Psi \vdash \overrightarrow{\gamma_1} \# \overrightarrow{\gamma_2}$ *if* $\Psi \vdash \gamma_2' \subseteq \gamma_1'$ *does not hold for any* $\gamma_1' \in \{\overrightarrow{\gamma_1}\}$ *and* $\gamma_2' \in \{\overrightarrow{\gamma_2}\}$.

Lemma 9 (Demotion).

(1) *Suppose* $\Psi \mid^{\overrightarrow{\gamma_1}^+,\gamma_2,\gamma_3} \varepsilon \vdash_{\mathbf{id}}^{\overrightarrow{\gamma_1}^+,\gamma_2,\gamma_3} e : A$. *If* $\Psi \vdash \gamma_3 \subseteq \gamma_2$, *then* $\Psi \mid^{\overrightarrow{\gamma_1}^+,\gamma_2 \blacktriangleright \gamma_3}\vdash_{\mathbf{id}}^{\overrightarrow{\gamma_1}^+,\gamma_2,\gamma_3} e : A$.

(2) *Suppose* $\Psi \mid^{\overrightarrow{\gamma_1},\gamma_2} \varepsilon \vdash_{\mathbf{id}}^{\overrightarrow{\gamma_1},\gamma_2} e : A$ *and* $\gamma_2(:\succeq \delta_0/:\supseteq \delta_k) \in \Psi$ *and* $\Psi \vdash \overrightarrow{\gamma_1} \# \gamma_2$. *If* $\gamma_4 \notin \mathbf{Dom}_C(\Psi)$, *then* $\Psi \mid^{\overrightarrow{\gamma_1},\delta_0,\ldots,\delta_k \blacktriangleleft_k^{\gamma_4}}\vdash_{\mathbf{id}}^{\overrightarrow{\gamma_1},\gamma_4} e : A[\gamma_2 \coloneqq \gamma_4]$.

(3) *Suppose* $\Psi \mid^{\overrightarrow{\gamma_1},\gamma_2} \varepsilon \vdash_{\mathbf{id}}^{\overrightarrow{\gamma_1},\gamma_2} e : A_2$ *and* $x : A_1 \mathbin{@} \gamma_2 :\succeq \gamma_4 \in \Psi$ *and* $\Psi \vdash \overrightarrow{\gamma_1} \# \gamma_2$. *If* $\gamma_5 \notin \mathbf{Dom}_C(\Psi)$, *then* $\Psi \mid^{\overrightarrow{\gamma_1},\gamma_4} x{:}^{\gamma_5} A_1[\gamma_2 \coloneqq \gamma_5] \vdash_{\mathbf{id}}^{\overrightarrow{\gamma_1},\gamma_5} e : A_2[\gamma_2 \coloneqq \gamma_5]$.

(4) *Let* $\sigma = \gamma_0 \coloneqq \delta_0', \ldots, \gamma_k \coloneqq \delta_k'$. *Suppose* $\Psi \mid^{\overrightarrow{\delta_1}^+,\gamma_0} \varepsilon \vdash_{\mathbf{id}}^{\overrightarrow{\delta_1}^+,\gamma_0} e : A$, *and* $[\gamma_0 :\succeq \gamma_0', \ldots, \gamma_k :\succeq \gamma_k'] \in \Psi$ *and* $\Psi \vdash \overrightarrow{\delta_1}^+ \# \gamma_0, \ldots, \gamma_k$. *If* $\delta_i' \notin \mathbf{Dom}_C(\Psi)$ *for* $0 \le i \le k$, *then* $\Psi \mid^{\overrightarrow{\delta_1}^+,\gamma_0'} [\delta_1' :\succeq \gamma_1', \ldots, \delta_k' :\succeq \gamma_k']^{\delta_0'} \vdash_{\mathbf{id}}^{\overrightarrow{\delta_1}^+,\delta_0'} e : A[\sigma]$.

With these lemmas, we can prove type soundness of the runtime type system with respect to the definitional interpreter. The proof proceeds by induction on the derivation of the given expression typing and the step limit k_2.

Theorem 1 (Soundness of Runtime Typing).

(1) *If* $\Psi_1 \mid^{\gamma_1} \varepsilon \vdash_R^{\gamma_1} e^0 : A, \Psi_1 \mid \Theta_1 \vdash^{\gamma_1} E$ **wf** *and* $\Psi_1 \vdash S_1 : \Theta_1$, *then either of the following holds.*
- $\mathcal{V}^{\mathrm{rt}}(e^0, E, R, \mathbf{Dom}_V(\Psi_1), S_1, k_2) = \mathbf{Timeout}$
- $\mathcal{V}^{\mathrm{rt}}(e^0, E, R, \mathbf{Dom}_V(\Psi_1), S_1, k_2) = \mathbf{Done}\,(v, \mathbf{Dom}_V(\Psi_2), S_2)$, $\Psi_2 \mid \Theta_2 \vdash^{\gamma_1} v : A$ **val** *and* $\Psi_2 \vdash S_2 : \Theta_2$ *for some* v, Ψ_2 *and* Θ_2 *such that* $\Psi_1 \subseteq \Psi_2$ *and* $\Theta_1 \subseteq \Theta_2$.

(2) *If* $\Psi_1 \mid^{\gamma_0,\ldots,\gamma_{k_1}{}^+} \varepsilon \vdash_R^{\gamma_0,\ldots,\gamma_{k_1}{}^+} e_1{}^{k_1{}^+} : A, \Psi_1 \mid \Theta_1 \vdash^{\gamma_0} E$ **wf** *and* $\Psi_1 \vdash S_1 : \Theta_1$, *then either of the following holds.*
- $\mathcal{V}^{\mathrm{fut}}(k_1, e_1{}^{k_1{}^+}, E, R, \mathbf{Dom}_V(\Psi_1), S_1, k_2) = \mathbf{Timeout}$
- $\mathcal{V}^{\mathrm{fut}}(k_1, e_1{}^{k_1{}^+}, E, R, \mathbf{Dom}_V(\Psi_1), S_1, k_2) = \mathbf{Done}\,(e_2{}^{k_1}, \mathbf{Dom}_V(\Psi_2), S_2)$, $\Psi_2 \mid^{\gamma_1,\ldots,\gamma_{k_1}{}^+} \varepsilon \vdash_{\mathbf{id}}^{\gamma_1,\ldots,\gamma_{k_1}{}^+} e_2{}^{k_1} : A$ *and* $\Psi_2 \vdash S_2 : \Theta_2$ *for some* e_2, Ψ_2 *and* Θ_2 *such that* $\Psi_1 \subseteq \Psi_2$ *and* $\Theta_1 \subseteq \Theta_2$.

Type soundness of the surface type system then follows as a special case of the theorem above, by instantiating Ψ_1 to ε, γ_1 to $!$, and Θ_1 to ε.

Corollary 1 (Soundness of Surface Typing). *If* $\varepsilon \vdash^{!} e^0 : A$, *then either of the following holds.*

- $\mathcal{V}^{\mathrm{rt}}(e^0, \varepsilon, \mathbf{id}, \varepsilon, \varepsilon, k_2) = \mathbf{Timeout}$
- $\mathcal{V}^{\mathrm{rt}}(e^0, \varepsilon, \mathbf{id}, \varepsilon, \varepsilon, k_2) = \mathbf{Done}\,(v, N, S)$ *where* $\Psi \mid \Theta \vdash^{!} v : A$ **val**, $\mathbf{Dom}_V(\Psi) = N$ *and* $\Psi \vdash S : \Theta$ *for some* Ψ *and* Θ.

7.2 Safety of Offline Code Generation

As another important property, we prove the safety of offline code generation. This property states that the content of generated code remains well-typed. In particular, we expect that generated code does not depend on runtime objects such as store or reserved names, which is also discussed in earlier staged calculi with mutable state [19, 21, 42]. It is clear that we can discard the store, since expressions do not directly depend on locations. It is less trivial if we can discard reserved names, since they can be used in generated code. However, we can show that generated code will not use reserved names if it is well-typed at the initial classifier, as stated in the following lemma.

Lemma 10. *If* $\Psi \mid^! \varepsilon \vdash^!_{\mathbf{id}} e^0 : A$, *then* $\varepsilon \mid^! \varepsilon \vdash^!_{\mathbf{id}} e^0 : A$.

We can then prove safety of offline code generation, stated as follows.

Theorem 2 (Safety of Offline Code Generation). *If* $\varepsilon \vdash^! {e_1}^0 : \langle A \rangle^!$ *and* $\mathcal{V}^{\mathrm{rt}}({e_1}^0, \varepsilon, \mathbf{id}, \varepsilon, \varepsilon, k_2) = \mathbf{Done}\,(v, N, S)$, *then* $v = \langle {e_2}^0 \rangle$ *and* $\varepsilon \vdash^! {e_2}^0 : A$ *for some* ${e_2}^0$.

7.3 Mechanization in Rocq

We mechanized in Rocq the definitions and metatheory of our calculus presented in Sections 6 and 7. In particular, the mechanization establishes the two main results of the paper: Theorems 1 and 2. The formalization comprises 66,201 lines of Rocq source, much of which was generated from the definitions and theorem statements in the paper and its supplementary material using OpenAI Codex with GPT-5.5. The authors reviewed and corrected the generated code, checked the complete development with Rocq, and verified its correspondence with the source material. They take full responsibility for its correctness.

We use the locally nameless representation with cofinite quantification over fresh names [3] for binders introduced by polymorphic classifiers. We do not use such a representation for object-language variables: they are represented directly as names, and their binding structure is managed by the operational semantics through fresh-name generation and renaming environments. As a result, the mechanization does not require reasoning about alpha-equivalence or substitution for object-language variables; locally nameless reasoning is needed only for classifier binders.

8 Related Work

8.1 Earlier proposals with Refined Environment Classifiers

Earlier REC-based systems are restricted to two-level staging and support neither runtime execution nor CSP [14, 19, 20, 28]. In that setting, classifiers need only be tracked at a single stage, simplifying the type systems. In contrast, supporting multi-level staging, `run`, and scoped CSP requires typing contexts and judgments that track scoping dependencies across multiple stages. Classifier polymorphism poses an additional challenge: although the need for it had previously been noted informally, integrating it into earlier systems was nontrivial.

To address these challenges, we redesign typing contexts and judgments to manage scoping information uniformly across multiple stages. The resulting system supports multi-level code generation, `run`, and scoped CSP, using the visibility relation to capture the classifier scoping required for safe nested code generation. The same context structure yields natural surface typing rules for polymorphic classifiers simply by adding the corresponding context form, an advantage over earlier REC proposals.

8.2 Typing Disciplines for MetaML-style Code Generation with Effects

Early work on multi-stage programming contrasted two approaches to typing MetaML-style code generation. The type $\bigcirc A$ of $\lambda^{\bigcirc}$ represents open code and accommodates code with free variables,

but supports neither runtime execution nor effectful code generation [10, 11]. The type $\Box A$ of $\lambda^{\Box}$ represents only closed code, but safely supports runtime execution and effects because scope extrusion cannot occur [4, 12]. The need for both capabilities motivated several systems combining the two types [6, 23, 45]. Among them, Calcagno et al. [6] proposed a sound system with mutable state that permits only closed code to be executed or stored, preventing scoping errors. Rather than combining separate types for open and closed code, *environment-classifier*[1] systems annotate a single code type with staging information, distinguishing potentially open code valid at a particular stage from closed code portable across stages [5, 38]. This supports safe runtime execution and could likely accommodate mutable state by restricting storage to closed code, though this has not been formally studied.

Xie et al. [42] and Li et al. [21] formalized MetaML-style typed calculi with mutable state and proved the safety of offline code generation; Li et al. additionally support runtime execution. Both focus on the interaction between mutable state and offline code generation, showing that generated code is independent of generation-time state, such as store locations. Consistent with this focus, they restrict mutable references to ground types, simplifying their formalizations but excluding references containing code and hence scope extrusion through mutable state.

Contextual types are another approach to typing open code, though usually in calculi unlike MetaML; see Section 8.5. One exception is $\lambda^{[]}$, a MetaML-style calculus with mutable state whose contextual code types explicitly list the free variables available to a code fragment [31]. However, they do not account for visibility and thus admit well-typed programs exhibiting scope extrusion, as explained in Section 2.3 and demonstrated in the supplementary material. This limitation appears difficult to address in a contextual type system.

8.3 Reasoning about Dynamic Aspects of Effectful Code Generation

Practical multi-stage languages tend to adopt simpler type systems based on $\lambda^{\bigcirc}$, allowing dynamic errors [17, 32, 35]. This motivates studying the dynamic behavior of effectful code generation, including erroneous behavior such as scope extrusion.

Moggi and Fagorzi's combinator-based calculus [22] makes scope extrusion through mutable state observable by treating fresh-name generation as an effect and generated names as part of the computation state. Combinator-based REC calculi adopt a similar view [14, 19, 28], but further distinguish harmful scope extrusion, where the final result of code generation contains a free generated name.

Lee et al. [20] study dynamic detection of scope extrusion in code generation with effect handlers, exploring the trade-off between precision and performance. Such mechanisms could complement our static discipline in practical extensions, particularly gradual typing [7, 43], and could be incorporated into our operational semantics.

Yin et al. [44] proposed a typed multi-stage calculus with mutable state and strong type-safety guarantees. They also developed a fully abstract trace model based on operational game semantics, providing a semantic foundation for effectful code generation. We note, however, that while they present their language as MetaML-style, we classify it as a contextual-type-based calculus; see Section 8.5.

Lightweight Modular Staging (LMS) [32] is a practical Scala framework that addresses scope extrusion through language design: it encapsulates effects such as deduplication and code motion within its staging primitives, allowing programmers to write staged programs in a pure style while

[1]Despite the similarity in terminology, environment classifiers and refined environment classifiers serve distinct purposes: the former track the staging dependencies of generated code, whereas the latter track its scoping dependencies.

benefiting from effectful code generation. This design makes scope extrusion less likely, though not impossible.

8.4 Cross-Stage Persistence

Cross-stage persistence (CSP) allows variables or values from one stage to be used at later stages, but admits several implementations. Xie et al. [42] classify them as *Value-based CSP*, *Path-based CSP*, and *Heap-based CSP*. Heap-based CSP stores values in the heap and embeds their locations in generated code for resolution at runtime; it corresponds to the original notion of CSP in MetaML [38, 39] and to what we call traditional CSP in this paper.

Our *scoped CSP* allows generated code to contain variable names resolved at runtime against the surrounding lexical environment. This mechanism is not new: a similar one appears in the staged C variant ‘C [30], showing that scoped CSP arises naturally from runtime execution under lexical scoping. Although more restrictive than heap-based CSP, scoped CSP supports similar use cases and fits naturally with our R2EC-based typing discipline (Section 3). By contrast, heap-based CSP requires non-trivial type-system extensions to ensure the safety of offline code generation [13, 21]. Comparing their use cases and type systems more closely remains future work.

8.5 MetaML-style MSP and Contextual-Types-based MSP

Besides MetaML-style MSP, another approach to MSP is based on contextual types [16, 25, 26, 29]. From a syntactic perspective, the main difference between the two approaches lies in how they treat free variables in code fragments. In MetaML-style MSP, free variables in code fragments can be bound by the surrounding context. In contrast, in contextual-types-based calculi, free variables in code fragments are bound by the context annotation on a quotation or contextual type, rather than directly by the surrounding context. An advantage of the contextual-types-based approach is that it is largely orthogonal to other features, since open code fragments are not lexically bound by surrounding contexts. For example, several contextual-types-based calculi have been proposed in combination with effectful features such as mutable state [16, 44] and session-typed communication [2, 33]. By contrast, subtle interactions between MetaML-style MSP and effectful computation have long been a challenge. From this perspective, the main result of this paper is significant because it provides a typing discipline for MetaML-style MSP that composes cleanly with mutable state while ensuring safety.

Since the two approaches offer different strengths, they can be hybridized to combine their advantages; see Chiang and Xie [8]. Thus, they are better viewed as complementary rather than competing approaches to MSP.

8.6 Logical Foundations of Multi-Stage Programming

Type systems for multi-stage programming are known to have logical foundations in modal logic [11, 12, 27, 40]. In a recent preprint, Murase and Maniwa [24] proposed *bounded modal logic* as a logical foundation for MetaML-style metaprogramming, generalizing the S4 modal lambda calculus to make the scope dependencies of code fragments explicit. Their formalization is similar to ours in that it uses classifiers to annotate modal types and supports polymorphism over classifiers. However, it does not account for scope visibility and would therefore admit programs that are undesirable in our setting. Elaborating their logic to provide a logical foundation for our type system is an interesting direction for future work.

9 Conclusion and Future Work

In this paper, we proposed a novel type system based on refined environment classifiers (RECs) for MetaML-style multi-stage programming with mutable state. We showed that extending RECs

beyond earlier settings is nontrivial: typing multi-level programs requires tracking not only variable scopes but also the scopes of classifiers themselves. We formalized this refinement through reachability and visibility and called the resulting typing discipline *refined² environment classifiers* (R2ECs). We further incorporated polymorphic classifiers to support more general and reusable staged code generators.

For the full calculus, we proved type soundness and the safety of offline code generation, demonstrating that R2ECs support expressive, effectful multi-stage programming while preventing harmful scope extrusion. We also presented a definitional-interpreter semantics that incorporates scoped CSP and renaming environments, providing an operational account of lexical scoping and a basis for future semantic studies. Taken together, these contributions establish a theoretical foundation for effectful MetaML-style MSP supporting multi-level code generation, runtime execution, scoped CSP, and classifier polymorphism.

As future work, we plan to investigate practical extensions, including subtyping [19, 31], type inference [5, 19], analytic metaprogramming [8, 29, 35], effect handlers [14, 20], and module systems [9, 36, 42].

Data-Availability Statement

We provide a supplementary document and an artifact package. The supplementary document contains material omitted from the main text, including full definitions, paper proofs of the metatheoretic properties, and counterexamples showing that $\lambda^{[]}$, a MetaML-style calculus with mutable state [31], is not type sound, as discussed in Section 8. The supplementary document has been uploaded separately and is also included in the artifact package described below.

The artifact package contains a browser-based implementation of our calculus and a Rocq mechanization of the formal definitions and proofs presented in the supplementary document. The implementation supports interactive exploration of the surface type system and the definitional interpreter. We provide Docker images for both components to facilitate reproducibility. The artifact package is available on Zenodo at https://doi.org/10.5281/zenodo.21468038. For convenience, the implementation can also be tried directly through an interactive web demo at https://www.fos.kuis.kyoto-u.ac.jp/~murase/r2ecs-demo/.

References

[1] Nada Amin and Tiark Rompf. 2017. Type Soundness Proofs with Definitional Interpreters. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL '17)*. Association for Computing Machinery, New York, NY, USA, 666–679. doi:10.1145/3009837.3009866

[2] Pedro Ângelo, Atsushi Igarashi, Yuito Murase, and Vasco T. Vasconcelos. 2026. Contextual Metaprogramming for Session Types. In *Programming Languages and Systems*, Robbert Krebbers (Ed.). Springer Nature Switzerland, Cham, 13–41. doi:10.1007/978-3-032-22720-1_2

[3] Brian Aydemir, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack, and Stephanie Weirich. 2008. Engineering Formal Metatheory. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '08)*. Association for Computing Machinery, New York, NY, USA, 3–15. doi:10.1145/1328438.1328443

[4] Cristiano Calcagno, Eugenio Moggi, and Walid Taha. 2000. Closed Types as a Simple Approach to Safe Imperative Multi-stage Programming. In *Automata, Languages and Programming*. Springer, Berlin, Heidelberg, 25–36. doi:10.1007/3-540-45022-X_4

[5] Cristiano Calcagno, Eugenio Moggi, and Walid Taha. 2004. ML-Like Inference for Classifiers. In *Programming Languages and Systems*, David Schmidt (Ed.), Vol. 2986. Springer, Berlin, Heidelberg, 79–93. doi:10.1007/978-3-540-24725-8_7

[6] Cristiano Calcagno, Walid Taha, Liwen Huang, and Xavier Leroy. 2003. Implementing Multi-stage Languages Using ASTs, Gensym, and Reflection. In *Generative Programming and Component Engineering*. Springer, Berlin, Heidelberg, 57–76. doi:10.1007/978-3-540-39815-8_4

[7] Tianyu Chen, Darshal Shetty, Jeremy G. Siek, Chao-Hong Chen, Weixi Ma, Arnaud Venet, and Rocky Liu. 2025. Gradual Metaprogramming. In *Proceedings of the 10th ACM SIGPLAN International Workshop on Type-Driven Development (TyDe '25)*. Association for Computing Machinery, New York, NY, USA, 1–14. doi:10.1145/3759538.3759650

[8] Tsung-Ju Chiang and Ningning Xie. 2025. Multi-Stage Programming with Splice Variables. *Proceedings of the ACM on Programming Languages* 9, ICFP (Aug. 2025), 249:400–249:434. doi:10.1145/3747518

[9] Tsung-Ju Chiang, Jeremy Yallop, Leo White, and Ningning Xie. 2024. Staged Compilation with Module Functors. *Proceedings of the ACM on Programming Languages* 8, ICFP (Aug. 2024), 260:693–260:727. doi:10.1145/3674649

[10] Rowan Davies. 1996. A Temporal-Logic Approach to Binding-Time Analysis. In *Proceedings 11th Annual IEEE Symposium on Logic in Computer Science*. 184–195. doi:10.1109/LICS.1996.561317

[11] Rowan Davies. 2017. A Temporal Logic Approach to Binding-Time Analysis. *J. ACM* 64, 1 (March 2017), 1:1–1:45. doi:10.1145/3011069

[12] Rowan Davies and Frank Pfenning. 2001. A Modal Analysis of Staged Computation. *J. ACM* 48, 3 (May 2001), 555–604. doi:10.1145/382780.382785

[13] Yuichiro Hanada and Atsushi Igarashi. 2014. On Cross-Stage Persistence in Multi-Stage Programming. In *Functional and Logic Programming (Lecture Notes in Computer Science)*, Michael Codish and Eijiro Sumii (Eds.). Springer International Publishing, Cham, 103–118. doi:10.1007/978-3-319-07151-0_7

[14] Kanaru Isoda, Ayato Yokoyama, and Yukiyoshi Kameyama. 2024. Type-Safe Code Generation with Algebraic Effects and Handlers. In *Proceedings of the 23rd ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE '24)*. Association for Computing Machinery, New York, NY, USA, 53–65. doi:10.1145/3689484.3690731

[15] Yukiyoshi Kameyama, Oleg Kiselyov, and Chung-chieh Shan. 2015. Combinators for Impure yet Hygienic Code Generation. *Science of Computer Programming* 112 (Nov. 2015), 120–144. doi:10.1016/j.scico.2015.08.007

[16] Ik-Soon Kim, Kwangkeun Yi, and Cristiano Calcagno. 2006. A Polymorphic Modal Type System for Lisp-like Multi-Staged Languages. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '06)*. Association for Computing Machinery, New York, NY, USA, 257–268. doi:10.1145/1111037.1111060

[17] 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

[18] Oleg Kiselyov. 2026. MetaOCaml: Ten Years Later – System Description. *Science of Computer Programming* 250 (May 2026), 103397. doi:10.1016/j.scico.2025.103397

[19] Oleg Kiselyov, Yukiyoshi Kameyama, and Yuto Sudo. 2016. Refined Environment Classifiers. In *Programming Languages and Systems (Lecture Notes in Computer Science)*, Atsushi Igarashi (Ed.). Springer International Publishing, Cham, 271–291. doi:10.1007/978-3-319-47958-3_15

[20] Michael Lee, Ningning Xie, Oleg Kiselyov, and Jeremy Yallop. 2026. Handling Scope Checks: A Comparative Framework for Dynamic Scope Extrusion Checks. *Proceedings of the ACM on Programming Languages* 10, POPL (Jan. 2026), 39:1123–39:1152. doi:10.1145/3776681

[21] Ka Wing Li, Maite Kramarz, Ningning Xie, and Jeremy Yallop. 2026. Mechanised Semantics of Multi-stage Programming. *Proceedings of the ACM on Programming Languages* 10, OOPSLA1 (April 2026), 152:1654–152:1681. doi:10.1145/3798260

[22] Eugenio Moggi and Sonia Fagorzi. 2003. A Monadic Multi-stage Metalanguage. In *Foundations of Software Science and Computation Structures*, Andrew D. Gordon (Ed.). Springer, Berlin, Heidelberg, 358–374. doi:10.1007/3-540-36576-1_23

[23] Eugenio Moggi, Walid Taha, Zine El-Abidine Benaissa, and Tim Sheard. 1999. An Idealized MetaML: Simpler, and More Expressive. In *Programming Languages and Systems*. Springer, Berlin, Heidelberg, 193–207. doi:10.1007/3-540-49099-X_13

[24] Yuito Murase and Akinori Maniwa. 2026. Bounded Modal Logic: Explicit Scope Dependencies in Multi-Stage Programming. arXiv:2602.09462 [cs.LO] doi:10.48550/arXiv.2602.09462

[25] Yuito Murase, Yuichi Nishiwaki, and Atsushi Igarashi. 2023. Contextual Modal Type Theory with Polymorphic Contexts. In *Programming Languages and Systems (Lecture Notes in Computer Science)*, Thomas Wies (Ed.). Springer Nature Switzerland, Cham, 281–308. doi:10.1007/978-3-031-30044-8_11

[26] Aleksandar Nanevski and Frank Pfenning. 2005. Staged Computation with Names and Necessity. *Journal of Functional Programming* 15, 6 (Nov. 2005), 893–939. doi:10.1017/S095679680500568X

[27] Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. 2008. Contextual Modal Type Theory. *ACM Transactions on Computational Logic* 9, 3 (June 2008), 23:1–23:49. doi:10.1145/1352582.1352591

[28] Junpei Oishi and Yukiyoshi Kameyama. 2017. Staging with Control: Type-Safe Multi-Stage Programming with Control Operators. In *Proceedings of the 16th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE 2017)*. Association for Computing Machinery, New York, NY, USA, 29–40. doi:10.1145/3136040.3136049

[29] 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), 13:1–13:33. doi:10.1145/3158101

[30] Massimiliano Poletto, Wilson C. Hsieh, Dawson R. Engler, and M. Frans Kaashoek. 1999. 'C and Tcc: A Language and Compiler for Dynamic Code Generation. *ACM Transactions on Programming Languages and Systems* 21, 2 (March 1999), 324–369. doi:10.1145/316686.316697

[31] Morten Rhiger. 2012. Staged Computation with Staged Lexical Scope. In *Programming Languages and Systems*, Helmut Seidl (Ed.). Springer, Tallinn, Estonia, 559–578. doi:10.1007/978-3-642-28869-2_28

[32] Tiark Rompf. 2012. *Lightweight Modular Staging and Embedded Compilers : Abstraction without Regret for High-Level High-Performance Programming*. Ph. D. Dissertation. EPFL. doi:10.5075/epfl-thesis-5456

[33] Chuta Sano, Deepak Garg, Ryan Kavanagh, Brigitte Pientka, and Bernardo Toninho. 2025. Fusing Session-Typed Concurrent Programming into Functional Programming. *Proceedings of the ACM on Programming Languages* 9, ICFP (Aug. 2025), 250:435–250:462. doi:10.1145/3747519

[34] Jeremy Siek. 2013. Type Safety in Three Easy Lemmas. https://siek.blogspot.com/2013/05/type-safety-in-three-easy-lemmas.html Online; accessed 2026-02-19.

[35] Nicolas Stucki. 2023. *Scalable Metaprogramming in Scala 3*. Ph. D. Dissertation. Swiss Federal Institute of Technology in Lausanne, Lausanne. doi:10.5075/epfl-thesis-8257

[36] Takashi Suwa and Atsushi Igarashi. 2024. An ML-Style Module System for Cross-Stage Type Abstraction in Multi-stage Programming. In *Functional and Logic Programming*, Jeremy Gibbons and Dale Miller (Eds.). Springer Nature, Singapore, 237–272. doi:10.1007/978-981-97-2300-3_13

[37] Walid Taha. 1999. A Sound Reduction Semantics for Untyped CBN Multi-Stage Computation. Or, the Theory of MetaML Is Non-Trival. In *Proceedings of the 2000 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '00)*. Association for Computing Machinery, New York, NY, USA, 34–43. doi:10.1145/328690.328697

[38] Walid Taha and Michael Florentin Nielsen. 2003. Environment Classifiers. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '03)*. Association for Computing Machinery, New York, NY, USA, 26–37. doi:10.1145/604131.604134

[39] Walid Taha and Tim Sheard. 2000. MetaML and Multi-Stage Programming with Explicit Annotations. *Theoretical Computer Science* 248, 1 (Oct. 2000), 211–242. doi:10.1016/S0304-3975(00)00053-0

[40] Takeshi Tsukada and Atsushi Igarashi. 2009. A Logical Foundation for Environment Classifiers. In *Typed Lambda Calculi and Applications (Lecture Notes in Computer Science)*, Pierre-Louis Curien (Ed.). Springer, Berlin, Heidelberg, 341–355. doi:10.1007/978-3-642-02273-9_25

[41] Andrew K. Wright and Matthias Felleisen. 1994. A Syntactic Approach to Type Soundness. *Information and Computation* 115, 1 (1994), 38–94. doi:10.1006/INCO.1994.1093

[42] Ningning Xie, Leo White, Olivier Nicole, and Jeremy Yallop. 2023. MacoCaml: Staging Composable and Compilable Macros. *Proceedings of the ACM on Programming Languages* 7, ICFP (Aug. 2023), 209:604–209:648. doi:10.1145/3607851

[43] Hiroto Yaguchi and Yukiyoshi Kameyama. 2025. Staged Gradual Typing. In *Proceedings of the 24th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE '25)*. Association for Computing Machinery, New York, NY, USA, 94–104. doi:10.1145/3742876.3742880

[44] Haoxuan Yin, Andrzej S. Murawski, and C.-H. Luke Ong. 2026. Contextual MetaML: Syntax and Full Abstraction. In *41st Annual Symposium on Logic in Computer Science (LICS 2026) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 380)*, Claudia Faggian and Joost-Pieter Katoen (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 83:1–83:27. doi:10.4230/LIPIcs.LICS.2026.83

[45] Yosihiro Yuse and Atsushi Igarashi. 2006. A Modal Type System for Multi-Level Generating Extensions with Persistent Code. In *Proceedings of the 8th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP '06)*. Association for Computing Machinery, New York, NY, USA, 201–212. doi:10.1145/1140335.1140360