MAYUZIN: Runtime Generative and Analytical Metaprogramming

Ya Mone Zin[1,a)] Yukiyoshi Kameyama[2,b)]

Received: xx xx, xxxx, Accepted: xx xx, xxxx

Abstract: This paper introduces Mayuzin, a new multi-stage programming (MSP) language designed to integrate both generative and analytical capabilities for run-time metaprogramming while preserving type safety. While code generation and code analysis play complementary roles in metaprogramming practice, theoretical foundation for code analysis has been largely overlooked in the literature, with a notable exception by Stucki et al., who studied code analysis in compile-time metaprogramming. In this study, we fill this gap by presenting a metaprogramming language that supports both runtime code generation and analysis, ensuring type safety throughout.

Mayuzin introduces three key features: (1) a code analysis framework that leverages pattern matching to enable dynamic code inspection and transformation, (2) integration of ML-style let polymorphism within the multi-stage setting using the Hindley-Milner type system as a foundation, and (3) support for manipulation of open codes, which is crucial for generating efficient code. Our design extends traditional MSP by providing robust facilities for runtime code analysis while aiming to maintain type guarantees across stages. We will illustrate Mayuzin's capabilities through examples of runtime code optimization in domain-specific applications, such as eliminating redundant computations in generated numerical code. These examples demonstrate how Mayuzin's unified approach to metaprogramming facilitates efficient domain-specific optimizations, with rigorous proof of type safety, ensuring safe runtime code transformations.

Keywords: Runtime metaprogramming, Code generation, Code analysis, Type safety, Open-code manipulation, Environment classifier

1. Introduction

Multi-stage programming (MSP) has emerged as a powerful paradigm for program generation and manipulation, enabling programmers to write code that generates, analyzes, and executes code at different stages [4], [16]. Originating from partial evaluation and runtime code generation systems [8], MSP empowers developers with explicit control over the timing of code generation and execution through staging annotations. This explicit control is key to improving performance and reducing runtime overhead, making MSP a significant advancement in modern programming methodologies.

Studies on MSP languages have focused on the static guarantee for type safety, as it subsumes that generated code is well-formed, well-scoped, and well-typed regardless of dynamic parameters. While fruitful theories and systems such as MetaOCaml have emerged as the results, most of them stayed in the **purely generative approach** to metaprogramming,[*1] where only the construction and composition of code fragments are allowed [13]. When introspection and optimization of generated code are required, one needs to design another layer to transform the code, limiting the applicability of MSP languages.

For example, an MSP version of the power function `spower` can be written as:

[1] Master's Program in Computer Science, University of Tsukuba, Tsukuba 305–8573, Japan
[2] Department of Computer Science, University of Tsukuba, Tsukuba 305–8573, Japan
[a)] yamone@logic.cs.tsukuba.ac.jp
[b)] kameyama@acm.org

[*1] There are exceptions that studied non-purely generative approaches [7], [14]. We will discuss them later in this paper.

```
let rec spower n x =
  if n = 0 then <1>
  else <~x * ~(spower (n−1) x)>
```

where we use **quotation** (written as $\langle e \rangle$) which constructs a code), and **anti-quotation** (written as $\sim e$) which splices a code into another code. Executing an expression `<fun x → ~(spower 5 <x>)>` yields the code `<fun x → x * x * x * x * x * 1>`. This generated code can be executed within the language by **run** (written as `run` e). For instance, `run <fun x → ~(spower 5 <x>)>` would produce an ordinary function equivalent to `fun x → x * x * x * x * x * 1`. While the computation above demonstrates how MSP can eliminate the run-time overhead, the result is not fully optimal, as there remains a redundant `... * 1` operation. Although modern compilers can eliminate such simple redundancies, it is often not feasible to remove all redundant operations without domain-specific knowledge, which underscores why we use MSP languages.

While code generation enables the creation of optimized, specialized program variants, **code analysis** allows for the inspection and transformation of code structures to enhance adaptability and enable runtime optimizations. Combining these capabilities, however, has remained a significant challenge due to the complexities involved in maintaining type safety and hygiene during runtime transformations.

In this paper, we present Mayuzin, a novel MSP language that aims to integrate generative and analytical capabilities for runtime metaprogramming within a type-safe framework. Mayuzin extends the traditional MSP paradigm by incorporating advanced features such as **let polymorphism**, **environment classifiers**, and the **run primitive**, facilitating sophisticated runtime transformations while maintaining rigorous type safety. Our work seeks to address the existing gap in supporting both runtime code generation and analysis, thereby enhancing code efficiency and adaptability across diverse computational environments.

Analytical metaprogramming in Mayuzin is realized by **pattern matching for code**. To illustrate its usage, we show a simple optimizer[*2] in Mayuzin as follows:

```
let rec opt expr1 =
  match expr1 with
  | < $x * 0 >  → < 0 >
  | < 0 * $x >  → < 0 >
  | < $x * 1 >  → opt x
  | < 1 * $x >  → opt x
  | < $x * $y > → < ~(opt x) * ~(opt y) >
  | _           → expr1
```

The code is strikingly simple and intuitive: it matches the expression `expr1` (which is supposed to be a nested multiplication) against code patterns such as `<$x * 0>`, and then returns a simplified result if one of the arguments of the multiplication is `0` or `1`. (Later, we will explain the code in detail.) For instance, `opt <2 * ((3 * 1) * (4 * 1))>` evaluates to `<2 * (3 * 4)>`, which shows that deeply nested occurrences of 1 may be eliminated. Furthermore, we can combine `opt` with `spower` to obtain an optimized code. For instance, `<fun x → ~(opt (spower 5 <x>))>` evaluates to code `<fun x → x * x * x * x * x * x>`, showing that `opt` effectively handles an open code (a code with free variables).

While optimizing generated code can yield significant improvements, using pattern matching for code has a potential risk of the scope extrusion problem. Suppose we can extract the body of a lambda abstraction in the code as follows:

```
let match_wrong expr1 =
  match expr1 with
  | <fun z → $x> → x
  | _            → expr1
```

Then `match_wrong <fun a → a + 7>` would evaluate to `<a + 7>`, which has a free variable `a` and does not compile safely. Obviously, we should not be able to extract a value with free variables. Stucki, Brachthäuser, and Odersky [14] proposed to use a higher-order pattern variable to extract a code under binders safely. The above erroneous code may be rewritten using a higher-order pattern variable `x` as follows:

```
let match_ok expr1 =
  match expr1 with
  | <fun z → $x(z)> → x <3>
  | _               → expr1
```

Here, `x` is a function that takes a code as an argument and returns the function body after substituting its argument for `z`. For instance, evaluating `match_ok <fun a → a + 7>` yields `<3 + 7>`, which has no free variables.

We have designed the surface-language syntax based on the statically typed language by Stucki et al., which has generative and analytical features for compile-time metaprogramming. A fundamental difference between their language and ours lies in the type systems. Since Mayuzin is a language for run-time metaprogramming which has the `run` primitive, a more involved type-theoretic tool is needed to ensure static type safety for Mayuzin, and we make essential use of environment classifiers by Taha and Nielsen [15].

[*2] The function `opt` is not an ideal optimizer, as it cannot fully reduce $\langle (0 * 2) * 3 \rangle$, but we can improve it easily.

Another important feature of Mayuzin is let polymorphism, which is used in ML-family languages. Let polymorphism also plays an important role in MSP languages, for instance, the function **eta** is a general combinator that turns a code-to-code function into a code of a function, defined as follows:

```
let eta f = < fun x → ~(f <x>) >
```

The expression `eta (spower 3)` evaluates to `< fun x → x * x * x * 1 >`. Polymorphism is essential for **eta** as a library function, since it should be applied to a variety of code-to-code functions of different types. We have added [*3] analytical metaprogramming to the ML-like let-polymorphic variant [4] of Taha and Nielsen's language, to obtain our language Mayuzin, and prove its type safety rigidly. That would open up a new paradigm of MSP languages, such as MetaOCaml, where we can freely manipulate generated code without losing type safety.

We are not the first to introduce pattern matching for code to a polymorphic multi-stage calculus for run-time metaprogramming. The language Mœbius [7] has these features with System-F style impredicative polymorphism. It is based on Contextual Modal Type Theory (CMTT) [12], in which a code type contains a typing context, leading to a very expressive type system. However, its expressiveness cannot be obtained without a price; CMTT needs an involved definition for dedicated substitutions, and its type system is way more complicated than other approaches, such as environment classifiers. Calcagno et al.'s type system [4], that underlies our type system, enjoys the principal-type property and has a complete type inference algorithm, that allowed Taha to design the first version of MetaOCaml.

Mayuzin is intended to be lightweight and practical, and, hence, we took the approach based on environment classifiers. While the authors of Mœbius [7] stated that "it [environment classifier] seems difficult to extend to support pattern on code.", the present paper shows that it is indeed possible in the framework of a let-polymorphic, run-time metaprogramming language without significantly complicating the type system.

Our contribution can be summarized as follows:

- We present a novel MSP language Mayuzin that combines generative and analytical metaprogramming, open-code manipulation, run-time code generation, and ML-style let polymorphism.
- We design a type system for Mayuzin and prove its type safety.
- We provide a proof-of-concept implementation for Mayuzin.

The rest of this paper is organized as follows. Sect. 2 presents the syntax of Mayuzin, and describes the operational semantics of pattern matching for code patterns. Sect. 3 introduces a type system of Mayuzin and an example of a type derivation. Its key properties and proof sketch are given in Sect. 4. Sect. 5 overviews our implementation of Mayuzin, and Sect. 6 compares our work with closely related work. Sect. 7 concludes the paper.

2. The Language Mayuzin

We introduce Mayuzin, a statically typed language for run-time code generation and code analysis.

2.1 Syntax

We assume that Var (ranged over by a), PatVar (ranged over by x and f), and Const (ranged over by c), resp. are mutually disjoint sets of variables, pattern variables, and constants, resp.

Fig. 1 defines the syntax of Mayuzin where e denotes an expression, v^i denotes a level-i value for a natural number i. We write v for v^i for some i. To avoid clutter, we have chosen a minimal language for our development: we only have two levels (two stages), hence the level i is either 0 or 1. Also, we consider pattern matching at the level 0 only.

An expression is either a variable a, a constant c, lambda abstraction $\lambda a.e$, application $e@e$, a let expression, a match expression, a quoted expression $\langle e \rangle$, an anti-quoted expression $\sim e$, or a run expression **run** e. The symbols used for quoted (and anti-quoted, resp.) expressions are called brackets (and escapes, resp.). In a match expression, $p \to e$ denotes a case where p is a pattern to be matched, and if it succeeds, its body e is executed. We assume that all match expressions has the last case $\_ \to e$ as the default case, which always succeeds in pattern matching.

A level-0 value v^0 is an ordinary value in lambda calculus with a code value $\langle v^1 \rangle$, while a level-1 value v^1 can be arbitrary expression except the anti-quoted expression $\sim e$.

Our language has standard patterns such as a pattern variable x, a constant pattern c, and a pair pattern (p, p) as well as a code pattern $\langle p \rangle$. For the pattern p in $\langle p \rangle$, not only standard patterns are allowed, but also other patterns such

[*3] Strictly speaking, our language is not an extension of theirs, since we do not have classifier polymorphism and more than two stages. However, we believe that extending our language to cover these features is not difficult.

$$
\begin{aligned}
e &::= a \mid c \mid \lambda a.e \mid e@e \mid (e,e) \mid \texttt{let}\ a = e\ \texttt{in}\ e \mid \texttt{match}\ e\ \texttt{with}\ (p \to e \mid \cdots \mid p \to e \mid \_ \to e) \mid \langle e \rangle \mid {\sim} e \mid \texttt{run}\ e \\
v^0 &::= c \mid \lambda a.e \mid (v^0, v^0) \mid \langle v^1 \rangle \\
v^1 &::= a \mid c \mid \lambda a.v^1 \mid v^1@v^1 \mid (v^1, v^1) \mid \texttt{let}\ a = v^1\ \texttt{in}\ v^1 \\
p &::= x \mid c \mid (p,p) \mid \langle p \rangle \mid \lambda x.p \mid p@p \mid \texttt{let}\ x = p\ \texttt{in}\ p \mid \$f(x, \cdots, x)
\end{aligned}
$$

Fig. 1 Syntax of Terms, Values, and Patterns

as $\lambda x.p$, $p@p$, $\texttt{let}\ x = p\ \texttt{in}\ p$, and $\$f(x, \cdots, x)$ may be used. Of particular interest is the pattern $\$f(x_1, \cdots, x_n)$ where x is an ordinary (first-order) pattern variable, and f is a higher-order pattern variable. This pattern matches with any value provided its free variables are $x_1, \cdots, x_n$ at most. For instance, $\langle \lambda x.\$f(x) \rangle$ is a code pattern in which a higher-order pattern variable f is used with a parameter x. This pattern matches any values in the form $\langle \lambda a.e \rangle$ even if e contains the variable a freely. On the other hand, the pattern $\langle \lambda x.\$f() \rangle$ does not match with the value $\langle \lambda a.a + 3 \rangle$, since the value matched against f is $a + 3$, which contains a freely, while the corresponding pattern variable `x` is not specified as a parameter of f. When a higher-order pattern variable has no parameters, we omit the parentheses, namely, $\$f$ denotes $\$f()$.

Variables are bound by lambda abstractions and pattern matching. The set of free variables in an expression, denoted as $\mathrm{FV}(X)$, is defined as usual. We consider alpha-equivalent terms as identical, and follow Barendregt convention in that all bound variables are mutually distinct and differ from free variables [2]. We write $\vec{x}$ for a sequence $x_1, \cdots, x_n$ where the length n is left implicit.

2.2 Pattern Matching

Pattern matching plays an essential role in Mayuzin, as it serves as the primary mechanism that can decompose (analyze) code values. Since the semantics of pattern matching is not as straightforward as one may think, we will investigate its semantics deeply in this section.

Fig. 2 presents an operational semantics of pattern matching. Given a level-i value v^i and a pattern p, $\mathcal{M}^i_\rho(v^i, p)$ denotes the result of pattern matching v against p, which is either a substitution θ (when the pattern matching succeeds) or fail (when the pattern matching fails). The term includes a renaming map ρ as an additional parameter, which will be explained later.

The result of successful pattern matching is a substitution θ, which is a finite map from pattern variables to values, and is denoted by $[x_1 \mapsto v_1, \cdots, x_k \mapsto v_k]$. The expression $X \cup Y$ in the third and fifth clauses denotes the union map of X and Y if both are substitutions ($X \neq \text{fail}$ and $Y \neq \text{fail}$) and the domain of X and the domain of Y are disjoint. If these conditions do not hold, $X \cup Y$ is fail.

Internal to the semantics, we use a renaming ρ, which maps term variables (such as a) to pattern variables (such as x). Note that the directions of the maps θ and ρ are opposite. We write the application of a substitution θ (and a renaming ρ, resp.) to X by $X\,\theta$ (and $X\,\rho$, resp.).

The first three clauses in Fig. 2 correspond to a pattern variable x, a constant pattern c, a pair pattern (p_1, p_2) and these clauses are the same as the standard semantics of pattern matching.

The fourth clause performs pattern matching against a code pattern $\langle p \rangle$ which is applicable only when $i = 0$: it first verifies whether the value v is a code-value, then performs pattern matching for its contents at the level 1.

The remaining clauses make sense only when $i = 1$, indicating that pattern matching is performed inside quasiquotation. For an application pattern $p_1@p_2$, it verifies whether the value has the same structure and performs pattern matching for each component. For an abstraction pattern $\lambda x.p$, it verifies whether the value matches the structure and proceeds with pattern matching on the abstraction body. We need to record that the bound variable x in the pattern corresponds to the bound variable a in the value, and therefore, this correspondence is added to ρ. The last clause is the most interesting case, where the pattern $\$f(x_1, \cdots, x_n)$ has a higher-order pattern variable f. This pattern intuitively represents a more involved pattern $\sim (f@\langle x_1 \rangle \cdots @\langle x_n \rangle)$, which indicates: the variable f is a level-0 variable denoting a function that takes n arguments, and returns a code that may contain the level-1 variables $x_1, \cdots, x_n$ freely. In other words, this pattern matches with a code value v, which must not contain level-1 bound variables other than $x_1, \cdots, x_n$. This consideration leads to the condition $(\mathrm{FV}(v))\,\rho \subseteq \{x_1, \cdots, x_n\}$ where we apply the renaming ρ to all members in $\mathrm{FV}(v)$, the set of free

$$\mathcal{M}^i_\rho(v,x) \coloneqq \begin{cases} [x \mapsto v] & \text{if } i = 0 \\ [\,] & \text{if } i > 0 \wedge v = a \wedge (x/a) \in \rho \\ \text{fail} & \text{otherwise} \end{cases}$$

$$\mathcal{M}^i_\rho(v,c) \coloneqq \begin{cases} [\,] & \text{if } v = c \\ \text{fail} & \text{otherwise} \end{cases}$$

$$\mathcal{M}^i_\rho(v,(p_1,p_2)) \coloneqq \begin{cases} \mathcal{M}^i_\rho(v_1,p_1) \cup \mathcal{M}^i_\rho(v_2,p_2) & \text{if } v = (v_1,v_2) \\ \text{fail} & \text{otherwise} \end{cases}$$

$$\mathcal{M}^0_\rho(v,\langle p\rangle) \coloneqq \begin{cases} \mathcal{M}^1_\rho(v_1,p) & \text{if } v = \langle v_1\rangle \\ \text{fail} & \text{otherwise} \end{cases}$$

$$\mathcal{M}^1_\rho(v,p_1@p_2) \coloneqq \begin{cases} \mathcal{M}^1_\rho(v_1,p_1) \cup \mathcal{M}^1_\rho(v_2,p_2) & \text{if } v = v_1@v_2 \\ \text{fail} & \text{otherwise} \end{cases}$$

$$\mathcal{M}^1_\rho(v,\lambda x.p) \coloneqq \begin{cases} \mathcal{M}^1_{\rho\cup[(x/a)]}(v_1,p) & \text{if } v = \lambda a.v_1 \\ \text{fail} & \text{otherwise} \end{cases}$$

$$\mathcal{M}^1_\rho(v,\$f(\vec{x})) \coloneqq \begin{cases} [f \mapsto \lambda\vec{x}.\langle v \downarrow \rho\rangle] & \text{if } (\mathrm{FV}(v) \cap \mathrm{Dom}(\rho))\,\rho \subseteq \{\vec{x}\} \\ \text{fail} & \text{otherwise} \end{cases}$$

Fig. 2 Semantics of Pattern Matching

variables in v, to obtain a set of pattern variables, and the condition constrains that the resulting set must be a subset of $\{x_1, \cdots, x_n\}$. The side condition for the last clause in Fig. 2 is slightly more general than this in that we take an intersection with $\mathrm{Dom}(\rho)$, the domain of ρ, since we want to allow a level-1 variable may appear in v if it is bound outside of the current pattern matching. This is necessary for the function `opt` in Sect. 1, which needs pattern matching for an open code, or pattern matching under a level-1 binder.

To compute the result of this pattern matching, we define $v \downarrow \rho$ as follows:

$$a \downarrow \rho \coloneqq \begin{cases} \sim x & \text{if } (x/a) \in \rho \\ a & \text{otherwise} \end{cases}$$

$$c \downarrow \rho \coloneqq c$$

$$(\lambda a.v_1) \downarrow \rho \coloneqq \lambda a.(v_1 \downarrow \rho) \;\text{ if } (x/a) \notin \rho$$

$$v_1@v_2 \downarrow \rho \coloneqq (v_1 \downarrow \rho)@(v_2 \downarrow \rho)$$

$$(v_1,v_2) \downarrow \rho \coloneqq (v_1 \downarrow \rho,\; v_2 \downarrow \rho)$$

$$(\texttt{let}\ a = v_1\ \texttt{in}\ v_2) \downarrow \rho \coloneqq \texttt{let}\ a = (v_1 \downarrow \rho)\ \texttt{in}\ (v_2 \downarrow \rho) \;\text{ if } (x/a) \notin \rho$$

The only interesting case for $v \downarrow \rho$ is the clause when v is a variable a. If $a \in \mathrm{Dom}(\rho)$, namely, a is bound within the current pattern matching, then we should replace it by $\sim x$ where $x = a\,\rho$, since a code value $\langle x_i\rangle$ will be substituted for this x. Otherwise, a is bound outside of the current pattern matching (which means that we are manipulating an open code), and we keep a intact. Other cases are homomorphic and their explanations are omitted.

To clarify the operational semantics of pattern matching presented in Fig. 2, we present examples where a code expression is matched against a pattern involving a higher-order pattern variable. These examples demonstrate how the pattern-matching function operates according to the rules, particularly focusing on the side condition for higher-order pattern matching, which ensures type safety and prevents scope extrusion.

Consider the following value and pattern to match:

$$\begin{aligned} \text{Value:} \quad & \langle \lambda a_1.\lambda a_2.(a_1,a_3)\rangle \\ \text{Pattern:} \quad & \langle \lambda x_1.\lambda x_2.\$f(x_1)\rangle \end{aligned}$$

To perform pattern matching, we apply the matching function at the level 0 with $\rho = [\,]$:

$$\begin{aligned} &\mathcal{M}^0_{[\,]}(\langle \lambda a_1.\lambda a_2.(a_1,a_3)\rangle, \langle \lambda x_1.\lambda x_2.\$f(x_1)\rangle) \\ &= \mathcal{M}^1_{[\,]}(\lambda a_1.\lambda a_2.(a_1,a_3), \lambda x_1.\lambda x_2.\$f(x_1)) \\ &= \mathcal{M}^1_{[(x_1/a_1)]}(\lambda a_2.(a_1,a_3), \lambda x_2.\$f(x_1)) \\ &= \mathcal{M}^1_{[(x_1/a_1),(x_2/a_2)]}((a_1,a_3), \$f(x_1)) \end{aligned}$$

To continue the process, we need to check the side condition for higher-order patterns $(\mathrm{FV}(v) \cap \mathrm{Dom}(\rho))\,\rho \subseteq \{\vec{x}\}$ where $v = (a_1,a_3)$, $\rho = [(x_1/a_1),(x_2/a_2)]$, and $\vec{x} = x_1$. Since $\mathrm{FV}(v) = \{a_1,a_3\}$ and $\mathrm{Dom}(\rho) = \{a_1,a_2\}$ hold, we can conclude that the side condition is satisfied, and pattern match succeeds. While the value contains an irrelevant variable a_3, we do not have to worry about it.

To get the result of pattern matching, we compute as follows:

$$(a_1, a_3) \downarrow \rho = (a_1 \downarrow \rho, a_3 \downarrow \rho) = (\sim x_1, a_3)$$

Hence, the final result of this pattern matching is:

$$[f \mapsto \lambda x_1.\langle(\sim x_1, a_3)\rangle]$$

The result shows that the higher-order pattern variable f gets bound to a level-0 function which receives a code and returns a code, thus higher-order.

Let us consider another example by taking $\langle\lambda a_1.\lambda a_2.(a_1, a_2)\rangle$ as a value and the same pattern as before. We perform pattern matching as:

$$\begin{aligned}&\mathcal{M}^0_{[\,]}(\langle\lambda a_1.\lambda a_2.(a_1, a_2)\rangle, \langle\lambda x_1.\lambda x_2.\$f(x_1)\rangle)\\ &= \mathcal{M}^1_\rho((a_1, a_2), \$f(x_1))\end{aligned}$$

where $\rho = [(x_1/a_1), (x_2/a_2)]$. Then, we have $(\mathrm{FV}((a_1, a_2)) \cap \mathrm{Dom}(\rho))\,\rho = \{x_1, x_2\}$ while $\{\vec{x}\} = \{x_1\}$, and this pattern matching fails.

These examples demonstrate the importance of the side condition for higher-order pattern matching in maintaining type safety and ensuring that no bound variables escape their intended scope. If the side condition were incorrectly modified, it could lead to improperly scoped variables, resulting in unsafe or ill-formed generated code.

Operational semantics of Mayuzin is given by **Fig. A.1** in the appendix.

3. Type System

This section presents a type system for Mayuzin, which is based on the type system for implicit classifiers by Calcagno, Moggi, and Taha [4].

The calculus for implicit classifiers was derived by restricting the one for environment classifiers by Taha and Nielsen [15]. Whereas both systems allow open-code manipulation and run-time metaprogramming, and enjoy type soundness, the latter is too expressive to enjoy the principal type property, leading to the lack of type inference. The former was designed to recover the property and type inference.

3.1 Environment Classifiers

An environment classifier (a classifier for short) γ is an abstraction of a typing environment $x_1 : t_1,\ x_2 : t_2, \cdots$. It is used to control the set of free variables in a code. A code $\langle 3 + x\rangle$ has the type $\langle\texttt{int}\rangle^\gamma$ where we assume that γ corresponds to the typing environment $x : \texttt{int}$. If a code has type $\langle t\rangle^\gamma$ for any γ, the code may be executed by the run primitive, as this indicates that the typing environment does not contain any free variables, implying that the code is closed.

In multi-stage programming languages, a stage is associated with a sequence of classifiers such as $\gamma_1\gamma_2\gamma_3$. The present stage is associated with an empty sequence, denoted as $\bullet$ in this paper. The length of the sequence indicates the level of the stage; for example, the stage $\gamma_1\gamma_2\gamma_3$ is at level 3. We use the notation $|S|$ to represent the level of the stage S, for instance, $|\bullet| = 0$ and $|\gamma_1\gamma_2\gamma_3| = 3$. The code $\langle\texttt{if}\ x\ \texttt{then}\ \langle y + 3\rangle\ \texttt{else}\ \langle z\rangle\rangle$ can have the type $\langle\langle\texttt{int}\rangle^{\gamma_2}\rangle^{\gamma_1}$ where γ_1 is associated with $x : \texttt{bool}$, and γ_2 is associated with $y : \texttt{int},\ z : \texttt{int}$.

Based on these ideas, Taha and Nielsen designed $\lambda\alpha$, a type-safe calculus that allows open-code manipulation and runtime metaprogramming. Meanwhile, Calcagno et al. designed λi, a type-safe calculus that allows let-polymorphism and type inference. A variant of the latter has become the foundational type system for the earlier version of MetaOCaml.

3.2 Type System of Mayuzin

This section presents a type system for Mayuzin. In this paper, we restrict ourselves to two levels, namely, a stage is either the present stage $\bullet$, or a level-1 stage γ.

The syntax of a stage (ranged over by S), a monotype (ranged over by t), and a polytype (ranged over by τ) are defined as follows:

$$\begin{aligned}S &::= \bullet \mid \gamma\\ t &::= \texttt{int} \mid t \to t \mid t \times t \mid \langle t\rangle^\gamma \mid \alpha\\ \tau &::= t \mid \forall\alpha.\tau\end{aligned}$$

Following ML-family languages, we distinguish monotypes (monomorphic types) from polytypes (polymorphic types). Monotypes are those in simply typed lambda calculus, the code type with a classifier $\langle t\rangle^\tau$, or a type variable α. Polytypes may have universal quantifiers for type variables at the outermost position. Most of the types are standard, except for the code type $\langle t\rangle^\gamma$, which is the type of code fragments whose content is of type t.

The set of free type variables (and free classifiers, resp.) in X is denoted by $\mathrm{FTV}(X)$ (and $\mathrm{FC}(X)$, resp.) where $\forall$ binds a type variable, and there are no binders for classifiers.

Fig. 3 presents the typing rules for terms. A judgment for terms takes the form $\Gamma \vdash^S e : t$, where Γ is a finite sequence of variable-type pairs in the form $(a : \tau)^{S'}$, which means that the variable a has the polytype τ at the stage S'. The judgment $\Gamma \vdash^S e : t$ means that, the term e is of type t under Γ

$$\frac{}{\Gamma \ \vdash^S c : \mathtt{type}(c)}\ (\text{T-Const}) \qquad \frac{(a:\tau)^S \in \Gamma \qquad t \prec \tau}{\Gamma \vdash^S a : t}\ (\text{T-Var}) \qquad \frac{\Gamma, (a:t_1)^S \vdash^S e : t_2}{\Gamma \vdash^S \lambda a.e : t_1 \to t_2}\ (\text{T-Fun}) \qquad \frac{\Gamma \vdash^S e_1 : t_2 \to t_1 \qquad \Gamma \vdash^S e_2 : t_2}{\Gamma \vdash^S e_1 @ e_2 : t_1}\ (\text{T-App})$$

$$\frac{\Gamma \vdash^S e_1 : t_1 \qquad \Gamma \vdash^S e_2 : t_2}{\Gamma \vdash^S (e_1, e_2) : t_1 \times t_2}\ (\text{T-Pair}) \qquad \frac{\Gamma \vdash^S e_1 : t_1 \qquad \Gamma, (a : \forall \vec{\alpha}.t_1)^S \vdash^S e_2 : t_2 \qquad \{\vec{\alpha}\} \subseteq \mathrm{FTV}(t_1) - \mathrm{FTV}(\Gamma)}{\Gamma \vdash^S \mathtt{let}\ a = e_1\ \mathtt{in}\ e_2 : t_2}\ (\text{T-Let})$$

$$\frac{\Gamma \ \vdash^\gamma e : t}{\Gamma \vdash^\bullet \langle e \rangle : \langle t \rangle^\gamma}\ (\text{T-Code}) \qquad \frac{\Gamma \vdash^\bullet e : \langle t \rangle^\gamma}{\Gamma \vdash^\gamma \sim e : t}\ (\text{T-Splice}) \qquad \frac{\Gamma \ \vdash^\bullet e : \langle t \rangle^\gamma \qquad \gamma \notin \mathrm{FC}(\Gamma, t)}{\Gamma \vdash^\bullet \mathtt{run}\ e : t}\ (\text{T-Run})$$

$$\frac{\Gamma \vdash^\bullet e : t_1 \qquad \forall j \in [1,n]\ (\Gamma; \emptyset \vdash^\bullet p_j : t_1 \dashv \Gamma_j \ \wedge \ \Gamma, \Gamma_j \vdash^\bullet e_j : t_2\ ) \qquad \Gamma \vdash^\bullet e_{n+1} : t_2}{\Gamma \vdash^\bullet \mathtt{match}\ e\ \mathtt{with}\ (p_1 \to e_1 \mid \cdots \mid p_n \to e_n \mid \_ \to e_{n+1}) : t_2}\ (\text{T-Match})$$

Fig. 3 Typing Rules for Expressions

$$\frac{}{\Gamma_1; \Gamma_2 \vdash^\bullet x : t \dashv (x:t)^\bullet}\ (\text{T-Pat-Var-0}) \qquad \frac{}{\Gamma_1; \Gamma_2 \ \vdash^S c : \mathtt{type}(c) \dashv \emptyset}\ (\text{T-Pat-Const}) \qquad \frac{\Gamma_1; \Gamma_2 \vdash^S p_1 : t_1 \dashv \Gamma_3 \qquad \Gamma_1; \Gamma_2 \vdash^S p_2 : t_2 \dashv \Gamma_4}{\Gamma_1; \Gamma_2 \vdash^S (p_1, p_2) : t_1 \times t_2 \dashv \Gamma_3, \Gamma_4}\ (\text{T-Pat-Pair})$$

$$\frac{(x:t)^\gamma \in \Gamma_1 \cup \Gamma_2}{\Gamma_1; \Gamma_2 \vdash^\gamma x : t \dashv \emptyset}\ (\text{T-Pat-Var-1}) \qquad \frac{\Gamma_1; \Gamma_2, (x:t_1)^\gamma \vdash^\gamma p : t_2 \dashv \Gamma_3}{\Gamma_1; \Gamma_2 \vdash^\gamma \lambda x.p : t_1 \to t_2 \dashv \Gamma_3}\ (\text{T-Pat-Fun}) \qquad \frac{\Gamma_1; \Gamma_2 \vdash^\gamma p_1 : t_2 \to t_1 \dashv \Gamma_3 \qquad \Gamma_1; \Gamma_2 \vdash^\gamma p_2 : t_2 \dashv \Gamma_4}{\Gamma_1; \Gamma_2 \vdash^\gamma p_1 @ p_2 : t_1 \dashv \Gamma_3, \Gamma_4}\ (\text{T-Pat-App})$$

$$\frac{\Gamma_1; \Gamma_2 \ \vdash^\gamma p : t \dashv \Gamma_3}{\Gamma_1; \Gamma_2 \vdash^\bullet \langle p \rangle : \langle t \rangle^\gamma \dashv \Gamma_3}\ (\text{T-Pat-Code}) \qquad \frac{(x_j : t_j)^\gamma \in \Gamma_2 \ \text{ for all } \ j \in [1,n]}{\Gamma_1; \Gamma_2 \vdash^\gamma \$f(x_1, \cdots, x_n) : t \dashv (f : \langle t_1 \rangle^\gamma \to ... \to \langle t_n \rangle^\gamma \to \langle t \rangle^\gamma)^\bullet}\ (\text{T-Pat-Hop})$$

Fig. 4 Typing Rules for Patterns

at the stage S.

The first six rules in Fig. 3 are standard except that each judgment as well as the variable-type pair in typing contexts is annotated with a stage. The rules T-Var and T-Let together allow a type variable to be polymorphic. The side condition $t \prec \tau$ in T-Var means that the monotype t is an instance of the polytype τ, where the 'instance-of' relation is defined as usual.

The next three rules, T-Code, T-Splice, and T-Run are the standard rules in the calculus for environment classifiers. The side condition $\gamma \notin \mathrm{FC}(\Gamma, t)$ is critically important to ensure the absence of scope extrusion. See the literature for a detailed discussion [15].

Finally, the rule T-Match is worth discussing, as it performs not only pattern matching for ordinary values, but also pattern matching for code values. The second assumption in this rule means that, for each case $p_j \to e_j$, the judgments $\Gamma; \emptyset \vdash^\bullet p_j : t_j \dashv \Gamma_j$ and $\Gamma, \Gamma_j \vdash^\bullet e_j : t_j$ must hold for some Γ_j. Here, Γ_j denotes a typing context, which intuitively means the set of free variables in p_j, and we need it when we type e_j. The default clause $\_ \to e_{n+1}$ binds no pattern variables.

Fig. 4 defines the typing rules for patterns. A judgment for a pattern p takes the form $\Gamma_1; \Gamma_2 \vdash^S p : t \dashv \Gamma_3$, which means that p has type t at the stage S under the typing contexts Γ_1 and Γ_2. Here, Γ_1 is the context for the variables outside of the current pattern matching, whereas Γ_2 is the one for local variables within the current pattern matching. Γ_3 is the 'output' of this pattern, that means p generates bindings in Γ_3. In several typing rules, we use the notation Γ_1, Γ_2, which is defined only when the contexts Γ_1 and Γ_2 share no variables.

Let us elaborate on a few interesting rules in Fig. 4.

The rule (T-Pat-Var-0) is applicable for level-0 variables

(variables at the stage $\bullet$). Since a level-0 variable cannot be bound in a pattern, it must be a free variable. Therefore, we output its information $(x : t)^{\bullet}$.

The rule (T-Pat-Var-1) is applicable for level-1 variables (variables at the stage γ). Since level-1 variables must not be free, they must be present in the current typing context Γ_1 or Γ_2, and we output no information about it.

The rule (T-Pat-HOP) checks the pattern with higher-order pattern variables. It states that the pattern $\$f(x_1, \cdots, x_n)$ is well-typed if and only if all the variables $x_1, \cdots, x_n$ are bound in the current typing context Γ_2 for locally bound variables. Additionally, the higher-order pattern variable f is a free variable in this pattern, and its binding along with its type is output.

The remaining rules are standard, and their explanation is omitted.

3.3 Example of Type Derivation

Fig. 5 demonstrates a type derivation for the function `opt` in Sect. 1 (we elided a few cases since they are similar), which is designed to perform code optimizations by recognizing and reducing specific patterns involving arithmetic expressions. The purpose of this detailed explanation is to break down the derivation step-by-step and to illustrate how our type system ensures that the function `opt` is type-safe in the context of Mayuzin.

Here, we assume that Mayuzin is extended suitably by `fix` for general recursion, integers, and operators such as multiplication. We also assume that the overall function is used to manipulate open code, namely, it should be typed under an arbitrary typing context Γ_0.

The derivation steps in Fig. 5 are mostly standard except for the step deriving the type for the `match` expression. This step requires verifying that the expression being pattern-matched (z in this example) is correctly typed, as well as ensuring that all cases of the pattern matching are properly typed. We will show type derivations for a normal case (written by $\square \wedge \square'$) and the default case.

The type derivations for $\square$ and $\square'$ are separately given in **Fig. 6**. The type derivation marked with $\square$ provides a type for the pattern $\langle \$x * 1 \rangle$ where x is a higher-order pattern variable.[*4] After a few steps of derivations, it produces a typing environment $(x : t_1)^{\bullet}$ for $t_1 = \langle \texttt{int} \rangle^{\gamma}$, which holds the typing for pattern variables. This typing environment may be used to type the body of the case, $f@x$ for this case. The type derivation marked with $\square'$ gives a typing for this expression under the typing context extended with $(x : t_1)^{\bullet}$.

Note that the body of each case may dereference to a variable in Γ_0, which is bound outside of this function, revealing the fact that our language allows analytical metaprogramming for open code freely and soundly.

4. Properties

In this section, we will consider theoretical properties of constant-free fragments of Mayuzin. In particular, we prove type soundness , which comprises two key properties: progress and subject reduction. We also present the key lemmas necessary to prove these properties, along with a proof sketch for important cases.

Our initial target is the progress property, which means that a well-typed, closed, level-0 expression does not get stuck immediately.

First, we need the Canonical-Form lemma, which can be proved easily.

Lemma 1.

- *If* $\Gamma \vdash^{\bullet} v^0 : t_1 \to t_2$, *then* $v^0 = \lambda a.e$ *for some* a *and* e.
- *If* $\Gamma \vdash^{\bullet} v^0 : t_1 \times t_2$, *then* $v^0 = (v_1^0, v_2^0)$ *for some* v_1^0 *and* v_2^0.
- *If* $\Gamma \vdash^{\bullet} v^0 : \langle t \rangle^{\gamma}$, *then* $v^0 = \langle v^1 \rangle$ *for some* v^1.

To prove the progress property for closed expressions by induction, we need to generalize the property to allow semi-closed expressions, which have no free level-0 variables, but may have level-1 variables.

Let us define a level ≥ 1 typing context[*5] as follows:

$$\Gamma^{\geq 1} ::= \emptyset \mid \Gamma^{\geq 1}, (a : \tau)^{\gamma}$$

We can prove the following lemma. Note that a match expression in Mayuzin must have a default case, so it does not get stuck.

Lemma 2. *Let* $i = |S|$. *If* $\Gamma^{\geq 1} \vdash^{S} e : t$, *then* e *is a level-*i *value, or there exists an* e' *such that* $e \longrightarrow^{i} e'$.

The lemma is proved by straightforward induction on e.

As a corollary, we obtain the progress property.

Theorem 1 (Progress)**.** *If* $\emptyset \vdash^{\bullet} e : t$, *then* e *is a level-*0 *value, or there exists an* e' *such that* $e \longrightarrow^{0} e'$.

Proof. We only have to choose $S = \bullet$ and $\Gamma^{\geq 1} = \emptyset$ in the above lemma. □

[*4] Although the pattern variable x is not really higher order, we call it a higher-order pattern variable for uniformity.

[*5] The notation ≥ 1 is equivalent to $= 1$ in this paper, but we can prove the property for a more general multi-stage calculus.

$$\dfrac{\dfrac{\dfrac{\overline{\Gamma_0, (f : t_1 \to t_1)^\bullet, (z : t_1)^\bullet \vdash^\bullet z : t_1} \quad \cdots \quad \Box \wedge \Box' \quad \cdots \quad \overline{\Gamma_0, (f : t_1 \to t_1)^\bullet, (z : t_1)^\bullet \vdash^\bullet z : t_1}}{\Gamma_0, (f : t_1 \to t_1)^\bullet, (z : t_1)^\bullet \vdash^\bullet \texttt{match}\ z\ \texttt{with}\ (\langle \$x * 0\rangle \to \langle 0\rangle \mid \langle \$x * 1\rangle \to f@x \mid \langle \$x * \$y\rangle \to \langle \sim (f@x) * \sim (f@y)\rangle \mid \_ \to z) : t_1}}{\Gamma_0, (f : t_1 \to t_1)^\bullet \vdash^\bullet \lambda z.\texttt{match}\ z\ \texttt{with}\ (\langle \$x * 0\rangle \to \langle 0\rangle \mid \langle \$x * 1\rangle \to f@x \mid \langle \$x * \$y\rangle \to \langle \sim (f@x) * \sim (f@y)\rangle \mid \_ \to z) : t_1 \to t_1}}{\Gamma_0 \vdash^\bullet \lambda f.\lambda z.\texttt{match}\ z\ \texttt{with}\ (\langle \$x * 0\rangle \to \langle 0\rangle \mid \langle \$x * 1\rangle \to f@x \mid \langle \$x * \$y\rangle \to \langle \sim (f@x) * \sim (f@y)\rangle \mid \_ \to z) : (t_1 \to t_1) \to (t_1 \to t_1)}}{\Gamma_0 \vdash^\bullet \mathit{fix}\ (\lambda f.\lambda z.\texttt{match}\ z\ \texttt{with}\ (\langle \$x * 0\rangle \to \langle 0\rangle \mid \langle \$x * 1\rangle \to f@x \mid \langle \$x * \$y\rangle \to \langle \sim (f@x) * \sim (f@y)\rangle \mid \_ \to z) : t_1 \to t_1)}$$

Fig. 5 Example of Type Derivation where $t_1 = \langle \texttt{int}\rangle^\gamma$ and Γ_0 is arbitrary

$$\Box = \dfrac{\dfrac{\dfrac{\overline{\Gamma_1; \emptyset \vdash^\gamma \$x : \texttt{int} \dashv (x : t_1)^\bullet} \quad \overline{\Gamma_1; \emptyset \vdash^\gamma 1 : \texttt{int} \dashv \emptyset}}{\Gamma_1; \emptyset \vdash^\gamma \$x * 1 : \texttt{int} \dashv (x : t_1)^\bullet}}{\Gamma_1; \emptyset \vdash^\bullet \langle \$x * 1\rangle : t_1 \dashv (x : t_1)^\bullet}}{}$$

$$\Box' = \dfrac{\overline{\Gamma_1, (x : t_1)^\bullet \vdash^\bullet f : t_1 \to t_1} \quad \overline{\Gamma_1, (x : t_1)^\bullet \vdash^\bullet x : t_1}}{\Gamma_1, (x : t_1)^\bullet \vdash^\bullet f@x : t_1}$$

Fig. 6 Deriving $\Box$ and $\Box'$ where $t_1 = \langle \texttt{int}\rangle^\gamma$ and $\Gamma_1 = \Gamma_0, (f : t_1 \to t_1)^\bullet, (z : t_1)^\bullet$

Our next goal is to prove the subject-reduction property of our type system, which states that the typing of an expression is preserved throughout evaluation. The property is stronger than the one for ordinary (unstaged) typed calculi, as it statically ensures not only the type soundness of program generators (level-0 expressions), but also the one for all generated programs (level-1 expressions). In particular, it implies that generated programs are well-formed, well-scoped and well-typed, namely, there will be no compilation errors for generated programs.

As is usual, the key lemma to prove the subject-reduction property is the preservation of typability under substitution (commonly phrased Substitution Lemma). Since Mayuzin supports let-polymorphism, we need Substitution Lemma that takes into account polymorphism.

Lemma 3. *Suppose* $\{\vec{\alpha}\} \subseteq \mathit{FTV}(t_1) - \mathit{FTV}(\Gamma)$. *If*

$$\Gamma \vdash^\bullet v : t_1 \quad \textit{and}$$
$$\Gamma, (a : \forall \vec{\alpha}.t_1)^\bullet \vdash^\bullet e : t_2$$

are derivable, then $\Gamma \vdash^\bullet e\ [a \mapsto v] : t_2$ *is also derivable.*

Proof. The lemma is proved by induction on the type derivation of e (the second assumption in the lemma). □

The following lemma is necessary to prove the case for the evaluation of `run` (the (E-Run) reduction in Sect. A.1). It says that if the classifier γ is not used in the derivation of a level-1 value v, we can regard v as a level-0 expression.

Lemma 4. *If* $\Gamma \vdash^\gamma v^1 : t$ *is derivable and* $\gamma \notin \mathit{FC}(\Gamma, t)$, *then* $\Gamma \vdash^\bullet v^1 : t$ *is also derivable.*

Proof. This lemma is proved by straightforward induction on the type derivation. □

The next technical lemma states a property about the renaming ρ.

Lemma 5. *Let* $\rho = (x_1/a_1, \cdots, x_k/a_k)$, $I \subseteq \{1, \cdots, k\}$, *and* $\Gamma_a = (a_1 : t_1)^\gamma, \cdots, (a_k : t_k)^\gamma$. *Suppose* Γ_x *consists of* $(x_i : \langle t_i\rangle^\gamma)^\bullet$ *for all* $i \in I$. *If*

$$\Gamma_1, \Gamma_a, \Gamma_2 \vdash^\gamma v : t \quad \textit{is derivable, and}$$
$$\mathit{FV}(v) \cap \{a_1, \cdots, a_k\} \subseteq \{a_i \mid i \in I\}$$

then $\Gamma_1, \Gamma_x, \Gamma_2 \vdash^\gamma v \downarrow \rho : t$ *is derivable.*

Note that we change not only the variable a_i to x_i, but also its type t_i to $\langle t_i\rangle^\gamma$, reflecting our semantics for higher-order pattern variables in Fig. 2.

Proof. The lemma is proved by induction on the structure of v. The only non-trivial case is when v is a variable a_j for some $j \le k$. We have $t = t_j$ and $v \downarrow \rho = \sim x_j$. By the condition on $\mathrm{FV}(v)$, we have $j \in I$. Hence, $(x_j : \langle t_j\rangle^\gamma)^\bullet \in \Gamma_x$, and we get the conclusion of this lemma. □

The next lemma takes care of successful pattern matching:

$$\texttt{match}\ v\ \texttt{with}\ (\cdots \mid p \to e \mid \cdots) \longrightarrow^0 e\ \theta$$

where $\theta = \mathcal{M}^0_{[\,]}(v, p) \neq \mathrm{fail}$.

Lemma 6. *Let* S *be a stage and* $i = |S|$. *Suppose* $\rho = (x_1/a_1, \cdots, x_k/a_k)$, *and let* $\Gamma_x = (x_1 : t_1)^\gamma, \cdots, (x_k : t_k)^\gamma$ *and* $\Gamma_a = (a_1 : t_1)^\gamma, \cdots, (a_k : t_k)^\gamma$. *Let* $\theta = \mathcal{M}^i_\rho(v, p) \neq \mathit{fail}$. *If*

$$\Gamma_1, \Gamma_a \vdash^S v : t,$$
$$\Gamma_1; \Gamma_x \vdash^S p : t \dashv \Gamma_2, \quad \textit{and}$$
$$\Gamma_1, \Gamma_2 \vdash^\bullet e : t_0$$

are derivable, then $\Gamma_1 \vdash^\bullet e\ \theta : t_0$ *is also derivable.*

Proof. This lemma is proved by induction on the type derivation of $\Gamma_1; \Gamma_x \vdash^S p : t \dashv \Gamma_2$.

We show the most interesting cases only.

(Case: $S = \gamma$ and $p = \lambda x.p_1$) Since pattern matching succeeds, v must have the form $\lambda a.v_1$. By the assumptions and inversion of typing rules, $\Gamma_1, \Gamma_a, (a : t')^\gamma \vdash^\gamma v_1 : t''$ and $\Gamma_1; \Gamma_x, (x : t')^\gamma \vdash^\gamma p_1 : t'' \dashv \Gamma_2$ are derivable for some t' and t''. We also have $\theta = \mathcal{M}^1_{\rho \cup [(x/a)]}(v_1, p_1)$. By the induction hypothesis, $\Gamma_1 \vdash^\bullet e\ \theta : t_0$ is derivable.

(Case: $S = \gamma$ and $p = \$f(\vec{y})$) Let n be $|\vec{y}|$. By the second derivation in the assumptions of the lemma and the typing rule (T-Pat-Hop), each y_j is x_{i_j} for some $i_j \leq k$ such that $(y_j : t_{i_j})^\gamma \in \Gamma_x$. Then, we have $\Gamma_2 = (f : t')^\bullet$ where $t' = \langle t_{i_1} \rangle^\gamma \to \cdots \to \langle t_{i_n} \rangle^\gamma \to \langle t \rangle^\gamma$.

Since pattern matching succeeds ($\theta \neq$ fail), we have $(\mathrm{FV}(v) \cap \{a_1, \cdots, a_k\})\,\rho \subseteq \{y_1, \cdots, y_n\}$, which is $\{x_{i_1}, \cdots, x_{i_n}\}$. Let I be $\{i_1, \cdots, i_n\}$. Applying Lemma 5 to $\Gamma_1, \Gamma_a \vdash^\gamma v : t$, we obtain

$$\Gamma_1, (x_{i_1} : \langle t_{i_1} \rangle^\gamma)^\bullet, \cdots, (x_{i_n} : \langle t_{i_n} \rangle^\gamma)^\bullet \vdash^\gamma v \downarrow \rho : t.$$

Since x_{i_j} is y_j, we can derive $\Gamma_1 \vdash^\bullet \lambda\vec{y}.\langle v \downarrow \rho \rangle : t'$. By applying Lemma 3 to it and $\Gamma_1, (f : t')^\bullet \vdash^\bullet e : t_0$, we get $\Gamma_1 \vdash^\bullet e\ [f \mapsto \lambda\vec{y}.\langle v \downarrow \rho \rangle] : t_0$. Since $\theta = [f \mapsto \lambda\vec{y}.\langle v \downarrow \rho \rangle]$, we are done. □

Theorem 2 (Type Preservation). *Let $i = |S|$. If $\Gamma \vdash^S e : t$ and $e \longrightarrow^i e'$, then $\Gamma \vdash^S e' : t$.*

Proof. The theorem is proved by induction on $e \longrightarrow^i e'$.

The evaluation rule for β-reduction ($(\lambda a.e)@v$) and let-reduction (for `let` $a = v$ `in` e) are proved by Lemma 3.

The rule for run (`run` $\langle v \rangle$) is proved by Lemma 4.

The rule for successful pattern matching is proved by Lemma 6.

The proof for the remaining cases is straightforward. □

Theorems 1 and 2 together establish type soundness of Mayuzin, that allows pattern-matching against code patterns, open-code manipulation, run-time code generation, and let-polymorphism.

5. Implementation

The primary motivation for this work was to provide a type-safe Multi-Stage Programming (MSP) language that supports both runtime generative and analytical approaches. To assess the practical utility of the approach, we implemented Mayuzin from scratch with the type system presented in the previous sections.

At present, our implementation is prototypical in the sense that it consists of an interpreter and a naïve type checker for the type system, where we need to annotate all variables with their types and stages as environment classifiers. We added integers and their operations, general recursion, and so on, to make our system practical and expressive for the examples in this paper. The limitations of our type system is inherited from the present paper, for instance, the restriction of two stages and no classifier polymorphism, yet, we can run the pattern-matching example in Sect. 1.

To make Mayuzin a practically useful language, we need to revise the implementation significantly. For instance, pattern matching in OCaml can be efficiently implemented [1], and extending their work to cover code patterns is interesting future work.

6. Related Work

In this section, we compare our work with the most closely related work.

Multi-Stage Programming (MSP) is a concept that underlies Mayuzin, providing a statically-scoped, type-safe mechanism for staged computation. MSP was first popularized by languages such as MetaML and MetaOCaml, which allowed developers to specify different stages of code generation and evaluation [3], [13]. These systems helped narrow the gap in generative programming, enabling tasks such as code optimizations and Domain-Specific Language (DSL) implementations more effectively. Similar to MetaOCaml, Mayuzin facilitates type-safe staging and emphasizes runtime code generation and simplification, optimizing generated code through an efficient staged evaluation mechanism.

Lisp and **Scheme** have long served as a foundational example of treating code as data through its uniform representation of programs as lists, or more generally, S-expressions. Techniques such as quotation (') and quasiquotation (`) allow seamless transitions between code fragments and data, making Lisp powerful for metaprogramming tasks. The notion of quasiquotation allows one to mix data and evaluable code fragments within the same expression, as seen in `` `(1 2 ,(+ 3 4)) ``, where unquote (`,`) facilitates the evaluation of a part of the quoted list. This approach to code manipulation provides significant flexibility in generating and transforming code [11].

Scala 3, inspired by MetaML and MacroML, has extended the capabilities of MSP by integrating generative and analytical macros into a statically typed system [14]. Unlike purely generative approaches, Scala 3 supports compile-time transformations that allow developers to analyze and optimize code during compilation. Mayuzin similarly aims to provide

both generative and analytical capabilities but focuses primarily on runtime code generation. This distinction makes Mayuzin particularly suitable for applications that require runtime adaptability, whereas Scala 3 emphasizes compile-time guarantees.

Mœbius, an alternative approach to MetaML, has extended Contextual Modal Type Theory (CMTT) by multi-stages, System-F style polymorphism, pattern patching, and other features [7]. CMTT provides a fine-grain representation for typing context, which allows us to represent the dependency between variables, including type variables, that is essential to handle impredicative polymorphism properly. Its pattern matching is also very expressive, for instance, it allows different type-instantiation for each case in pattern matching. A price for the expressiveness of Mœbius is its complexity; The calculus needs an involved definition for typing rules and simultaneous substitutions. Also, the manipulation of types and typing contexts is involved, which makes one wonder whether it can be a basis for a practical programming language. Our approach is opposite; we stick to a simple setting based on environment classifiers, and live with MetaML-like languages, for which a complete type inference algorithm in the presence of let polymorphism is strongly desired. Although it is left for future work to provide a type inference algorithm for Mayuzin, we believe that this paper has made a solid first step to design a suitable extension of MetaML with pattern matching for generated code.

MacoCaml, an extension of OCaml, focuses on compile-time code generation using macros combined with phase separation and staging [17]. It introduces composable and compilable macros as compile-time bindings, and provides a framework that integrates these macros with OCaml's type system, enabling a safe and structured form of macro staging. MacoCaml supports both compile-time bindings for macro definitions and runtime evaluations, effectively interleaving typing and code generation. The emphasis on phase separation allows for precise control over different stages of evaluation, offering benefits for compile-time efficiency and correctness.

Apart from extending MSP languages to allow analytical metaprogramming, a number of researchers have devised a way to analyze and transform programs either after program generation, or simultaneously during program generation. Typically, they design Domain-Specific Languages for this purpose to exploit domain-specific knowledge for custom optimizations. Among many studies, Kiselyov developed the typed tagless-final encoding for an embedded DSL and showed that various low-level optimizations for high-performance computation can actually be implemented in this DSL [10]. While powerful and general, these approaches have an obvious drawback that the expressiveness of the DSL is limited by the host language and encoding, and it is not easy to cover polymorphism and other sophisticated typing mechanisms. Our work provides strong evidence that combining generative and analytical metaprogramming in a single language can be achieved. Further investigation into the merits and defects of our approach is to be left for future work.

7. Conclusion

This paper has presented Mayuzin, a new multi-stage programming language that successfully integrates both generative and analytical metaprogramming capabilities while maintaining type safety. Our key contributions include demonstrating how a unified approach can combine runtime code generation and analysis, thereby addressing a notable gap in multi-stage programming theory.

We have shown the practical benefits of Mayuzin by integrating let polymorphism and environment classifiers within the multi-stage framework. This combination allows for sophisticated code transformations while ensuring the generated code maintains type safety. Through practical examples, we highlighted how Mayuzin facilitates code optimizations, effectively eliminating redundant computations, simplifying the code, and enhancing program efficiency.

The type safety proof for our system establishes the theoretical soundness of these mechanisms. Our formal treatment serves as a basis for further research, and our implementation will provide evidence of its practicality. This foundation sets the stage for future exploration into broader applications of multi-stage metaprogramming, particularly in domains requiring dynamic code analysis and optimization.

As future work, we plan to extend our language to more than two stages [15], cross-stage persistence (CSP) [5], classifier polymorphism [4], and computational effects, including control operators [9] and effect handlers [6]. We also hope to build a type-inference algorithm that would eliminate the burden of annotating programs with types and classifiers. These extensions will allow us to use Mayuzin for realistic applications.

Acknowledgments We would like to thank the anonymous reviewers and the participants of the IPSJ SIG-Programming meeting held on January 15–16, 2025 for their

constructive comments and suggestions. The authors are supported in part by JSPS Grant-in-Aid for Scientific Research (B) 23K24819. The first author is supported in part by the Urakami Scholarship Foundation.

References

[1] Augustsson, L.: Compiling pattern matching, *Functional Programming Languages and Computer Architecture* (Jouannaud, J.-P., ed.), Berlin, Heidelberg, Springer Berlin Heidelberg, pp. 368–381 (1985).

[2] Barendregt, H. P.: *The lambda calculus - its syntax and semantics*, Studies in logic and the foundations of mathematics, Vol. 103, North-Holland (1985).

[3] Calcagno, C., Moggi, E. and Sheard, T.: Closed types for a safe imperative MetaML, *J. Funct. Program.*, Vol. 13, No. 3, pp. 545–571 (online), DOI: 10.1017/S0956796802004598 (2003).

[4] Calcagno, C., Moggi, E. and Taha, W.: ML-Like Inference for Classifiers, *Programming Languages and Systems, 13th European Symposium on Programming, ESOP 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings* (Schmidt, D. A., ed.), Lecture Notes in Computer Science, Vol. 2986, Springer, pp. 79–93 (online), DOI: 10.1007/978-3-540-24725-8_7 (2004).

[5] Hanada, Y. and Igarashi, A.: On Cross-Stage Persistence in Multi-Stage Programming, *Functional and Logic Programming - 12th International Symposium, FLOPS 2014, Kanazawa, Japan, June 4-6, 2014. Proceedings* (Codish, M. and Sumii, E., eds.), Lecture Notes in Computer Science, Vol. 8475, Springer, pp. 103–118 (online), DOI: 10.1007/978-3-319-07151-0_7 (2014).

[6] Isoda, K., Yokoyama, A. and Kameyama, Y.: Type-Safe Code Generation with Algebraic Effects and Handlers, *Proceedings of the 23rd ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE 2024, Pasadena, CA, USA, October 22, 2024* (Chiba, S. and Thüm, T., eds.), ACM, pp. 53–65 (online), DOI: 10.1145/3689484.3690731 (2024).

[7] Jang, J., Gélineau, S., Monnier, S. and Pientka, B.: Mœbius: metaprogramming using contextual types: the stage where system F can pattern match on itself, *Proc. ACM Program. Lang.*, Vol. 6, No. POPL, pp. 1–27 (online), DOI: 10.1145/3498700 (2022).

[8] Jones, N. D., Gomard, C. K. and Sestoft, P.: *Partial evaluation and automatic program generation*, Prentice Hall (1993).

[9] Kameyama, Y., Kiselyov, O. and Shan, C.-c.: Shifting the Stage: Staging with Delimited Control, *Proceedings of the 2009 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, PEPM '09, New York, NY, USA, ACM, pp. 111–120 (online), DOI: 10.1145/1480945.1480962 (2009).

[10] Kiselyov, O.: Reconciling Abstraction with High Performance: A MetaOCaml approach, *Found. Trends Program. Lang.*, Vol. 5, No. 1, pp. 1–101 (online), DOI: 10.1561/2500000038 (2018).

[11] McCarthy, J.: Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I, *Commun. ACM*, Vol. 3, No. 4, pp. 184–195 (online), DOI: 10.1145/367177.367199 (1960).

[12] Nanevski, A., Pfenning, F. and Pientka, B.: Contextual modal type theory, *ACM Trans. Comput. Log.*, Vol. 9, No. 3, pp. 23:1–23:49 (online), DOI: 10.1145/1352582.1352591 (2008).

[13] Sheard, T., Benaissa, Z. and Pasalic, E.: DSL implementation using staging and monads, *Proceedings of the Second Conference on Domain-Specific Languages (DSL '99), Austin, Texas, USA, October 3-5, 1999* (Ball, T., ed.), ACM, pp. 81–94 (online), DOI: 10.1145/331960.331975 (1999).

[14] Stucki, N., Brachthäuser, J. I. and Odersky, M.: Multi-stage programming with generative and analytical macros, *GPCE '21: Concepts and Experiences, Chicago, IL, USA, October 17 - 18, 2021* (Tilevich, E. and Roover, C. D., eds.), ACM, pp. 110–122 (online), DOI: 10.1145/3486609.3487203 (2021).

[15] Taha, W. and Nielsen, M. F.: Environment classifiers, *Conference Record of POPL 2003: The 30th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, New Orleans, Louisisana, USA, January 15-17, 2003* (Aiken, A. and Morrisett, G., eds.), ACM, pp. 26–37 (online), DOI: 10.1145/604131.604134 (2003).

[16] Taha, W. and Sheard, T.: MetaML and multi-stage programming with explicit annotations, *Theor. Comput. Sci.*, Vol. 248, No. 1-2, pp. 211–242 (online), DOI: 10.1016/S0304-3975(00)00053-0 (2000).

[17] Xie, N., White, L., Nicole, O. and Yallop, J.: MacoCaml: Staging Composable and Compilable Macros, *Proc. ACM Program. Lang.*, Vol. 7, No. ICFP, pp. 604–648 (online), DOI: 10.1145/3607851 (2023).

Appendix

A.1 Operational Semantics

The operational semantics of Mayuzin is given as the standard call-by-value semantics for lambda calculus, extended with pattern matching for code explained in Sect. 2.2. Below, we give the semantics as the set of small-step evaluation rules where we assume that the evaluation rules for constants are separately given.

$$\frac{e_1 \longrightarrow^i e_1'}{e_1@e_2 \longrightarrow^i e_1'@e_2}\ \text{(E-App-1)} \qquad \frac{e_2 \longrightarrow^i e_2'}{v_1^i@e_2 \longrightarrow^i v_1^i@e_2'}\ \text{(E-App-2)}$$

$$(\lambda a.e_1)@v_2^0 \longrightarrow^0 e_1\ [a \mapsto v_2^0] \quad \text{(E-Beta)}$$

$$\frac{e \longrightarrow^i e' \qquad i > 0}{\lambda a.e \longrightarrow^i \lambda a.e'}\ \text{(E-Fun)} \qquad \frac{e \longrightarrow^{i+1} e'}{\langle e \rangle \longrightarrow^i \langle e' \rangle}\ \text{(E-Code)}$$

$$\frac{e \longrightarrow^{i-1} e' \qquad i > 0}{\sim e \longrightarrow^i \sim e'}\ \text{(E-Splice)} \qquad \sim \langle v^1 \rangle \longrightarrow^1 v^1\ \text{(E-Splice-Red)}$$

$$\frac{e_1 \longrightarrow^i e_1'}{(e_1, e_2) \longrightarrow^i (e_1', e_2)}\ \text{(E-Pair-1)} \qquad \frac{e_2 \longrightarrow^i e_2'}{(v_1^i, e_2) \longrightarrow^i (v_1^i, e_2')}\ \text{(E-Pair-2)}$$

$$\frac{e_1 \longrightarrow^i {e_1}'}{\texttt{let}\ a = e_1\ \texttt{in}\ e_2 \longrightarrow^i \texttt{let}\ a = {e_1}'\ \texttt{in}\ e_2}$$
(E-Let-1)

$$\texttt{let}\ a = v_1^0\ \texttt{in}\ e_2 \longrightarrow^0 e_2\ [a \mapsto v_1^0]$$
(E-Let-Red)

$$\frac{e_2 \longrightarrow^i e_2' \qquad i > 0}{\texttt{let}\ a = v_1^i\ \texttt{in}\ e_2 \longrightarrow^i \texttt{let}\ a = v_1^i\ \texttt{in}\ e_2'}$$
(E-Let-2)

$$\frac{e \longrightarrow^i e'}{\texttt{run}\ e \longrightarrow^i \texttt{run}\ e'}$$
(E-Run)

$$\texttt{run}\ \langle v^1 \rangle \longrightarrow^0 v^1$$
(E-Run-Red)

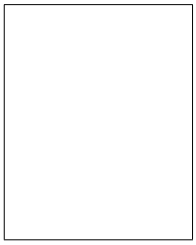
In the evaluation rules for pattern matching below, we use abbreviation $\overline{p \to e}$ for the sequence of cases $p_1 \to e_1 \mid \cdots \mid p_n \to e_n$ where the last pattern p_n may be the default pattern _.

$$\frac{e_0 \longrightarrow^i e_0'}{\texttt{match}\ e_0\ \texttt{with}\ (\overline{p \to e}) \longrightarrow^i \texttt{match}\ e_0'\ \texttt{with}\ (\overline{p \to e})}$$
(E-Match-Scrut)

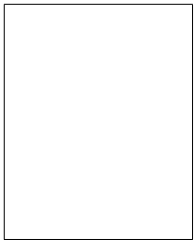
$$\frac{\begin{array}{c}1 \le k \le n \\ \forall j < k.\ \mathcal{M}^0_{[\,]}(v^0, p_j) = \text{fail} \\ \theta = \mathcal{M}^0_{[\,]}(v^0, p_k) \neq \text{fail}\end{array}}{\texttt{match}\ v^0\ \texttt{with}\ (\overline{p \to e}) \longrightarrow^0 e_k\ \theta}$$
(E-Match-Succ)

$$\frac{\forall j \le n.\ \mathcal{M}^0_{[\,]}(v^0, p_j) = fail}{\texttt{match}\ v^0\ \texttt{with}\ (\overline{p \to e} \mid \_ \to e') \longrightarrow^0 e'}$$
(E-Match-Default)

$$\frac{e_0 \longrightarrow^i e_0' \qquad i > 0}{\begin{array}{l}\texttt{match}\ v_0^i\ \texttt{with}\ (\overline{p \to v^i} \mid p' \to e_0 \mid \overline{p'' \to e'}) \longrightarrow^i \\ \texttt{match}\ v_0^i\ \texttt{with}\ (\overline{p \to v^i} \mid p' \to e_0' \mid \overline{p'' \to e'})\end{array}}$$
(E-Match-Plus)



Ya Mone Zin has graduated from the University of Computer Studies Yangon, Myanmar. and studies at Master's Program in Computer Science, the University of Tsukuba. She is interested in the design and the implementation of staged programming languages.



Yukiyoshi Kameyama received his M.S. degree from the University of Tokyo, and his Ph.D. degree from Kyoto University. He is a full professor at the Department of Computer Science, the University of Tsukuba. His research interests include staged computation, functional programming and type theory, and programming logic. He is a member of ACM, JSSST, and IFIP WG2.11 (Program Generation).