Staged Gradual Typing

Hiroto Yaguchi
University of Tsukuba
Tsukuba, Japan
yaguchi@logic.cs.tsukuba.ac.jp

Yukiyoshi Kameyama
University of Tsukuba
Tsukuba, Japan
kameyama@acm.org

Abstract

Staging dynamically typed programming languages safely is a challenge, as the programming-language support for staged computation typically relies on static type systems. To solve this problem, we propose a staged gradual type system that seamlessly integrates static and dynamic typing with staged computation. Our system combines the basic gradual type system and the let-polymorphic staged type system for run-time code generation safely. We discuss the design and issues in developing the calculus, and present a type system and operational semantics via a translation to a cast calculus where dynamic type checking is made explicit. We also show several applications, such as lightweight stage polymorphism.

CCS Concepts: • **Theory of computation → Type structures**; • **Software and its engineering → General programming languages**.

Keywords: staging, run-time code generation, gradual typing, type soundness, lexical scope

ACM Reference Format:
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), July 3–4, 2025, Bergen, Norway*. ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3742876.3742880

1 Introduction

Staged computation a la MetaML [23] distinguishes itself from other sorts of program-generation techniques by a static guarantee for safety properties such as well-typedness and well-scopedness. The *static* guarantee means that a program generator is checked *before* being executed against parameters, and it is guaranteed that the generated code is safe regardless of the values of the parameters.

While this paradigm has been proved effective in a variety of application domains, it is not always the best; this strong guarantee can impose a strong restriction on the applicability of staging; we sometimes encounter situations where the safety property of generated codes depends on the values of its inputs, while the inputs are really dynamic in the sense that some combination of inputs may lead to unsafe programs to be generated. For another example, a program generator has to manipulate heterogeneous data and programs, which can be handled by a sophisticated type system, but we may want to have a simpler solution that does not need an involved type system for the metaprogramming language.

Note that we still want to ensure certain safety properties for the program generator and generated programs, so ignoring static type checking completely is not our solution. For instance, the program generator itself should not be ill-scoped, or sometimes, there is a case where the program generator can raise an exception, but the generated code should be statically typed if program generation succeeds without causing an exception. Hence, totally abandoning a type system is not desirable. We need a way to express both static and dynamic types for staged programming languages.

Gradual typing [16, 18] has been proposed to mediate static typing and dynamic typing seamlessly in a single language. Technically, it extends a static type system with the type ? (or Any), which designates program fragments whose type-checking is postponed to runtime. Program fragments that have non- ? types are type-checked as in a static type system, hence a programmer can combine statically and dynamically typed fragments in a program. Gradual typing can serve as a tool for gradual development from an ill-typed program to a well-typed program.

This paper introduces a gradually typed calculus for staged computation. Our aim is to build a staged programming language whose application needs both static type checking and dynamic type checking (type checking after programs are generated). We also aim to building a solid foundation for program generation, where the object language for generated programs is gradually typed, for instance, TypeScript has the type Any that allows program fragments to bypass static type checking. We design our calculus in the hope that it serves as the foundation of a staged extension of TypeScript. In summary, our calculus should have the type Any at both stages, namely, at the present stage when a program generator is executed to generate a program, and at the future stage when a generated program is executed. Indeed, we allow more than two stages, and the type Any can be used at all stages in our calculus.

GPCE '25, Bergen, Norway

ACM ISBN 979-8-4007-1995-0/25/07
https://doi.org/10.1145/3742876.3742880

In this paper, we take the minimalist approach to focus on the foundational issues, including the design space and the basic properties. Hence, we combine the simple gradual type system proposed by Siek and Taha [16] with a polymorphic extension of the staged calculus λ° [6]. The latter allows the manipulation of open code, and has served as the foundation for several staged programming languages such as BER MetaOCaml [11].

The contribution of this paper is summarized as follows:

- We propose a type system for $\lambda^{G\circ}$, the gradually typed calculus for staged computation.
- We present a type-preserving translation from $\lambda^{G\circ}$ to a cast calculus λ^C, and give operational semantics for the latter. Together, they define an operational semantics for $\lambda^{G\circ}$.
- We show illustrative examples of our calculus.
- We show a few properties including type preservation and decidability of type checking for λ^C.

The rest of this paper is organized as follows: Sect. 2 informally gives a few programming examples in our calculus and explains the issues addressed in this paper. Sect. 3 argues the design decisions we made and mentions technical issues in this work. Sect. 4 presents our calculus $\lambda^{G\circ}$ for gradually typed staged computation. Sect. 5 gives a cast calculus, and a translation from $\lambda^{G\circ}$ to it, and Sect. 6 presents its operational semantics, and desirable properties such as the subject reduction property. Sect. 7 shows an illustrative example of $\lambda^{G\circ}$ with type case. In Sect. 8, we compare our work with related work, and Sect. 9 concludes the paper.

2 Preliminaries

In this section, we introduce several examples to address the issues in combining a staged programming language and a gradually typed system.

2.1 Basics of Staged Computation

We take the MetaML approach for staged computation, which uses quasi-quotation (anti-quotation) for generating code, is purely generative in the sense that the decomposition of generated code is not allowed, and allows run-time code generation as opposed to compile-time code generation (e.g., macros) such as Template Haskell and MacoCaml. The MetaML approach is realized by MetaOCaml, a staged extension of OCaml, that has been used for various practical application domains.

We start with a simply typed lambda calculus extended with staging constructs, as illustrated by the following example:

$$e_1 \equiv (\lambda x.\ <\sim x * 5 + \sim x>) < 2 + 3 >$$
$$\hookrightarrow\ <(2+3) * 5 + (2+3)>$$

The term $<e>$ generates a code, which is not evaluated at the present stage, but its evaluation is delayed. The term $\sim e$ splices a code into another code, as shown above. We can run a generated code within the language as follows:

$$\textbf{run}\ e_1 \hookrightarrow 30$$

The purpose of the type systems for staged programming languages is to guarantee safety properties such as well-stagedness (there is no confusion of stages), well-scopedness (there are no scope errors for variables), and well-typedness. To understand them, we consider the following terms where f is a polymorphic function of type $\forall\beta.\ \beta \to \beta$ (which is the type for the identity function):

$$e_2 \equiv <\lambda x : \textbf{int}\ .\ \sim(f\ x)>$$
$$e_3 \equiv <\lambda x : \textbf{int}\ .\ \sim(f\ <x>)>$$
$$e_4 \equiv <\lambda x : \textbf{int}\ .\ \sim(\textbf{run}\ (f\ <x>);\ <3>)>$$

The term e_2 should be rejected by any reasonable type system since it is ill-staged: While the variable x is bound inside a bracket, which implies x is a future-stage (or level-1) variable, its use site (the term $f\ x$) is inside a spliced term, namely, $f\ x$ is evaluated at the present stage (at level 0). To evaluate this term, we need to know the future value of x at the present stage, violating the chronological order of stages.

The term e_3 is well-typed since x is used at the future stage (x appears in $<x>$), so stages are respected.

The term e_4 is problematic.[1] Since $<x>$ is an open code (it contains a free variable x), the value of $f\ <x>$ may be an open code, and running it would potentially violate the lexical scope of variables. Therefore, a static type system must reject e_4 regardless of the actual value of f.

Preventing an open code from being run has been a major goal of designing a type system for staged programming languages [2, 12, 21]. In this work, however, we use a simpler type system than theirs. Hence, we cannot statically detect the closedness of a code before running it, and instead, we will dynamically check it. It is left for future work to extend our calculus to statically detect closed code.

2.2 Basics of Gradual Typing

Gradual typing [16] combines static typing and dynamic typing by introducing the type $?$ (the type Any) to static typing. A program fragment of type $?$ is not typechecked at compile time (statically), but its value will be typechecked at runtime (dynamically). Consider the following term:

$$e_5 \equiv (\lambda x :\ ?\ .\ x + 3)\ \texttt{true}$$

The term e_5 typechecks in a gradual type system: $(\lambda x :\ ?\ .\cdots)\ \texttt{true}$ is well typed since the types **bool** and $?$ are consistent, and $x + 3$ typechecks under the typing context $x : ?$, since $?$ and **int** are consistent.

To evaluate the term e_5, we translate gradually typed terms into those in a cast calculus where the dynamic type checking

[1] Readers can ignore the subterm <3> in e_4, which is inserted here to keep typability, but our analysis does not depend on this subterm.

is made explicit by the cast operation $[\tau]e$.[2] The above term e_5 is translated to a term e_5' in the cast calculus:

$$e_5' \equiv (\lambda x : ? \ . ([\mathbf{int}]x) + 3) \ ([\ ?\ ]\texttt{true})$$

which has two cast operations [**int**] and [?]. When evaluating e_5' under the call-by-value strategy, we first check if `true` has type ?, which succeeds. Then, we substitute `true` for x, evaluate the body, and finally check if the value of x has type **int**, which fails. This way, delayed typechecking caused by the type ? is executed at runtime.

Since Siek and Taha's seminal paper introduced the simplest possible gradual type system, a number of authors have proposed various extensions of gradual type systems, for instance, [3, 8, 14, 17]. In this work, we use the original calculus by Siek and Taha [16] to focus on the most essential features.

3 Design Decisions and Technical Issues

Our goal is to build a single calculus that combines gradual typing and staged computation. This section explains our design decisions using illustrative examples, and then discusses technical issues to be addressed to achieve the goal.

3.1 Design Decisions

To design a type system for our combined calculus, we revisit the examples in Sect. 2.1, and replace several types by the any type (denoted by ?). Below we assume that f is a term of type $? \to ?$.

$$e_2' \equiv <\lambda x : ? \ . \sim (f\ x)>$$
$$e_3' \equiv <\lambda x : ? \ . \sim (f\ <x>)>$$
$$e_4' \equiv <\lambda x : ? \ . \sim (\mathbf{run}\ (f\ <x>);\ <3>)>$$

The primed terms e_i' assign the type ? to the variable x, which implies that, we do not statically typecheck it.

We think that the term e_2' should be rejected. To run the subterm $f\ x$, the value of x is needed, but it is a future-stage variable, and its value is not available at the present stage. Hence, running e_2' would cause a serious error. In short, it violates well-stagedness.

The term e_3' is non-problematic, since $<x>$ is allowed, and the static type of $f\ <x>$ is ?, which implies that we will not check if it has a code type at the present stage.

The term e_4' is the most interesting example. If f is an identity function, we need to evaluate **run** $<x>$, but $<x>$ is an open code, and hence, the evaluation of e_4' would cause an error. On the other hand, if $f\ <x>$ returns a code like <10>, then running such a code is not problematic. To handle these two cases, our calculus should accept e_4', and insert dynamic type checking into this term that checks if the argument of the run operator is a closed code or not.

[2]The cast operation is usually written as $<\cdot>\cdot$, but we use the notation $[\cdot]\cdot$ to avoid conflict with brackets.

In summary, we employ the following design principles for staged gradual typing:

- The type system should ensure well-scopedness (lexical scope).
- The type system should ensure well-stagedness.
- The type system should allow ill-typed terms by the type ?. More precisely, it should allow both static type-checking and dynamic type-checking, and the programmer can choose either one.

Following the design choices above, we design our type system by adjusting typing rules. The adjustment is straightforward for constructor terms such as λ. For eliminators such as application, we need to have an extra rule which incorporates the case when the term decomposed by the rule has type ?.

For instance, the splice-term is an eliminator, hence we need to add one typing rule as follows:

$$\frac{\Gamma \vdash^{i} \ e \ : \ ?}{\Gamma \vdash^{i+1} \ \sim e \ : \ ?}\ \text{Esc2}$$

This rule means that when the subterm e has type ?, then $\sim e$ is still well typed with type ?.

Our type system will be shown in Sect. 4.

3.2 Technical Issues

In the development of our calculus, we found a few technical issues to be addressed in this work.

The first issue is related to dynamic type checking. Recall the term e_3' in the previous section. Following Siek and Taha's approach, we translate it to the following term e_3'':

$$e_3'' \equiv <\lambda x : ? \ . \sim ([\langle\ ?\ \rangle](f\ ([\ ?\ ]<x>)))>$$

Suppose f is the identity function, then $f\ ([\ ?\ ]<x>)$ returns $[\ ?\ ]<x>$.[3] Then, we need to check if $<x>$ has type $\langle\ ?\ \rangle$, that requires the type information of x. In general, staged computation allows evaluation *under* binders. It follows that dynamic checking also needs to be done *under* binders. Hence, we need the type information (the typing context) of bound variables.

We solve this issue by making the operational semantics of the cast calculus slightly more involved. Namely, our operational semantics should collect the typing context of bound variables from the evaluation contexts. In the above typecheck, the evaluation context is $<\lambda x : ? \ . \sim \bullet>$, where the hole $\bullet$ designates the place for the redex. From this evaluation context, we can extract the typing context $(x : ?)^1$. Alternatively, we can collect the type information of bound variables at the translation time, and attach it to the cast term. For example, the above cast can be changed to $[\Gamma \vdash \tau]$ where Γ is the type information of bound variables. We do not take this approach, since Γ can always be recovered from the evaluation context if we assume that we evaluate closed terms only.

[3]In our formulation, $[\ ?\ ]v$ is a value if v is a value.

Variable	$x \in X$
Term	$e ::= x \mid c \mid \lambda x : \tau\,.\,e \mid e\ e \mid \texttt{<}e\texttt{>} \mid {\sim}e \mid \textbf{run}\ e$ $\mid \textbf{let}\ x : \tau\ =\ e\ \textbf{in}\ e$

Figure 1. Terms of $\lambda^{G\circ}$

Stage	s is a natural number
Basic Type	$b \in BT$
Type Variable	$\beta \in TV$
Type	$\tau ::= b \mid \beta \mid ? \mid \tau \to \tau \mid \langle\tau\rangle$
Type Scheme	$\sigma ::= \tau \mid \forall\beta.\sigma$

Figure 2. Stages, Types and Type Schemes

(For types)

$$\frac{}{\tau \sim \tau} \qquad \frac{}{\tau \sim\ ?} \qquad \frac{}{?\ \sim \tau}$$

$$\frac{\tau_1 \sim \tau_1' \qquad \tau_2 \sim \tau_2'}{\tau_1 \to \tau_2 \sim \tau_1' \to \tau_2'} \qquad \frac{\tau_1 \sim \tau_2}{\langle\tau_1\rangle \sim \langle\tau_2\rangle}$$

(For type schemes)

$$\frac{\sigma \sim \sigma'}{\forall\beta.\sigma \sim \forall\beta.\sigma'}$$

Figure 3. Type Consistency

The second issue is caused by the interplay between polymorphism and dynamic type checking. Since we extract types for dynamic type checking, we need to be careful about free type variables in these types, which should not be ill-scoped. For this purpose, we translate the source calculus to an intermediate calculus where universally quantified type variables are made explicit.

The third issue is related to the **run** primitive. To ensure that the argument of the **run** primitive must be closed, which is a necessary condition for a generated code to be run safely, the staged calculi in the literature required rather involved type systems, that would complicate dynamic type checking in our formulation. In this work, we take a simple type system for staged computation, and rely on dynamic closedness checking before running the generated code.

4 The Calculus $\lambda^{G\circ}$

We introduce $\lambda^{G\circ}$, a gradually typed multi-stage calculus. It is based on two classic calculi: the first one is λ^{G}, a gradually typed calculus proposed by Siek and Taha [16], which has the type ? (or Any) to allow dynamic type checking. The second one is a polymorphic extension of λ°, a staged calculus proposed by Davies [6]. The latter allows the manipulation of open code.

In this section, we give the calculus and its type system.

4.1 Syntax of $\lambda^{G\circ}$

Fig. 1 gives the syntax of terms of $\lambda^{G\circ}$.

In this figure, X denotes a set of variables. Terms are those in the standard lambda calculus with the **let** expression, extended with staging constructs: a bracket term <e> for generating a code, a splice term $\sim e$ for splicing a code into another code, and a run term **run** e for running a code. In this work, we omit cross-stage persistence (CSP) that allows a value created at some stage to be used in a later stage, since CSP would complicate evaluation rules [21].

Our calculus uses the Church style in the sense that all bound variables are explicitly given their types. It is desirable to formulate a Curry-style calculus, and develop a type inference algorithm, but we leave it for future work.

Variables are bound by λ and **let**, and we identify α-equivalent terms as usual.

4.2 Stages and Types

Fig. 2 defines stages, types and type schemes where BT and TV, resp. denote mutually distinct sets of basic types, and type variables, resp.

In this calculus, a stage is simply a natural number, where the stage 0 means the present stage when a code is generated, and the stage 1 means the next stage when the generated code is executed. We allow $i > 1$ stages, in which case executing a generated code will generate another code, and so on. Historically, more involved machineries such as environment classifiers [2, 21] and contextual modal type theory [10, 15] have been used to distinguish closed code from arbitrary code, to allow the safe evaluation of a run term. In this work, we do not employ these machineries to avoid clutter. Instead, we will discuss how we can introduce a run-primitive with the help of environment classifiers.

A type is either a basic type, a type variable, the type ? (sometimes written as Any), a function type, or a code type $\langle\tau\rangle$. For instance, the term <<<$x + y$>>> has the type $\langle\langle\langle\textbf{int}\rangle\rangle\rangle$ if x and y are stage-3 variables of type **int**.

The calculus $\lambda^{G\circ}$ has ML-style let-polymorphism where type variables are abstracted. For this purpose, types in Fig. 2 represent monomorphic types, while type schemes represent polymorphic types abstracted by type variables β. In type schemes, $\forall$ binds type variables, and we identify α-equivalent types as usual.

4.3 Type Consistency

In Siek and Taha's formulation of gradual typing, the type ? designates dynamically typed terms, and all other type constructors behave the same as statically typed calculi. To

allow dynamic typing by $?$, they introduced the notion of type consistency; two types τ_1 and τ_2 are consistent if they differ only at the occurrence of $?$. For instance, $b \to (? \to \beta)$ is consistent with $? \to ?$, but not with $b \to (b \to b)$ or $(? \to ?) \to ?$.

Fig. 3 formally defines type consistency for types and type schemes.

4.4 Type System

The type system of $\lambda^{G\circ}$ is given by the typing rules in Fig. 4.

Before explaining each rule, we first introduce a few auxiliary notions.

- We say a type τ is an instance of a type scheme σ, written by $\sigma \succ \tau$, if σ has the form $\forall\beta_1.\cdots\forall\beta_m.\tau_0$, and τ is $\tau_0[\beta_1 := \tau'_1, \cdots \beta_m := \tau'_m]$ for some types $\tau'_1, \cdots, \tau'_m$ where $\cdot[\cdot := \cdot]$ denotes the usual substitution. The instance relation is common in ML-like let-polymorphic calculi.
- For any syntactic entity X, we write $\mathbf{Free}(X)$ for the set of free types in X. For example, $\mathbf{Free}(\forall\beta_2.(\langle\beta_1\rangle \to \langle\beta_2\rangle))$ is $\{\beta_1\}$.
- $\overline{\beta}$ denotes a sequence $\beta_1, \cdots, \beta_n$.

A judgment in $\lambda^{G\circ}$ takes the form

$$(x_1 : \sigma_1)^{s_1}, \cdots, (x_k : \sigma_k)^{s_k} \vdash^s e : \tau$$

where x_i are variables, σ_i are type schemes, s_i and s are stages (natural numbers), e is a term, and τ is a type. It is mostly standard except that each assumption and the judgment itself are annotated by a stage. Also, $\lambda^{G\circ}$ has let-polymorphism which allows type schemes in the assumption (left to $\vdash$), but allows only types in the conclusion (right to $\vdash$).

Let us explain each typing rule. The rules Var and Lam are as usual except that we have stages that annotate assumptions and judgments.

For an application term $e_1\ e_2$, we have two rules App1 and App2. The rule App1 is used when the term e_1 is known to have a function type $\tau_1 \to \tau_2$. This rule generalizes the standard application rule in that the argument e_2 may have a different type τ' than the expected type τ_1, but they must be consistent $\tau_1 \sim \tau'$. If these two terms are not identical, we will perform dynamic type checking. The second rule App2 is used when the type of e_1 is statically unknown. For this case, e_2 still needs to be well typed, but its type can be an arbitrary type τ. The resulting type of $e_1\ e_2$ is also unknown.

It is worth mentioning that the typing rule for a constructor such as λ remains essentially the same as the typing rules in a static type system, while the rule for an eliminator such as application needs to be split into two rules, as shown above.

For staging constructs, a bracket term has essentially the same typing rule as that in λ°, while a splice term $\sim e$ needs two typing rules just like an application term.

The first rule for splice, Esc1, is used when e is known to have a code type, in which case the rule is essentially the same as one in λ°. The second rule for splice, Esc2, is used when the type of e is unknown statically. It is handled similarly to the rule App2.

The two rules for **run**, Run1 and Run2, are designed in the same way as App1 and App2.

Finally, the rule Let is formulated in the same way as the standard let-rule in ML-like polymorphic languages. The type τ_1 of e_1 is made polymorphic over type variables $\overline{\beta}$ provided they appear in τ_1 freely, and must not appear in Γ freely.

This concludes the type system of $\lambda^{G\circ}$.

5 Translation to Cast Calculus

We shall give the semantics for the calculus $\lambda^{G\circ}$ via a translation to the cast calculus. The cast calculus λ^{C} is similar to $\lambda^{G\circ}$, but it has the explicit *cast* operation[4] for dynamic type-checking, written by $[\tau]e$.

As explained in Sect. 3.2, our translation inserts the cast operators, namely, a term e is translated to $[\tau]e$ if the dynamic type checking for e is needed. The type information τ is extracted from static typechecking, which implies that the input of the translation is not just a term, but a type derivation of the term.

It is important to note that the evaluation in the cast calculus needs to handle types explicitly, therefore, we need to make it explicit that how a type scheme is instantiated with a concrete type at each use of a polymorphic variable. Hence, we modify the Var rule so that a variable is accompanied by a sequence of types, such as $x\{\tau_1, \cdots, \tau_n\}$. For instance, the $\lambda^{G\circ}$ term $\mathbf{let}\ x : \beta \to \beta\ =\ \lambda y : \beta\ .\ y\ \mathbf{in}\ (x\ 5);\ (x\ \mathit{true})$ becomes a more verbose term:

$$\mathbf{let}\ x : \forall\beta.\beta \to \beta\ =\ \Lambda\beta.\lambda y : \beta\ .\ y\ \mathbf{in}$$
$$(x\{\mathbf{int}\}\ 5);\ (x\{\mathbf{bool}\}\ \mathit{true})$$

in the cast calculus. The subterm $x\{\mathbf{int}\}$ means that the type scheme $\forall\beta.\ \beta \to \beta$ has been instantiated by **int**, and the type of this subterm is $\mathbf{int} \to \mathbf{int}$. Similarly for $x\{\mathbf{bool}\}$. Then, we can smoothly define operational semantics for the resulting calculus. For instance, the above term evaluates to $((\lambda y : \mathbf{int}\ .\ y)\ 5);\ ((\lambda y : \mathbf{bool}\ .\ y)\ \mathit{true})$.

In this section, we first introduce the calculus λ^{C}, then translate $\lambda^{G\circ}$ to λ^{C}, and finally, define the operational semantics of λ^{C}. By composing the ingredients, we obtain the operational semantics of $\lambda^{G\circ}$.

[4]Despite the name, the cast operation in this paper does not change the type of a term, and merely checks it. We follow the terminology in the literature [16].

$$\frac{(x:\sigma)^s \in \Gamma \qquad \sigma \succ \tau}{\Gamma \vdash^s x : \tau}\ \text{Var} \qquad \frac{\Gamma, (x:\tau_1)^s \vdash^s e : \tau_2}{\Gamma \vdash^s \lambda x:\tau_1.\, e : \tau_1 \to \tau_2}\ \text{Lam}$$

$$\frac{\Gamma \vdash^s e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash^s e_2 : \tau' \qquad \tau_1 \sim \tau'}{\Gamma \vdash^s e_1\ e_2 : \tau_2}\ \text{App1} \qquad \frac{\Gamma \vdash^s e_1 : ? \qquad \Gamma \vdash^s e_2 : \tau}{\Gamma \vdash^s e_1\ e_2 : ?}\ \text{App2}$$

$$\frac{\Gamma \vdash^{s+1} e : \tau}{\Gamma \vdash^s <e> : \langle\tau\rangle}\ \text{Brkt} \qquad \frac{\Gamma \vdash^s e : \langle\tau\rangle}{\Gamma \vdash^{s+1} \sim e : \tau}\ \text{Esc1} \qquad \frac{\Gamma \vdash^s e : ?}{\Gamma \vdash^{s+1} \sim e : ?}\ \text{Esc2}$$

$$\frac{\Gamma \vdash^s e : \langle\tau\rangle}{\Gamma \vdash^s \mathbf{run}\ e : \tau}\ \text{Run1} \qquad \frac{\Gamma \vdash^s e : ?}{\Gamma \vdash^s \mathbf{run}\ e : ?}\ \text{Run2}$$

$$\frac{\Gamma \vdash^s e_1 : \tau' \qquad \Gamma, (x:\overline{\forall\beta}.\tau_1)^s \vdash^s e_2 : \tau_2 \qquad \{\overline{\beta}\} \subseteq \mathbf{Free}(\tau_1) - \mathbf{Free}(\Gamma) \qquad \tau' \sim \tau_1}{\Gamma \vdash^s \mathbf{let}\ x:\tau_1 = e_1\ \mathbf{in}\ e_2 : \tau_2}\ \text{Let}$$

Figure 4. Type System for $\lambda^{G\circ}$

5.1 Cast Calculus λ^C

We extend $\lambda^{G\circ}$ by a cast term to λ^C.

$$\textbf{Term} \qquad e ::= \ldots \mid x\{\overline{\tau}\} \mid [\tau]e \mid \mathbf{let}\ x:\overline{\forall\beta}.\tau = \overline{\Lambda\beta}.e\ \mathbf{in}\ e'$$

where $x\{\overline{\tau}\}$ denotes a variable with type instantiation, and $[\tau]e$ is a cast term for dynamic type checking. We write x for $x\{\}$. The syntax of **let** is more vrebose than that in $\lambda^{G\circ}$.

The typing rules for new terms are given as follows:

$$\frac{(x:\sigma)^s \in \Gamma \qquad \sigma = \overline{\forall\beta}.\tau}{\Gamma \vdash^s x\{\overline{\tau'}\} : \tau[\overline{\beta} := \overline{\tau'}]}\ \text{Var'}$$

$$\frac{\Gamma \vdash^s e : \tau \qquad \tau \sim \tau'}{\Gamma \vdash^s [\tau']e : \tau'}\ \text{Cast}$$

We use the same typing rules as Lam, Brkt, Esc1, and Run1, and remove App2, Esc2, and Run2. The rule App1 is modified so that $\tau_1 \sim \tau'$ is replaced by $\tau_1 = \tau'$.

The rule Let is slightly modified as follows:

$$\frac{\Gamma \vdash^s e_1 : \tau_1 \qquad \Gamma, (x:\overline{\forall\beta}.\tau_1)^s \vdash^s e_2 : \tau_2 \qquad \{\overline{\beta}\} \subseteq \mathbf{Free}(\tau_1) - \mathbf{Free}(\Gamma)}{\Gamma \vdash^s \mathbf{let}\ x:\overline{\forall\beta}.\tau_1 = \overline{\Lambda\beta}.e_1\ \mathbf{in}\ e_2 : \tau_2}\ \text{Let'}$$

The modified rule has the same assumptions as the Let rule, while in the conclusion, we have made explicit which type variables are universally quantified by $\overline{\forall\beta}$. We have added the type abstraction $\Lambda\beta.$ to the term $[\tau' \Rightarrow \tau_1]e_1$, to make it easier to understand the term. When it is substituted for x in e_2, the type variables $\overline{\beta}$ are instantiated with $\overline{\tau}$ specified by type application $x\{\overline{\tau}\}$.

Since we use the type system of λ^C during runtime, it is important to have the following properties.

Theorem 5.1 (Decidable typability)**.** *The type checking problem in λ^C is decidable. In other words, given Γ, s, and e, there is an algorithm to decide whether $\Gamma \vdash^s e : \tau$ is derivable for some type τ in λ^C. Moreover, this type τ is unique.*

Proof sketch. Since λ^C is in the Church style, all variables are given their types (and stages) in e or Γ. Hence, we can build a type derivation for e from the bottom-up, and this process is unique. In $\lambda^{G\circ}$, there is freedom about the choice of universally quantified type variables $\overline{\beta}$, however, in λ^C, they are explicitly shown in the let-term, so we can fully recover the type derivation. The uniqueness of the type is also obvious from this argument. □

5.2 Translation from $\lambda^{G\circ}$ to λ^C

Fig. 5 gives our translation from $\lambda^{G\circ}$ to λ^C. A judgment takes the form $\Gamma \vdash^s e \Rightarrow e' : \tau$, which intuitively means that e is translated to e' under the context Γ at stage s with type τ. In the translation, we use the notation $[\tau \Rightarrow \tau']e$, which is e if τ is identical to τ', and $[\tau']e$ otherwise.

The translation needs not only a term e, but also a typing derivation for e as input. The name of each typing rule is in the form Cname, where name is the corresponding typing rule in Fig. 4.

We shall explain a few interesting rules. The rule CVar translates a variable to a variable with type instantiation. The rule CApp1 inserts a cast $[\tau' \Rightarrow \tau_1]e_2'$ since we use the consistency $\tau_1 \sim \tau'$ to derive this term. The rule CApp2 enforces that e_1' needs to have a function type whose argument type is τ. Similarly for the rules CEsc2, CRun2 and CLet.

The above definitions work in the sense that the translation is well-defined and preserves typing.

Theorem 5.2 (Type Preserving Translation)**.** *Suppose $\Gamma \vdash^s e : \tau$ is derivable in $\lambda^{G\circ}$. Then, $\Gamma \vdash^s e \Rightarrow e' : \tau$ is derivable for some term e' such that $\Gamma \vdash^s e' : \tau$ is derivable in λ^C.*

Proof outline. The theorem is proved by straightforward induction on the type derivation. $\Gamma \vdash^s e : \tau$. Also, given a type derivation, the choice of e' in the theorem is essentially unique. □

$$\frac{(x:\sigma)^s \in \Gamma \qquad \sigma = \overline{\forall\beta}.\tau_0}{\Gamma \vdash^s x \Rightarrow x\{\overline{\tau'}\} \ : \ \tau_0[\overline{\beta} := \overline{\tau'}]}\ \text{CVar} \qquad \frac{\Gamma, (x:\tau_1)^s \vdash^s e \Rightarrow e' \ : \ \tau_2}{\Gamma \vdash^s \lambda x:\tau_1 \,.\, e \Rightarrow \lambda x:\tau_1 \,.\, e' \ : \ \tau_1 \to \tau_2}\ \text{CLam}$$

$$\frac{\Gamma \vdash^s e_1 \Rightarrow e_1' \ : \ \tau_1 \to \tau_2 \qquad \Gamma \vdash^s e_2 \Rightarrow e_2' \ : \ \tau' \qquad \tau_1 \sim \tau'}{\Gamma \vdash^s e_1\ e_2 \Rightarrow e_1'\ ([\tau' \Rightarrow \tau_1]e_2') \ : \ \tau_2}\ \text{CApp1} \qquad \frac{\Gamma \vdash^s e_1 \Rightarrow e_1' \ : \ ? \qquad \Gamma \vdash^s e_2 \Rightarrow e_2' \ : \ \tau}{\Gamma \vdash^s e_1\ e_2 \Rightarrow ([\tau \to ?]e_1')\ e_2' \ : \ ?}\ \text{CApp2}$$

$$\frac{\Gamma \vdash^{s+1} e \Rightarrow e' \ : \ \tau}{\Gamma \vdash^s <e> \Rightarrow <e'> \ : \ \langle\tau\rangle}\ \text{CBrkt} \qquad \frac{\Gamma \vdash^s e \Rightarrow e' \ : \ \langle\tau\rangle}{\Gamma \vdash^{s+1} \sim e' \ : \ \tau}\ \text{CEsc1} \qquad \frac{\Gamma \vdash^s e \Rightarrow e' \ : \ ?}{\Gamma \vdash^{s+1} \sim e \Rightarrow \sim[\langle\, ?\,\rangle]e' \ : \ ?}\ \text{CEsc2}$$

$$\frac{\Gamma \vdash^s e \Rightarrow e' \ : \ \langle\tau\rangle}{\Gamma \vdash^s \mathbf{run}\ e \Rightarrow \mathbf{run}\ e' \ : \ \tau}\ \text{CRun1} \qquad \frac{\Gamma \vdash^s e \Rightarrow e' \ : \ ?}{\Gamma \vdash^s \mathbf{run}\ e \Rightarrow \mathbf{run}\ [\langle\, ?\,\rangle]e' \ : \ ?}\ \text{CRun2}$$

$$\frac{\Gamma \vdash^s e_1 \Rightarrow e_1' \ : \ \tau' \qquad \Gamma, (x:\overline{\forall\beta}.\tau_1)^s \vdash^s e_2 \Rightarrow e_2' \ : \ \tau_2 \qquad \{\overline{\beta}\} \subseteq \mathbf{Free}(\tau_1) - \mathbf{Free}(\Gamma)}{\Gamma \vdash^s \mathbf{let}\ x:\tau_1 = e_1\ \mathbf{in}\ e_2 \Rightarrow \mathbf{let}\ x:\overline{\forall\beta}.\ \tau_1 = \overline{\Lambda\beta}.\ [\tau' \Rightarrow \tau_1]e_1'\ \mathbf{in}\ e_2' \ : \ \tau_2}\ \text{CLet}$$

Figure 5. Translation from $\lambda^{G\circ}$ to λ^C

6 Semantics of Cast Calculus

To complete the picture, we give the operational semantics of the cast calculus.

Evaluation of the cast calculus employs the call-by-value strategy. The notion of values is dependent on stages, so we index a value by a stage s, and write v^s for stage-s values.

$$\begin{aligned}
u &::= x \mid \lambda x:\tau\,.\,e \mid <v^1> \\
v^0 &::= u \mid [\ ?\ ]u \\
v^1 &::= x \mid \lambda x:\tau\,.\,v^1 \mid v^1\ v^1 \mid <v^2> \mid \mathbf{run}\ v^1 \\
&\quad \mid [\tau]v^1 \mid \mathbf{let}\ x\ =\ v^1\ \mathbf{in}\ v^1 v^1 \\
(s>1)\ v^s &::= x \mid \lambda x:\tau\,.\,v^s \mid v^s\ v^s \mid <v^{s+1}> \mid \sim v^{s-1} \mid \mathbf{run}\ v^s \\
&\quad \mid [\tau]v^s \mid \mathbf{let}\ x\ =\ v^s\ \mathbf{in}\ v^s v^s
\end{aligned}$$

The intuition behind this definition is that we are concerned with the stage-0 computation only, and v^s means it is a stage-s term that does not have a stage-0 redex. Such a redex may exist in a stage $s > 0$ term, since it may contain a splice term. For instance, $(\lambda x : \mathbf{int}\,.\,x)\ 3$ is not a stage-0 value, but is a stage-1 value, hence, $<(\lambda x : \mathbf{int}\,.\,x)\ 3>$ is a stage-0 value. The term $<(\lambda x : \mathbf{int}\,.\,\sim e)\ 3>$ is not a stage-0 value.

We use the following function *unbox* to remove a cast from a stage-0 value:

$$\begin{aligned}
&unbox\ u = u \\
&unbox\ ([\ ?\ ]u) = u
\end{aligned}$$

We define the call-by-value, operational semantics of λ^C in the small-step semantics.

Fig. 6 gives evaluation rules for non-cast terms, and Fig. 7 gives evaluation rules for cast terms.

The basic judgment in the semantics takes the form of: $\Gamma \rhd e_1 \hookrightarrow^s e_2$ where Γ is a typing context, s is a stage, and e_1 and e_2 are stage-s terms. As we have discussed in Sect. 3, we need a context Γ to evaluate a cast term $[\tau]e$, since, the term being evaluated may be a (locally) open term in staged calculi, and we need to typecheck such an open term in cast calculi. We can compute the necessary typing context Γ smoothly in the style of structural operational semantics (SOS).

The evaluation rules in Fig. 6 are indexed by stages. The stage-0 evaluation is similar to standard lambda calculi, while the stage-s ($s > 0$) evaluation does not evaluate the term, except for the spliced terms. For example, $\lambda x : \tau\ .e$ is a value at stage 0, while we may possibly reduce it at stage 1 since e may contain splice terms (Rule ELamI). For instance, the term $<\lambda x : \tau\ .\ \sim((\lambda y : \tau\ .\ y)\ <x>)>$ has a redex inside it. On the other hand, $(\lambda x : \tau\ .e)\ v^s$ is a redex at stage $s = 0$ (EApp0), while it is a value at stage s if e is also a stage-s value.

The ERun0 rule shows how we evaluate a run term at stage 0. Unlike the involved type systems using environment classifiers or contextual modal type theory, our type system λ^C (and $\lambda^{G\circ}$) does not prevent an open code from being an argument of a run term. We, therefore, need to check at runtime if its value is a closed code before running it. Dynamic checking for the closedness of generated code is implemented in the latest version of BER MetaOCaml, which combines static and dynamic checking. Namely, if the run primitive takes as an argument a code which is statically known to be closed, dynamic checking is not called. Otherwise, dynamic checking is inserted into the code, and it will be performed at runtime. Extending our work to formalize the practice of the latest BER MetaOCaml would be an interesting future work.

The ELet0 rule reduces a let-term as we expect. In the substitution, we need to instantiate type variables $\overline{\beta}$ with appropriate concrete types. It is defined straightforwardly except for the variable case $x\{\overline{\tau}\}$, shown below:

$$\begin{aligned}
&x\{\tau_1, \cdots, \tau_n\}[x := \Lambda\beta_1.\cdots.\Lambda\beta_n.v^0] \\
&:= v^0[\beta_1 := \tau_1, \cdots, \beta_n := \tau_n]
\end{aligned}$$

$$\text{ELamI}\ \frac{\Gamma, (x:\tau)^s \rhd e \hookrightarrow^s e' \quad s > 0}{\Gamma \rhd \lambda x:\tau\,.\,e \hookrightarrow^s \lambda x:\tau\,.\,e'} \qquad \text{EApp0}\ \frac{}{\Gamma \rhd (\lambda x:\tau\,.\,e)\ v^0 \hookrightarrow^0 e[x := v^0]}$$

$$\text{EAppI1}\ \frac{\Gamma \rhd e_1 \hookrightarrow^s e_1'}{\Gamma \rhd e_1\ e_2 \hookrightarrow^s e_1'\ e_2} \qquad \text{EAppI2}\ \frac{\Gamma \rhd e \hookrightarrow^s e'}{\Gamma \rhd v^s\ e \hookrightarrow^s v^s\ e'}$$

$$\text{EBrackets}\ \frac{\Gamma \rhd e \hookrightarrow^{s+1} e'}{\Gamma \rhd \mathord{<}e\mathord{>} \hookrightarrow^s \mathord{<}e'\mathord{>}}$$

$$\text{EEscape0}\ \frac{}{\Gamma \rhd\ \sim\mathord{<}v^1\mathord{>} \hookrightarrow^1 v^1} \qquad \text{EEscapeI}\ \frac{\Gamma \rhd e \hookrightarrow^s e'}{\Gamma \rhd\ \sim e \hookrightarrow^s \sim e'}$$

$$\text{ERun0}\ \frac{v^1 \text{ is closed}}{\Gamma \rhd \textbf{run}\ \mathord{<}v^1\mathord{>} \hookrightarrow^0 \mathit{unbox}\ v^1} \qquad \text{ERunI}\ \frac{\Gamma \rhd e_1 \hookrightarrow^s e_1'}{\Gamma \rhd \textbf{run}\ e_1 \hookrightarrow^s \textbf{run}\ e_1'}$$

$$\text{ELet0}\ \frac{}{\Gamma \rhd \textbf{let}\ x : \overline{\forall\beta}.\tau\ =\ \overline{\Lambda\beta}.v^0\ \textbf{in}\ e_2 \hookrightarrow^0 e_2[x := \overline{\Lambda\beta}.v^0]}$$

$$\text{ELetI}\ \frac{\Gamma \rhd e_1 \hookrightarrow^s e_1'}{\Gamma \rhd \textbf{let}\ x : \overline{\forall\beta}.\tau\ =\ \overline{\Lambda\beta}.e_1\ \textbf{in}\ e_2 \hookrightarrow^s \textbf{let}\ x : \overline{\forall\beta}.\tau\ =\ \overline{\Lambda\beta}.e_1'\ \textbf{in}\ e_2}$$

Figure 6. Operational Semantics for λ^C: Non-cast terms

$$\text{ECastB}\ \frac{\Gamma \vdash^0 \mathit{unbox}\ v^0\ :\ b}{\Gamma \rhd [b]v^0 \hookrightarrow^0 \mathit{unbox}\ v^0}$$

$$\text{ECastU}\ \frac{\Gamma \vdash^0 \mathit{unbox}\ v^0\ :\ \tau}{\Gamma \rhd [\ ?\ ]([\tau]v^0) \hookrightarrow^0 [\ ?\ ]v^0}$$

$$\text{ECastF}\ \frac{\Gamma \vdash^0 \mathit{unbox}\ v^0\ :\ \tau_1' \to \tau_2' \qquad \tau_1 \to \tau_2 \sim \tau_1' \to \tau_2' \qquad z \text{ is fresh}}{[\tau_1 \to \tau_2]v^0 \hookrightarrow^0 \Gamma \rhd \lambda z:\tau_1\,.\,[\tau_2]((\mathit{unbox}\ v^0)([\tau_1 \Rightarrow \tau_1']z))}$$

$$\text{ECastC}\ \frac{\Gamma \vdash^0 \mathit{unbox}\ v^0\ :\ \langle\tau'\rangle \qquad \tau \sim \tau'}{\Gamma \rhd [\langle\tau\rangle]v^0 \hookrightarrow^0 \mathord{<}[\tau' \Rightarrow \tau]\ \sim(\mathit{unbox}\ v^0)\mathord{>}}$$

$$\text{ECastI}\ \frac{\Gamma \rhd e \hookrightarrow^s e'}{\Gamma \rhd [\tau]e \hookrightarrow^s [\tau]e'}$$

Figure 7. Operational Semantics for λ^C: Cast terms

The right-hand side of this definition is an ordinary substitution for type variables, which is necessary in this definition, since v^0 may contain types that have free occurrences of the type variable β_i.

The rules in Fig. 7 shall be explained in detail below.

The evaluation rules, except (ECastI), define stage-0 evaluation for a cast term $[\tau]v^0$ where v^0 is already a stage-0 value. We have four cases depending on the type τ.

The ECastB rule is used when τ is a basic type b, in which case we only have to check if v^0 has the type b. Note that v^0 may have free variables of stage $s > 0$, hence we need a typing context Γ built from the surrounding context.

The ECastU rule is used when τ is the any type $?$. If v^0 has another cast $[\tau]$, then we check it and eliminate it if the test succeeds. If v^0 has no cast, then $[\ ?\ ]v^0$ is a value.

The ECastF rule is used when τ is a function type. It looks complicated, but essentially it is the same as the rule in Siek and Taha's work, which η-expands v^0 and splits one single checking (for the type $\tau_1 \to \tau_2$) into two checking (one for τ_1', and the other for τ_2).

The ECastC rule is used when τ is a code type. We designed this rule in the same spirit as the ECastF. Namely, we first check if $\mathit{unbox}\ v^0$ has a type under the context Γ. If it succeeds and returns a type that is consistent with the expected type, we insert another, simpler cast, and continue the evaluation.

The ECastI rule defines the evaluation for cast terms if its argument is not a value at the stage s.

We remark that if there is no applicable rule for a cast-term, the term gets stuck, which means a runtime error.

Note that there is no rule in the form $\Gamma \rhd [\beta]v^0 \hookrightarrow^0 e$ where β is a type variable. Namely, the term $[\beta]v^0$ is stuck. We may encounter such a term during the evaluation since we evaluate under the $\Lambda\beta$-binder (see the ELetI rule). However,

type checking caused by it must fail, since there is no stage-0 value that has no free stage-0 variables[5] and has type β.

For the operational semantics and the type system in the previous section, we have the subject reduction property in the following form.

Theorem 6.1 (Subject Reduction). *Suppose* $\Gamma \vdash^s e : \tau$ *is derivable in* λ^C, *and* $\Gamma \rhd e \hookrightarrow^s e'$ *where* Γ *contains no stage-0 variables, and* $\mathbf{Free}(\Gamma, \tau)$ *is empty.*[6] *Then* $\Gamma \vdash^s e' : \tau$ *is derivable in* λ^C.

Proof outline. We can prove the theorem by induction on $\Gamma \rhd e \hookrightarrow^s e'$. The case for β-reduction (the EApp0 rule in Fig. 6) is proved by the substitution lemma as usual. The case for the bracket-splice reduction (EEscape0) is easy. The case for the run reduction (ERun0) is the most complicated, but since v^1 is closed, we can adjust the stages of all subterms of v^1. The case for the let reduction (ELet0) also looks complicated, but it is just an instance of System F-style redex (where type abstraction and type application are explicitly performed), and is shown to be type preserving. □

The calculus λ^C does not have the progress property, since the evaluation in λ^C sometimes gets stuck due to the failure of dynamic type checking. We can define a variant of λ^C in which stuck terms in λ^C evaluate to the constant `wrong`, and the evaluation propagates `wrong`. Then, the variant would enjoy the modified progress property in the following form: every closed, typable term at stage 0 is either a stage-0 value or makes progress in evaluation (possibly to `wrong`).

7 Examples and Discussion

We show a few programming examples of our calculus in this section. Whereas we give the formal account to the foundational calculus in the previous sections, we make it more practical by extending it with familiar data structures such as integers, and also typecase, which allows the dispatching on the type of a given value if its type is not fully specified.

7.1 Typecase in Program Generation

The syntax of the term is extended with a typecase term as follows:

$$\textbf{tcase}\ \ e_0\ \ \textbf{with}\ x_1 : \tau_1 \to e_1 \mid \cdots \mid x_n : \tau_n \to e_n\ \ \textbf{end}$$

It is abbreviated as $\textbf{tcase}\ \ e_0\ \ \textbf{with}\ \overline{x : \tau \to e}\ \ \textbf{end}$. We assume that the last case $x_n : \tau_n \to e_n$ is the default case, which is realized by setting $\tau_n = ?$ in our calculus.

To evaluate this term, we first evaluate e_0 to obtain a value v, then compute its type τ_0 (under the suitable typing context). If $\tau_0 \sim \tau_1$ holds, then the first case is selected and the above term one-step-reduces to e_1. Otherwise, we continue traversing the cases until we reach a successful case. (Since we assume that a typecase term has the default case, we will eventually succeed.)

We omit the formal typing rule, the translation rules and the evaluation rules for typecase in this paper, but they are easily formulated.

7.2 Stage Polymorphism, Partially

Below we show an example in the MetaOCaml-like notation:

```
let add :(?->?->?) a:? b:? =
    tcase a with
    | x:int ->
        tcase b with
        | y:int   -> x+y
        | y:<int> -> < x + ~y >
        | y:?     -> false
        end
    | x:<int> ->
        tcase b with
        | y:int   -> < ~a + b >
        | y:<int> -> < ~a + ~b >
        | y:?     -> false
        end
    | x:?         -> false
    end
```

The above function receives two arguments of type Any (?) and its return type is also Any. When called on two arguments, it dispatches on the types of the two arguments a and b: if both are integers, it adds them and returns the result. If both are codes of integers, it builds a code to add two integers. It also handles the case where one argument is an integer, but the other is a code of an integer, provided that an integer value can be lifted to a later stage. If any of the arguments is not an integer or a code of an integer, it returns false.

We can test the function with different types of arguments:

```
add <3+5> <7*2>  --> <(3+5)+(7*2)>
add (3+5) <7*2>  --> <8    +(7*2)>
add (3+5) (7*2)  --> 22
```

This function shows that gradual typing can be used to build a partially staged polymorphic code that can work on both integers and the code of integers. This simple function is useful to convert a more complicated non-staged program into a staged program automatically:

```
let foo:(t -> t -> t) a:t =
  add (mul a a) (add a (const 1))
```

Assuming that mul and const are similarly defined, the above function foo can be used to compute an integer, or to build a code that computes an integer.

In fact, we can make it recursive so that it works more than two stages (code of code of integers, and so on). Whereas this kind of programming using the type ? is more fragile than the calculi with the built-in mechanism for stage-polymorphism, we can still guarantee a certain safety property. For the above example, the type ? appears in the

[5] We can easily show that, if e is closed and $\rhd e \hookrightarrow^{e'}$, then e' is closed.

[6] This condition is necessary since we do not have a cast rule for a free type variable. Note that the Let' rule in λ^C makes type variables bound by $\forall$, so there is no loss of expressivity by this condition.

present stage computation only, hence program generation may fail, but if it succeeds, the generated code is statically typed.

7.3 Discussion

Let us discuss practical use cases of our calculus $\lambda^{G\circ}$.

One of the most promising applications of our language is to have gradual typing at the present stage (stage 0) only, which means that the stage s (for $s > 0$) employs static typing. Then, the cast operation is not invoked during the execution of generated programs, except for the meaningless cast $[\ ?\ ]e$. When generating codes, it is sometimes useful to be freed from the static type discipline, for instance, we generate code from input files that include the specification of types in the generated code. In such a case, we want to generate an efficient specialized function to manipulate the data whose type is given only as an input of the code generator. Although generic programming may solve such issues, it is helpful to write a dynamically typed program generator that generates statically typed programs. In our calculus, we can ensure that, if the source program contains stage-0 casts only, generated codes do not have to perform dynamic type checking at all, except for the spurious checking caused by $[\ ?\ ]e$.

The above discussion applies to the stage-n code to have gradual types, if our final code is generated at stage $n + 1$. In general, we can statically check in our calculus whether dynamic type checking is necessary at stage n.

It also makes sense to generate gradually typed code from gradually or statically typed program generators. Today, we see a growing interest in dynamically typed programming languages that allow some form of program generation, including Ruby and JavaScript/TypeScript. For instance, JavaScript (and TypeScript) has template literals which are equipped with the quasi-quotation (the same as brackets in MetaML) and anti-quotation (the same as splices) mechanisms. Unfortunately, it uses the string datatype to represent programs as codes, which provides no guarantee for syntactic correctness (well-scopedness) and well-typedness. Even worse, they cannot be nested; if we use nested quasi-quotations and anti-quotations, we would get ill-formed or ill-scoped programs as strings. As the program-generation features in dynamically typed languages tend to rely on the notorious programs-as-strings paradigm, reformulating them based on our work would be promising future work.

8 Related Work

Program generation, or metaprogramming in general, has been studied in Lisp and Scheme communities for years, which has had a great influence on other studies. Efforts have been made to guarantee desirable properties such as hygiene (keeping lexical scope) statically. [4, 5] However, to state static properties without a type system is hard, and most studies focus on dynamic checking for properties such as contracts. As far as the authors know, there is no work on combining dynamic typing and static typing for metaprogramming, and take advantage of both worlds.

TypeScript may be considered a language that combines both worlds.[7] In fact, TypeScript has a mechanism for generating a program as a string, and running it as a program by the eval operator. As is well known, the string representation has a serious problem: generated strings may represent a syntactically incorrect program, let alone a well-scoped or well-typed program. Our work may be considered a reformulation of TypeScript's metaprogramming feature with a solid foundation.

Taha applied the 'staged interpreter' technique to an interpreter whose object language is a Lisp-like dynamically typed language. [20] He discussed how to estimate the type of a term as much as possible, and gave a few optimizations based on it. Although his setting is rather different from ours, it is interesting to relate these two works.

Gradual typing has been intensively studied and extended in various ways in the last decade. Recent work by Igarashi et al. [7] reported a nice formulation in the presence of parametric polymorphism, and it is an interesting future work to extend our work towards more expressive type systems.

Type systems for staged programming languages tend to be rather complicated, if one desires to guarantee the safety properties statically as in the MetaML approach. [22, 23]. For instance, allowing open code manipulation and guaranteeing safety of the run term (closedness of its argument) have been a major topic, and a number of type systems have been proposed. Yet, we are not in a fully satisfactory state for this study, since practical staged programs need computational effects for efficiency reasons, that would complicate the type system further, leading to intractable type systems. We think that combining static and dynamic typing in a single system can be a clue for the problem, and our calculus $\lambda^{G\circ}$ is a first step for it.

9 Conclusion

We have designed $\lambda^{G\circ}$, a staged gradual typed calculus with ML-style let-polymorphism. It combines two fundamental calculi: one is Siek and Taha's gradual type system λ^{G} and the other is Davies' linear-time temporal logic λ°. We presented the calculus, a type system, a translation to a cast calculus λ^{C}, and the operational semantics of λ^{C}. Altogether, we can run terms in $\lambda^{G\circ}$ to dynamically check the typability of possibly open terms with the help of our refined evaluation rules.

We believe that the work presented in this paper constitutes a solid first step towards merging two paradigms seamlessly. Due to exploratory purposes, we combined the minimal calculi for both sides, yet, found a few interesting issues and solutions for them. It is left for future work to extend

[7] In practice, TypeScript programs of type Any are not typechecked at runtime, but it is doable to typecheck such programs at runtime.

our calculus to various directions, including set-theoretic types such as union and intersection [3], computational effects [9, 13, 24], runtime code generation including the safe run primitive [1, 21], analytical metaprogramming [10, 19], and parametric polymorphism [7].

Acknowledgments

We would like to thank anonymous reviewers for their constructive comments. The second author is supported in part by JSPS Grant-in-Aid for Scientific Research (B) 23K24819.

References

[1] Cristiano Calcagno and Eugenio Moggi. 2000. Multi-Stage Imperative Languages: A Conservative Extension Result. In *Semantics, Applications, and Implementation of Program Generation, International Workshop SAIG 2000, Montreal, Canada, September 20, 2000, Proceedings (Lecture Notes in Computer Science, Vol. 1924)*, Walid Taha (Ed.). Springer, 92–107. doi:10.1007/3-540-45350-4_9

[2] Cristiano Calcagno, Eugenio Moggi, and Walid Taha. 2004. ML-Like Inference for Classifiers. In *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 (Lecture Notes in Computer Science, Vol. 2986)*, David A. Schmidt (Ed.). Springer, 79–93. doi:10.1007/978-3-540-24725-8_7

[3] Giuseppe Castagna and Victor Lanvin. 2017. Gradual typing with union and intersection types. *Proc. ACM Program. Lang.* 1, ICFP (2017), 41:1–41:28. doi:10.1145/3110285

[4] William D. Clinger and Jonathan Rees. 1991. Macros That Work. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages, Orlando, Florida, USA, January 21-23, 1991*, David S. Wise (Ed.). ACM Press, 155–162. doi:10.1145/99583.99607

[5] William D. Clinger and Mitchell Wand. 2020. Hygienic macro technology. *Proc. ACM Program. Lang.* 4, HOPL (2020), 80:1–80:110. doi:10.1145/3386330

[6] Rowan Davies. 1996. A Temporal-Logic Approach to Binding-Time Analysis. In *Proceedings, 11th Annual IEEE Symposium on Logic in Computer Science, New Brunswick, New Jersey, USA, July 27-30, 1996*. IEEE Computer Society, 184–195. doi:10.1109/LICS.1996.561317

[7] Atsushi Igarashi, Shota Ozaki, Taro Sekiyama, and Yudai Tanabe. 2024. Space-Efficient Polymorphic Gradual Typing, Mostly Parametric. *Proc. ACM Program. Lang.* 8, PLDI (2024), 1585–1608. doi:10.1145/3656441

[8] Yuu Igarashi, Taro Sekiyama, and Atsushi Igarashi. 2017. On polymorphic gradual typing. *Proc. ACM Program. Lang.* 1, ICFP (2017), 40:1–40:29. doi:10.1145/3110284

[9] 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 2024, Pasadena, CA, USA, October 22, 2024*, Shigeru Chiba and Thomas Thüm (Eds.). ACM, 53–65. doi:10.1145/3689484.3690731 https://doi.org/10.1145/3689484.3690731.

[10] Junyoung Jang, Samuel Gélineau, Stefan Monnier, and Brigitte Pientka. 2022. Mœbius: metaprogramming using contextual types: the stage where system f can pattern match on itself. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–27. doi:10.1145/3498700

[11] Oleg Kiselyov. 2014. The design and implementation of BER MetaOCaml. In *International Symposium on Functional and Logic Programming*. Springer, 86–102.

[12] Oleg Kiselyov, Yukiyoshi Kameyama, and Yuto Sudo. 2016. Refined Environment Classifiers - Type- and Scope-Safe Code Generation with Mutable Cells. In *Programming Languages and Systems - 14th Asian Symposium, APLAS 2016, Hanoi, Vietnam, November 21-23, 2016, Proceedings*. 271–291. doi:10.1007/978-3-319-47958-3_15

[13] Oleg Kiselyov, Yukiyoshi Kameyama, and Yuto Sudo. 2016. Refined Environment Classifiers - Type- and Scope-Safe Code Generation with Mutable Cells. In *Programming Languages and Systems - 14th Asian Symposium, APLAS 2016, Hanoi, Vietnam, November 21-23, 2016, Proceedings (Lecture Notes in Computer Science, Vol. 10017)*, Atsushi Igarashi (Ed.). 271–291. doi:10.1007/978-3-319-47958-3_15 https://doi.org/10.1007/978-3-319-47958-3_15.

[14] Nico Lehmann and Éric Tanter. 2017. Gradual refinement types. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 775–788. doi:10.1145/3009837.3009856

[15] Aleksandar Nanevski and Frank Pfenning. 2005. Staged computation with names and necessity. *J. Funct. Program.* 15, 5 (2005), 893–939. doi:10.1017/S095679680500568X

[16] Jeremy G. Siek and Walid Taha. 2006. Gradual Typing for Functional Languages. In *Proceedings of the Scheme and Functional Programming Workshop*. University of Chicago, 81–92.

[17] Jeremy G. Siek and Walid Taha. 2007. Gradual Typing for Objects. In *ECOOP 2007 - Object-Oriented Programming, 21st European Conference, Berlin, Germany, July 30 - August 3, 2007, Proceedings (Lecture Notes in Computer Science, Vol. 4609)*, Erik Ernst (Ed.). Springer, 2–27. doi:10.1007/978-3-540-73589-2_2

[18] Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, and John Tang Boyland. 2015. Refined Criteria for Gradual Typing. In *1st Summit on Advances in Programming Languages, SNAPL 2015, May 3-6, 2015, Asilomar, California, USA (LIPIcs, Vol. 32)*, Thomas Ball, Rastislav Bodík, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morrisett (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 274–293. doi:10.4230/LIPICS.SNAPL.2015.274

[19] Nicolas Stucki, Jonathan Immanuel Brachthäuser, and Martin Odersky. 2021. Multi-stage programming with generative and analytical macros. In *GPCE '21: Concepts and Experiences, Chicago, IL, USA, October 17 - 18, 2021*, Eli Tilevich and Coen De Roover (Eds.). ACM, 110–122. doi:10.1145/3486609.3487203

[20] Walid Taha. 2007. A Gentle Introduction to Multi-stage Programming, Part II. In *Generative and Transformational Techniques in Software Engineering II, International Summer School, GTTSE 2007, Braga, Portugal, July 2-7, 2007. Revised Papers (Lecture Notes in Computer Science, Vol. 5235)*, Ralf Lämmel, Joost Visser, and João Saraiva (Eds.). Springer, 260–290. doi:10.1007/978-3-540-88643-3_6

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

[22] Walid Taha and Michael Florentin Nielsen. 2003. Environment classifiers. In *Conference Record of POPL 2003: The 30th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, New Orleans, Louisisana, USA, January 15-17, 2003*, Alex Aiken and Greg Morrisett (Eds.). ACM, 26–37. doi:10.1145/604131.604134

[23] Walid Taha and Tim Sheard. 2000. MetaML and multi-stage programming with explicit annotations. *Theor. Comput. Sci.* 248, 1-2 (2000), 211–242. doi:10.1016/S0304-3975(00)00053-0

[24] Edwin Westbrook, Mathias Ricken, Jun Inoue, Yilong Yao, Tamer Abdelatif, and Walid Taha. 2010. Mint: Java Multi-stage Programming Using Weak Separability. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation* (Toronto, Ontario, Canada) *(PLDI '10)*. ACM, New York, NY, USA, 400–411. doi:10.1145/1806596.1806642