Incremental Type-Checking for Type-Reflective Metaprograms

Weiyu Miao Jeremy Siek
University of Colorado at Boulder
{weiyu.miao, jeremy.siek}@colorado.edu

Abstract

Garcia introduces a calculus for type-reflective metaprogramming that provides much of the power and flexibility of C++ templates and solves many of its problems. However, one of the problems that remains is that the residual program is not type checked until after meta computation is complete. Ideally, one would like the type system of the metaprogram to also guarantee that the residual program will type check, as is the case in MetaML. However, in a language with type-reflective metaprogramming, type expressions in the residual program may be the result of meta computation, making the MetaML guarantee next to impossible to achieve.

In this paper we offer an approach to detecting errors earlier without sacrificing flexibility: we incrementally type check code fragments as they are created and spliced together during meta computation. The incremental type system is a variant of the gradual type system of Siek and Taha, in which we use type variables to represent type expressions that are not yet normalized and a new dynamic variation on existential types to represent residual code fragments. A type error in a code fragment is treated as a run-time error of the meta computation. We show that the incremental type checker can be implemented efficiently and we prove that if a well-typed metaprogram generates a residual program, then the residual program is also well-typed.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definition and Theory; D.3.4 [*Programming languages*]: Processors

General Terms Languages

Keywords metaprogramming, type systems, type reflection, gradual typing

1. Introduction

Metaprogramming is the writing of computer programs that generate or manipulate programs. Reflection, in the context of metaprogramming, is the ability to inspect and/or manipulate a program's metadata (such as types). We say that a language supports *reflective metaprogramming* if it supports both staged computation and reflection. This combination of features enables software developers to build libraries that are versatile and easy to use because the libraries can, during compilation, adapt to the contexts in which they are used.

GPCE'10, October 10–13, 2010, Eindhoven, The Netherlands.

γ type constants (e.g. **int**, **bool**)
x variables
c value constants

code language
$e ::= x \mid c \mid \lambda x : e^m.e \mid e\ e \mid \textbf{let}\ x = e\ \textbf{in}\ e \mid \sim e^m \mid \textbf{let meta}\ x = e^m\ \textbf{in}\ e \mid e^m\langle e^m \rangle \mid \textbf{if}\ e\ \textbf{then}\ e\ \textbf{else}\ e$

meta language
$e^m ::= x \mid c \mid c^m \mid \gamma \mid \lambda x : \tau^m.e^m \mid e^m\ e^m \mid \%e^m \mid \textbf{let}\ x = e^m\ \textbf{in}\ e^m \mid \textbf{if}\ e^m\ \textbf{then}\ e^m\ \textbf{else}\ e^m \mid \prec e \succ$
$c^m ::= \rightarrow \mid \gamma^? \mid \rightarrow^? \mid =_\tau \mid \textbf{dom} \mid \textbf{cod} \mid \textbf{typeof}$

meta types
$\tau^m ::= \gamma \mid \textbf{code} \mid \textbf{type} \mid \tau^m \rightarrow \tau^m$

Figure 1. Garcia's reflective metaprogramming calculus

Programming languages such as C++ [1, 2, 3], MetaML [4], and Template Haskell [5], enable metaprogramming by providing multiple stages of computation, where earlier stages can manipulate code for later stages [1]. Both C++ and Template Haskell provide reflection to some degree, whereas MetaML does not. The template feature of C++ enables the manipulation of types-as-data and provides a way to obtain the type of an expression. Template Haskell supports intensional analysis, a mechanism that can inspects the type and the internal structure of an expression in the compiler.

Garcia captured the fundamental capabilities of C++ in his reflective metaprogramming calculus [6, 7]. The syntax of his calculus is shown in Figure 1 [2]. The calculus consists of two interwoven languages, a *code language* to represent code that will be evaluated at run time, and a *meta language*, which is evaluated at compile time. The calculus includes the bracket $\prec e \succ$, escape $\sim e^m$, and lift $\%e$ constructs from MetaML [4, 8, 9], which were in turn inspired by the quasi-quote mechanism of Lisp [10].

Garcia's design preserves much of the power and flexibility of C++ templates while providing more support for reflection on meta data such as types. In his calculus, the meta language contains type primitives to analyze and manipulate types (i.e. **typeof** queries the type of a piece of code and **dom** accesses the domain of a function type). Consider the example of serializing arguments for a remote procedure call: we need to first serialize parameters into bitstrings. A simple serialization function is shown in Figure 2. We assume that the following primitive functions are available

int2str : **int** $\rightarrow$ bitstring
bool2str : **bool** $\rightarrow$ bitstring

[1] C++ and Template Haskell provide two stages whereas MetaML supports arbitrary numbers of stages.

[2] The calculus presented here differs in several minor ways from Garcia's surface language. We omit functional generators, which are a convenience feature and we include the splice and lift features from Garcia's kernel language.

```
/* serialize : (type list) → code → code */
let meta serialize = fix λrecur : (type list) → code → code.
  λtys : (type list). λinit : code.
    case tys of [ ] ⇒ init
      | ty :: res ⇒
          if ty =τ int then
            ≺λi : int.
                ∼ (recur res ≺(int2str i)+","+(∼ init)≻)≻
          else
            if ty =τ bool then
              ≺λb : bool.
                  ∼ (recur res ≺(bool2str b)+","+(∼ init)≻)≻
            else ≺"Parameter Type Error"≻;;
```

Example use of serialize and the resulting code:

```
∼ (serialize [bool, int] ≺""≻);;
⟹
λb : bool.
    λi : int.
        (bool2str b)+","+(int2str i)+","+""
```

Figure 2. Example use of reflective metaprogramming.

and the meta level part of Garcia's surface language is simply extended with list, pattern matching over list, and list type constructor, which can be borrowed from Standard ML.

The serialize function, at compile time, iterates over the types of the arguments of the function, dispatches to a type-specific print function corresponding to the type of each argument, and generates a C-like printf function that converts the parameter into a string and then concatenates the resulting strings. In the program, we utilizes the type reflection capability of Garcia's calculus: for instance, (ty $=_\tau$ **int**) is used to inspect if ty is equal to type **int**.

Type System Design for Metaprogramming In the design of type systems for metaprogramming languages, there is a tension between static type safety, that is, guaranteeing that well-typed metaprograms generate well-typed code, and expressiveness, enabling programmers to express their intent in a straightforward manner.

A modular type system can catch type errors at their place of origin but it restricts the set of metaprograms one can write. On the other hand, non-modular type systems (such as that of C++) accept a larger set of metaprograms, but some of those metaprograms may generate ill-typed code which is difficult to debug because the error messages point into the generated code instead of the source of the error in the metaprogram.

Ideally, one would like the type system of a metaprogramming language to guarantee that the residual program (object program) will type check, as is the case in MetaML. However, adding reflective capabilities to the language, like Garcia's calculus, makes the MetaML guarantee next to impossible to achieve. For MetaML, there is a phase distinction between types and meta evaluation, while for a reflective metaprogramming language, the type of a residual program may depend on the result of meta evaluation. For Garcia's calculus, there are two situations that cause this that we discuss in the following two paragraphs.

In Garcia's calculus, the type of a function's argument could be a meta-level type expression. Consider the following program, where the last line is a pair.

```
let meta id =
  λx : bool. ≺λy : if x then int else string. y≻
in
  let id_int = ∼ (id true) in
    let id_string = ∼ (id false) in
      (id_int 0, id_string "zero")
```

The body of meta function id contains a piece of residual code where the type of parameter y is a type expression. If the type expression is evaluated into **int**, then the residual code has type **int**→ **int**. If it is evaluated to **string**, then the residual has type **string**→ **string**.

Another reason that the type of residual code can be dependent on meta evaluation in Garcia's calculus is that all code fragments are given the same type, **code**, ignoring the type of value computed by the code fragment. Thus, meta-level if-expressions can produce code fragments with different types in their branches. Consider the following example that generates addition functions for different types.

```
/* generic_add : type → code */
let meta generic_add =
  λt : type.
    if (t =τ int) then ≺+int≻
    else if (t =τ float) then ≺+float≻
         else if (t =τ string) then ≺+string≻
              else if (t =τ list) then ≺append_list≻
                   else %(print_error "addition not supported")
in
  (∼ (generic_add int) 2 3)
```

At compile time, generic_add matches over the type of the parameter and selects a specific addition for that type. Please note that each branch has a different type and the return type of generic_add depends on the type parameter t.

MetaML does not provide computation over types and it records the type of a code fragment in the meta-level type of the code, i.e., **code** T instead of just **code**. Thus, applying a MetaML-style type system to a reflective metaprogramming language, like Garcia's calculus, would restrict the set of metaprograms we can write.

Garcia proposed a type system for his calculus that type-checks the meta language before meta evaluation and type-checks the object language after meta evaluation. Before meta evaluation, the type system guarantees that the residual programs inside a metaprogram are well formed, but not necessarily well typed. After meta evaluation, it type-checks the code generated by the metaprogram. For example, the following metaprogram

```
∼ ((λ x : code. ≺2 * (∼ x)≻) ≺true≻)
```

is accepted by Garcia's type system even though it generates the ill-typed code: (2 * **true**).

Although Garcia's type system is quite permissive, thereby providing expressiveness, the type system is not modular. That is, type checking a code fragment does not mean that all uses of the code fragment will be well typed. One symptom of this is that type error messages do not necessarily point to the source of the problem (in the metaprogram) but instead point to the generated code. Thus, programmers spend time tracking down the source of type errors. Consider the following metaprogram in which function f_0 produces ill-typed code.

```
let meta f0 =
  λt : type.
    let meta id =
      λx : bool. ≺λx : (if x then t else bool) . x≻
    in
      ≺∼ (id false) 1≻        // type error originates at here
in
  let meta f1 = ≺∼ (f0 int)≻ in
    let meta f2 = ≺... ∼ f1...≻ in
      ... // fn is deeply nested
        let meta fn = ≺...∼ fn−1...≻ in
          ∼ fn;;
```

Inside the definition of f_0, the expression ≺∼ (id **false**) 1≻ generates ill-typed code: (λx : **bool**. x) 1. So ideally we would like the

compiler to report an error message that accurately points to this expression inside the definition of f_0. Instead, Garcia's type system accepts the definition and the type error is caught after meta evaluation: after the metaprogram has generated a large amount of code of which the ill-typed code is a small part.

In this paper, we present a new type system for reflective metaprogramming (presented in Section 3). It type-checks both the meta and object fragments before meta evaluation. The type system is a variant of the gradual type system of Siek and Taha, in which we use type variables to represent type expressions that are not yet normalized and a new dynamic variation on existential types to represent residual code fragments. We use a constraint-based approach to provide an efficient implementation of our type system (Section 4) and give some simple examples on how our implementation works in Section 5. During meta evaluation, type information is added to the constraints when it becomes available. Thus, type-checking is interleaved with meta evaluation in the form of constraint solving. We discuss related work in Section 6 and conclude the paper in Section 7.

2. Incremental Type-Checking

Motivated by the problems discussed in the previous section, we created a new approach that type-checks metaprograms throughout meta evaluation, thereby catching type errors as soon as there is enough type information to discover them. Our idea is to *incrementally* type-check code fragments as they are produced and spliced together. Figure 3 shows a metaprogram e_0^m that, through n steps of evaluation, becomes the value v_n^m. The code fragments within the meta program are type-checked between each step of computation.

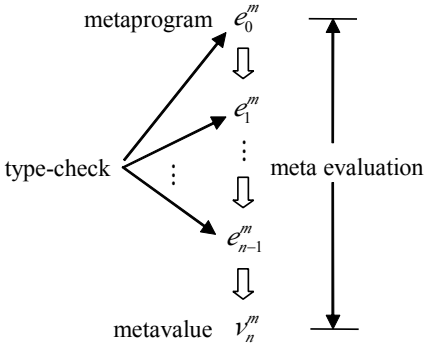

Figure 3. Incremental type-checking

Incremental type-checking can be applied to object-level functions whose argument types are meta-level type expressions. We take the following object expression as example.

$$\lambda \mathsf{x}_1 : \mathsf{e}_1^m.\ \lambda \mathsf{x}_2 : \mathsf{e}_2^m.\ \ldots\ \lambda \mathsf{x}_{n-1} : \mathsf{e}_{n-1}^m.\ \lambda \mathsf{x}_n : \mathsf{e}_n^m.\ \mathsf{e}$$

Suppose all the variables are annotated with type expressions that need to be evaluated. It is difficult to fully type-check the initial program before meta evaluation, but after some small steps of meta evaluation such that the type of x_1 is evaluated into some simple type (see the below code. Simple type represents those types that are fully evaluated and do not contain variables. The formal definition of simple type is in Figure 7),

$$\lambda \mathsf{x}_1 : \tau_1^s.\ \lambda \mathsf{x}_2 : \mathsf{e}_2^m.\ \ldots\ \lambda \mathsf{x}_{n-1} : \mathsf{e}_{n-1}^m.\ \lambda \mathsf{x}_n : \mathsf{e}_n^m.\ \mathsf{e}$$

we can then type-check the expression to discover if there is a type error caused by the simple type of x_1, that is τ_1^s. If there is no type error, then we meta-evaluate the expression more steps forward until the simple type of x_2 is available and then we can type-check it again. This continues until after all the type expressions are evaluated and the code eventually generated is definitely well-typed.

Incremental type-checking can also be applied to dynamically-typed meta-level if-expressions. We take the following program as example.

$$\begin{array}{l}\textbf{let meta } \mathsf{x}_1 = \textbf{if } \mathsf{e}_{11}^m \textbf{ then } \prec\mathsf{e}_{12}\succ \textbf{ else } \prec\mathsf{e}_{13}\succ \textbf{ in}\\ \quad \textbf{let meta } \mathsf{x}_2 = \textbf{if } \mathsf{e}_{21}^m \textbf{ then } \prec\mathsf{e}_{22}\succ \textbf{ else } \prec\mathsf{e}_{23}\succ \textbf{ in}\\ \qquad \ldots\\ \qquad\quad \textbf{let meta } \mathsf{x}_n = \textbf{if } \mathsf{e}_{n1}^m \textbf{ then } \prec\mathsf{e}_{n2}\succ \textbf{ else } \prec\mathsf{e}_{n3}\succ \textbf{ in}\\ \qquad\qquad \mathsf{e}\end{array}$$

Suppose for each of the meta-level if-expressions above, the type of the residual object code fragment in then branch could be different from that in else branch. Before meta evaluation it is difficult to fully type-check the entire program because the if-expressions are dynamically typed. But after some meta evaluation steps such that e_{11}^m is evaluated to some boolean value, the branch of the first meta-level if expression is selected, and the type of x_1 becomes available. We can then type-check the program to discover if there is a type error caused by the type of x_1. So, the entire program is incrementally type-checked as more of the if-expressions are evaluated.

Literally type checking the entire metaprogram at every step of evaluation would incur considerable cost. Instead, we derive a set of typing constraints (equalities between type expressions) from the program and then incrementally solve and update these constraints during meta evaluation (see Figure 4). As meta evaluation proceeds,

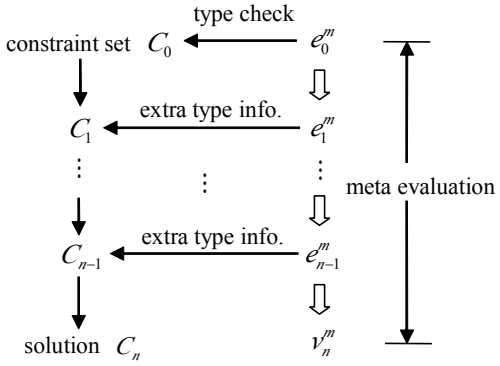

Figure 4. Constraint-based incremental type-checking

more type information is available, which can add to the set of constraints. Solving the constraint set (via unification) simplifies the set and also discovers any type conflicts, which would indicate a type error. After meta evaluation, if the constraint set is still satisfiable, then the generated piece of code is guaranteed to be well-typed.

3. Type System

We have illustrated a general picture of how incremental type-checking works for metaprograms. In practice, we need to consider how to carefully handle type expressions and meta-level polymorphism during type-checking so that our metaprogramming type system not only catches type errors as early as possible but also enables the writing of expressive metaprograms. As a result, we design a variant of the type system for gradual typing [11, 12], which satisfies all of our requirements for a reflective metaprogramming type system.

x		variables: meta-level or object-level variables
α, β		object type variables
c		value constants
$\mathcal{T}$		type primitives (e.g. **dom**, **cod**, $=_\tau$, $\gamma^?$, $\rightarrow^?$)
γ		type constants (e.g. **int**, **bool**)
simple types		
τ^s	$::=$	$\gamma \mid \tau^s \rightarrow \tau^s$
object types		
τ	$::=$	$x \mid \gamma \mid \alpha \mid \tau \rightarrow \tau$
object language		
e	$::=$	$x \mid c \mid \lambda x : e^m.\, e \mid e\, e \mid \sim e^m$
types for proper code		
τ^o	$::=$	$x \mid \gamma \mid \tau^o \rightarrow \tau^o$
proper code		
e^o	$::=$	$x \mid c \mid \lambda x : \tau^o.\, e^o \mid e^o\, e^o$
meta types		
τ^m	$::=$	$\gamma \mid \exists\overline{\alpha}.(\textbf{code}\ \tau) \mid \textbf{type} \mid \tau^m \rightarrow \tau^m$
meta language		
e^m	$::=$	$x \mid c \mid \mathcal{T} \mid \gamma \mid e^m \rightarrow e^m \mid \lambda x : \tau^m.\, e^m \mid e^m\, e^m \mid \textbf{typeof}\ e^m \mid \prec e \succ \mid \%e^m \mid \textbf{if}\ e^m\ \textbf{then}\ e^m\ \textbf{else}\ e^m$
meta values		
v^m	$::=$	$c \mid \gamma \mid v^m \rightarrow v^m \mid \lambda x : \tau^m.\, e^m \mid \prec e^o \succ$

Figure 5. Kernel calculus

The simplified kernel metaprogramming language syntax that we use is shown in Figure 5, and part of the syntax is borrowed from Garcia's calculus for type-reflective metaprogramming. Proper code, written as e^o, represents the object code generated by metaprograms after meta evaluation. For convenience, we integrate the type reflection operators appearing in Garcia's calculus into one category: type primitives, written as $\mathcal{T}$, each of which can be applied to a sequence of type expressions. Please note that in the meta types, **code** is modified into $\exists\overline{\alpha}.(\textbf{code}\ \tau)$, where τ means that it is a requirement to type-check a piece of object code thoroughly before meta evaluation in our type system, and existential quantification over the type variables appearing in τ means that if a piece of code is well-typed, then during meta evaluation, we can always find some actual types to substitute those quantified type variables in τ.

Gradual Typing. Gradual typing gives the programmer control over which portions of the program are statically checked based on the presence or absence of type annotations. It assigns the dynamic type, written ? for short, to any expressions (including variables) whose types are not available at compile time. For example, the following is a portion of the generic_add function already mentioned in Section 1 and suppose we have available functions $+_{int} : (\textbf{int} * \textbf{int}) \rightarrow \textbf{int}$ and $+_{float} : (\textbf{float} * \textbf{float}) \rightarrow \textbf{float}$,

```
let meta generic_add =
  λx : type.
    if x =τ int then ⟨code ((?∗?)→ ?)⟩≺+int≻
    else
      if x =τ float then ⟨code ((?∗?)→ ?)⟩≺+float≻
      else ...
```

In the above program, we use $\langle\textbf{code}\ ((?*?)\rightarrow ?)\rangle$ to explicitly type-cast the type of each branch into the same type that contains the dynamic type, and thus function generic_add has type $\textbf{type} \rightarrow \textbf{code}\ ((? * ?) \rightarrow ?)$. In gradual typing, the use of type equality is replaced by the type consistency relation, written $\sim$. The dynamic type is consistent with any ground types (or called type constants) but two different ground types are inconsistent with each other. Two function types are consistent if their domains and codomains are consistent, respectively.

We use existential types (written $\exists\alpha.\tau$, where α is an existentially quantified type variable that appears in type τ) to describe a piece of residual object code fragment if its type information is not complete (or unavailable) before meta evaluation. However, we do not use the standard parametric interpretation of existentials, but instead a non-parametric variation on existentials that generalizes the dynamic type: for parametric polymorphism, the existence of actual types (called *witness types*) are statically guaranteed (they are statically provided) for those existentially quantified type variables, but instead for our non-parametric version, the existence of witness types have to be dynamically checked, and if they exist then they are dynamically generated. In the implementation, an object type variable that is existentially quantified represents a piece of type information that is not available before meta evaluation, and it could be instantiated into some actual type during meta evaluation. For instance, the code fragment inside the following metaprogram has type $\exists\alpha.$ **code** $(\alpha \rightarrow \alpha)$, and when function id is applied to **true**, α is instantiated into **int** during meta evaluation.

```
let meta id = λx : bool. ≺λy : if x then int else bool. y≻ in
  ∼ (id true)
```

Existential types usually can capture more precise type information than the dynamic type. Take the previous function generic_add as example. According to the dynamic typing, the residual code in the branches could share the same type **code** $((? * ?) \rightarrow ?)$, which means the generated function could be any binary function, i.e. **code** $((\textbf{int} * \textbf{bool}) \rightarrow \textbf{string})$. However, in fact, function generic_add always generates a type-specific addition, which is a binary operation and should have a type that can be represented in a more precise form: $\exists\alpha.$ **code** $((\alpha * \alpha) \rightarrow \alpha)$. This form represents the set of binary operations, the subset of binary functions, thus precluding **code** $((\textbf{int} * \textbf{bool}) \rightarrow \textbf{string})$.

Type consistency $\boxed{\tau^m \sim \tau^m}$

$$\text{(C-Con)}\ \frac{}{\gamma \sim \gamma} \qquad \text{(C-Fun)}\ \frac{\tau_1^m \sim \tau_3^m \quad \tau_2^m \sim \tau_4^m}{\tau_1^m \rightarrow \tau_2^m \sim \tau_3^m \rightarrow \tau_4^m}$$

$$\text{(C-Typ)}\ \frac{}{\textbf{type} \sim \textbf{type}} \qquad \text{(C-Ext)}\ \frac{\phi_1(\tau_1) = \phi_2(\tau_2)}{\exists\overline{\alpha}.(\textbf{code}\ \tau_1) \sim \exists\overline{\beta}.(\textbf{code}\ \tau_2)}$$

Naive subtyping $\boxed{\tau^m <:_n \tau^m}$

$$\text{(S-Con)}\ \frac{}{\gamma <:_n \gamma} \qquad \text{(S-Fun)}\ \frac{\tau_1^m <:_n \tau_3^m \quad \tau_2^m <:_n \tau_4^m}{\tau_1^m \rightarrow \tau_2^m <:_n \tau_3^m \rightarrow \tau_4^m}$$

$$\text{(S-Typ)}\ \frac{}{\textbf{type} <:_n \textbf{type}} \qquad \text{(S-Ext)}\ \frac{\tau_1 = \phi(\tau_2)}{\exists\overline{\alpha}.(\textbf{code}\ \tau_1) <:_n \exists\overline{\beta}.(\textbf{code}\ \tau_2)}$$

Figure 6. Type consistency and naive subtyping relations for metatypes, where ϕ is a substitution of free object type variables for object types.

We define $[\tau/\alpha](\cdot)$ as the substitution of a free object type variable for an object type in a metatype, an object type, a meta expression, or an object expression. So, $[\overline{\tau}/\overline{\alpha}](\cdot)$, written ϕ for short, is the simultaneous substitution of free object type variables for object types. Figure 6 shows the type consistency and the *naive subtyping* [13] relations for the metatypes. For example, from rule C-Ext we know metatypes $\exists\alpha.$ **code** $(\alpha \rightarrow \textbf{int})$ and $\exists\alpha, \beta.$ **code** $(\beta \rightarrow \alpha)$ are consistent because we can find two substitutions $[\beta/\alpha]$ and $[\textbf{int}/\alpha]$ such that $[\beta/\alpha](\alpha \rightarrow \textbf{int}) = [\textbf{int}/\alpha](\beta \rightarrow \alpha)$.

Naive subtyping is covariant in the domain of function types (shown from rule S-Fun) instead of contravariant like ordinary

subtyping. Rule S-EXT defines the naive subtype relation of two existential metatypes. Suppose we have two different existential metatypes τ_1^m and τ_2^m, and if $\tau_1^m <:_n \tau_2^m$, then τ_1^m must be *more specific* than τ_2^m. For example, we know that $\exists\alpha.$ **code** $(\alpha \to \alpha$) is a naive subtype of $\exists\alpha, \beta.$ **code** $(\alpha \to \beta)$ because there is a substitution $[\alpha/\beta]$ such that $(\alpha \to \alpha) = [\alpha/\beta](\alpha \to \beta)$. Obviously, the first one is more specific than the second one because the first one refers to the set of unary operations whose domain and codomain must have the same type while the second refers to any kind of unary functions.

We say that τ^m is an *upper bound* of τ_1^m and τ_2^m if $\tau_1^m <:_n \tau^m$ and $\tau_2^m <:_n \tau^m$. Two metatypes do not necessarily have an upper bound. For example, **int** and **bool** do not have an upper bound. However, any two existential metatypes must have at least one upper bound, that is $\exists\alpha.$ **code** α. Suppose τ_1^m and τ_2^m have some upper bound(s), then we define $\tau_1^m \vee_n \tau_2^m$ as the *least upper bound* of τ_1^m and τ_2^m.

τ^m is $(\tau_1^m \vee_n \tau_2^m)$ iff

(*i*). $(\tau_1^m <:_n \tau^m) \wedge (\tau_2^m <:_n \tau^m)$, and

(*ii*). $\forall \tau_t^m.\ (\tau_1^m <:_n \tau_t^m) \wedge (\tau_2^m <:_n \tau_t^m) \Rightarrow (\tau^m <:_n \tau_t^m)$.

For example, the following

$\exists\alpha.$ **code** α,
$\exists\alpha, \beta.$ **code** $(\alpha \to \beta)$,
$\exists\alpha.$ **code** $(\alpha \to \alpha)$

are all the upper bounds of **code** $(\mathsf{int} \to \mathsf{int})$ and **code** $(\mathsf{bool} \to \mathsf{bool})$, but only $\exists\alpha.$ **code** $(\alpha \to \alpha)$ is the least upper bound of the two and it is also the most specific among all the upper bounds.

Figure 7 shows a selection of the typing rules for the object language and the meta language (we omit some typing rules that are conventional and they can be found in Garcia's paper [6]). For object-level functions, when the type annotation of a function parameter is not a simple type (not yet evaluated), we allow the parameter to take any type. Consider the following type rule, e^m is some type expression but not a simple type, that is $\nexists\tau^s.(\tau^s = e^m)$, therefore the type of x is allowed to be any object type τ_1 that allows the body e to type check.

$$\frac{\Gamma; \Psi \vdash e^m : \textbf{type} \quad \Gamma; \Psi \vdash \tau_1 \textbf{ wf} \quad \Gamma, x : \tau_1; \Psi \vdash e : \tau_2}{\Gamma; \Psi \vdash \lambda x : e^m.\ e : \tau_1 \to \tau_2}$$

Further along in the meta computation, when the type of the parameter becomes a simple type, the following rule applies instead.

$$\frac{\Gamma, x : \tau^s; \Psi \vdash e : \tau}{\Gamma; \Psi \vdash \lambda x : \tau^s.\ e : \tau^s \to \tau}$$

Typing rule M-CODE fully type checks the residual object expression inside the brackets and it introduces existential types as a variant of the gradual type for a piece of code by existentially quantifying the object type variables appearing in the type of the object expression. In typing rule M-CODE, function ftv collects all the free object type variables appearing in the type environments and returns a set of object type variables. Proposition $\overline{\alpha} \sharp\ ftv(\Gamma, \Psi)$ specifiies that the set of the object type variables appearing in $\overline{\alpha}$ is disjoint with the set of the free object type variables appearing in Γ and Ψ. When a piece of code fragment is escaped (see rule T-ESP), we instantiate the existential type by substituting those quantified type variables for some types that we guess or infer. In typing rule M-IF, meta-level if expression can be dynamically typed as it allows two branches to have different metatypes but the metatypes should have the least upper bound.

Let $\longrightarrow$ and $\longrightarrow^m$ be the reduction relations for meta evaluation of the object and the meta languages respectively. Let E be an evaluation context whose hole is in an object context (say object evaluation context for short) and E^m be an evaluation context

---

Object-level type environment $\quad \Gamma \quad ::= \quad \cdot \mid \Gamma, x : \tau \mid \Gamma, \alpha$

Meta-level type environment $\quad \Psi \quad ::= \quad \cdot \mid \Psi, x : \tau^m$

Well-formed object types $\quad \boxed{\Gamma; \Psi \vdash \tau \textbf{ wf}}$

$$\frac{x : \textbf{type} \in \Psi}{\Gamma; \Psi \vdash x \textbf{ wf}} \qquad \frac{}{\Gamma; \Psi \vdash \gamma \textbf{ wf}}$$

$$\frac{\alpha \in \Gamma}{\Gamma; \Psi \vdash \alpha \textbf{ wf}} \qquad \frac{\Gamma; \Psi \vdash \tau_1 \textbf{ wf} \quad \Gamma; \Psi \vdash \tau_2 \textbf{ wf}}{\Gamma; \Psi \vdash \tau_1 \to \tau_2 \textbf{ wf}}$$

Well-formed object type sequence $\quad \boxed{\Gamma; \Psi \vdash \overline{\tau} \textbf{ wf}}$

$$\frac{\overline{\tau} = \tau_1 ... \tau_n \quad \Gamma; \Psi \vdash \tau_1 \textbf{ wf} \quad ... \quad \Gamma; \Psi \vdash \tau_n \textbf{ wf}}{\Gamma; \Psi \vdash \overline{\tau} \textbf{ wf}}$$

Type system for the object language $\quad \boxed{\Gamma; \Psi \vdash e : \tau}$

$$\text{(T-ABS1)}\ \frac{\Gamma; \Psi \vdash e^m : \textbf{type} \quad \Gamma; \Psi \vdash \tau_1 \textbf{ wf} \quad \Gamma, x : \tau_1; \Psi \vdash e : \tau_2}{\Gamma; \Psi \vdash \lambda x : e^m.\ e : \tau_1 \to \tau_2}$$

$$\text{(T-ABS2)}\ \frac{\Gamma, x : \tau^s; \Psi \vdash e : \tau}{\Gamma; \Psi \vdash \lambda x : \tau^s.\ e : \tau^s \to \tau}$$

$$\text{(T-ESP)}\ \frac{\Gamma; \Psi \vdash e^m : \exists\overline{\alpha}.(\textbf{code}\ \tau_1) \quad \Gamma; \Psi \vdash \overline{\tau} \textbf{ wf}}{\Gamma; \Psi \vdash \sim e^m : [\overline{\tau}/\overline{\alpha}]\tau_1}$$

Type system for the meta language $\quad \boxed{\Gamma; \Psi \vdash e^m : \tau^m}$

$$\text{(M-TYPEOF)}\ \frac{\Gamma; \Psi \vdash e^m : \exists\overline{\alpha}.(\textbf{code}\ \tau)}{\Gamma; \Psi \vdash \textbf{typeof}\ e^m : \textbf{type}}$$

$$\text{(M-CODE)}\ \frac{\Gamma, \overline{\alpha}; \Psi \vdash e : \tau \quad \overline{\alpha} \sharp\ ftv(\Gamma; \Psi)}{\Gamma; \Psi \vdash \prec e \succ : \exists\overline{\alpha}.(\textbf{code}\ \tau)}$$

$$\text{(M-APP)}\ \frac{\Gamma; \Psi \vdash e_1^m : \tau_1^m \to \tau_3^m \quad \Gamma; \Psi \vdash e_2^m : \tau_2^m \quad \tau_1^m \sim \tau_2^m}{\Gamma; \Psi \vdash e_1^m\ e_2^m : \tau_3^m}$$

$$\text{(M-IF)}\ \frac{\Gamma; \Psi \vdash e_1^m : \textbf{bool} \quad \Gamma; \Psi \vdash e_2^m : \tau_2^m \quad \Gamma; \Psi \vdash e_3^m : \tau_3^m \quad \exists\tau^m.\ \tau^m = \tau_2^m \vee_n \tau_3^m}{\Gamma; \Psi \vdash \textbf{if}\ e_1^m\ \textbf{then}\ e_2^m\ \textbf{else}\ e_3^m : \tau^m}$$

Figure 7. Selected rules from the type system

Single-step meta evaluation for E $\quad \boxed{E[e] \longmapsto E[e']}$

$$\frac{e \longrightarrow e'}{E[e] \longmapsto E[e']}$$

Single-step meta evaluation for E^m $\quad \boxed{E^m[e^m] \longmapsto E^m[e^{m\prime}] \text{ or } \textbf{error}}$

$$\frac{e^m \longrightarrow^m e^{m\prime} \quad \Gamma; \Psi \vdash E^m[e^{m\prime}] : \tau^m}{E^m[e^m] \longmapsto E^m[e^{m\prime}]} \qquad \frac{e^m \longrightarrow^m e^{m\prime} \quad \Gamma; \Psi \nvdash E^m[e^{m\prime}] : \tau^m}{E^m[e^m] \longmapsto \textbf{error}}$$

$$\frac{e^m \longrightarrow^m e^{m\prime} \quad \Gamma; \Psi \vdash E^m[e^{m\prime}] : \tau}{E^m[e^m] \longmapsto E^m[e^{m\prime}]} \qquad \frac{e^m \longrightarrow^m e^{m\prime} \quad \Gamma; \Psi \nvdash E^m[e^{m\prime}] : \tau}{E^m[e^m] \longmapsto \textbf{error}}$$

Figure 8. Single-step meta evaluation

whose hole is in a meta context (say meta evaluation context for short). Then single-step evaluation is defined in Figure 8. We omit all the evaluation contexts and the reduction rules because they are presented in Garcia's paper [6]. The only difference is the symbol used to represent meta evaluation context: in Garcia's paper, meta evaluation context is E^s while ours is E^m. We use object evaluation context to explain how evaluation context works and meta evaluation context adopts the same mechanism. An object evaluation context E can be turned into either an object language expression (i.e. $\lambda x : \tau^s.\,\square$) or a meta language expression (i.e. $\prec\!\square\!\succ$) by plugging an object expression e into its hole, expressed with the notation $E[e]$. Evaluation contexts are used specifically to define program evaluation steps, for instance:

$$\frac{e \longrightarrow e'}{E[e] \longmapsto E[e']}$$

and the reduction relation $e \longrightarrow e'$ specifies how computations are performed. Meta computation results in $\mathbf{error}$ as soon as a type error is detected.

Given the formalization of the incremental type system for a type-reflective metaprogramming language, we next reason about its behavior, in particular to ensure that the type system is sound.

Theorem 1 (Progress)**.**

1. *If e is closed and $\Gamma; \Psi \vdash e : \tau$, then e is a proper object code, or there is some e' such that $e \longmapsto e'$, or $e \longmapsto \mathbf{error}$.*
2. *If e^m is closed and $\Gamma; \Psi \vdash e^m : \tau^m$, then e^m is a meta value, or there is some $e^{m\prime}$ such that $e^m \longmapsto e^{m\prime}$, or $e^m \longmapsto \mathbf{error}$.*

Proof. Straightforward induction on typing derivations. □

Theorem 2 (Preservation)**.**

1. *If $\Gamma; \Psi \vdash e : \tau$ and $e \longmapsto e'$, then $\Gamma; \Psi \vdash e' : \tau'$.*
2. *If $\Gamma; \Psi \vdash e^m : \tau^m$ and $e^m \longmapsto e^{m\prime}$, then $\Gamma; \Psi \vdash e^{m\prime} : \tau^{m\prime}$.*

Proof. Straightforward induction on typing derivations. □

Theorem 3 (Type Safety)**.**

1. *If $\Gamma; \Psi \vdash e : \tau$ and $e \longmapsto^* e'$, then $\Gamma; \Psi \vdash e' : \tau'$, and e' is a proper object code, or $\exists e''.e' \longmapsto e''$, or $e' \longmapsto \mathbf{error}$.*
2. *If $\Gamma; \Psi \vdash e^m : \tau^m$ and $e^m \longmapsto^* e^{m\prime}$, then $\Gamma; \Psi \vdash e^{m\prime} : \tau^{m\prime}$, and $e^{m\prime}$ is a meta value, or $\exists e^{m\prime\prime}.e^{m\prime} \longmapsto e^{m\prime\prime}$, or $e^{m\prime} \longmapsto \mathbf{error}$.*

Proof. Using Progress and Preservation. □

4. Implementation

In detail, our implementation of type-checking type-reflective metaprograms is *unification-based* incremental type-checking. The compiler first automatically inserts fresh non-indexed unification variables and labels into the initial program. Next, it generates a set of type constraints (type equalities) from the initial program during type-checking. We normalize the constraints using unification. If successful, we begin meta computation, and meanwhile update and unify the constraints when more type information is available. Figure 9 shows the language syntax for the implementation of unification-based incremental type-checking (please note that we omit the parts that overlap with those in the language syntax presented in Figure 5).

Unification variables, written κ for short, represent metatype unification variables or object type unification variables, and also represent *non-indexed* unification variables or *indexed* unification variables. We distinguish indexed unification variables from non-indexed ones to deal with the intricacy of dynamic typing, which

ℓ		labels
α, β, δ		(meta or object) non-indexed unification variables
$\alpha_n, \beta_n, \delta_n$		(meta or object) indexed unification variables
κ		unification variables, indexed or non-indexed
object types		
τ	$::=$	$x \mid \gamma \mid \kappa \mid \tau \to \tau$
object language		
e	$::=$	$\lambda x : (e^m)^\kappa.\, e \mid (\sim e^m)^{\overline{\ell}} \mid \cdots$
meta types		
τ^m	$::=$	$\exists\overline{\kappa}.(\mathbf{code}\ \tau) \mid \cdots$
meta types for unification		
σ^m	$::=$	$\kappa \mid \tau^m$
meta language		
e^m	$::=$	$(\mathbf{if}\ e^m\ \mathbf{then}\ (e^m)^{\kappa_1}\ \mathbf{else}\ (e^m)^{\kappa_2})^{\kappa_3} \mid ((e^m)^{\kappa_1})^{\kappa_2} \mid \cdots$
type constraint		
$cons$	$::=$	$\tau = \tau \mid \sigma^m = \sigma^m$
type constraint set		
C	$::=$	a set of type constraints

Figure 9. Selected kernel calculus for unification-based incremental type-checking

we will explain in more detail in the first example in the later section. The equality relation ($\bowtie$) of unification variables is defined in the following:

$$\frac{}{\alpha \bowtie \alpha} \qquad \frac{\textit{for any } n}{\alpha \bowtie \alpha_n} \qquad \frac{m = n}{\alpha_m \bowtie \alpha_n}$$

$$\frac{\kappa \bowtie \kappa'}{\kappa' \bowtie \kappa} \qquad \frac{\kappa \bowtie \kappa' \quad \kappa' \bowtie \kappa''}{\kappa \bowtie \kappa''} \qquad \frac{}{\nexists x.(\kappa \bowtie x)}$$

We define $\{\tau/\kappa\}(\cdot)$ as object type unification variable substitution: substituting κ for τ if κ is free in $\cdot$. The substitution can be applied to meta expressions, object expressions, metatypes, and a set of type constraints. The core part of the definition is as follows:

$$\{\tau/\kappa\}(\alpha) = \begin{cases} \tau, & \text{if } \kappa = \alpha \\ \kappa, & \text{otherwise} \end{cases}$$

$$\{\tau/\kappa\}(\alpha_n) = \begin{cases} \tau, & \text{if } \kappa \bowtie \alpha_n \\ \kappa, & \text{otherwise} \end{cases}$$

$$\{\tau/\kappa\}(\exists\overline{\kappa'}.(\mathbf{code}\ \tau)) = \begin{cases} \exists\overline{\kappa'}.(\mathbf{code}\ \tau), & \text{if } \kappa \text{ is in } \overline{\kappa'} \\ \exists\overline{\kappa'}.(\mathbf{code}\ \{\tau/\kappa\}\tau), & \text{otherwise} \end{cases}$$

So, the simultaneous substitution of object type unification variables for object types, written θ for short, is defined as:

$$\{\overline{\tau}/\overline{\kappa}\}(\cdot) \equiv \{\tau_1/\kappa_1\}...\{\tau_n/\kappa_n\}(\cdot)$$

where n is the length of the sequence of object types to be substituted. Both metatype unification variable substitution, written $\{\tau^m/\kappa\}(\cdot)$, and the simultaneous substitution of metatype unification variables, written $\{\overline{\tau^m}/\overline{\kappa}\}(\cdot)$, adopt the same mechanism.

From the Surface Calculus to the Kernel. The surface calculus is close to Garcia's calculus for the surface language. It is not annotated with any unification variables and labels. When the surface language is translated to (written $\rightsquigarrow$) the kernel, we annotate it with fresh non-indexed unification variables and labels. See the following rules, where the expressions with subscript s represent the expressions in the surface language.

$$\frac{e_s^m \rightsquigarrow e^m \quad e_s \rightsquigarrow e \quad \alpha \text{ is fresh}}{\lambda x : e_s^m.\, e_s \rightsquigarrow \lambda x : (e^m)^\alpha.\, e} \qquad \frac{e_s^m \rightsquigarrow e^m \quad \ell \text{ is fresh}}{\sim e_s^m \rightsquigarrow (\sim e^m)^{[\ell]}}$$

$$\frac{e_s^m \rightsquigarrow e^m \quad e_s^{m\prime} \rightsquigarrow e^{m\prime} \quad e_s^{m\prime\prime} \rightsquigarrow e^{m\prime\prime} \quad \alpha, \beta, \delta \text{ are fresh}}{\mathbf{if}\ e_s^m\ \mathbf{then}\ e_s^{m\prime}\ \mathbf{else}\ e_s^{m\prime\prime} \rightsquigarrow (\mathbf{if}\ e^m\ \mathbf{then}\ (e^{m\prime})^\alpha\ \mathbf{else}\ (e^{m\prime\prime})^\beta)^\delta}$$

Please note that each non-indexed unification variable and each label that initially appears in a kernel metaprogram is unique: each unification variable used to bind the type of a function argument is unique, each unification variable used in a meta-level if-expression is unique, and each label initially attached to a code splice is also unique. In the surface language, there are no existential types for object code fragments. However, the metatypes include both **code** and **code** τ, and we enable programmers to choose whether to type a piece of object code fragment in a more specific form or not. For instance, a programmer can type $\prec 1 \succ$ to be either **code** or **code int**, but code fragment $\prec \lambda \mathsf{x} : \mathsf{e}_s^m.\ \mathsf{x} \succ$ (where e_s^m is not some simple type) has to be typed with **code** because it is dynamically typed. When the surface language is translated into the kernel, **code** is translated into $\exists \alpha.\ (\textbf{code}\ \alpha)$ where α is fresh.

Object-level type environment	Γ	::=	$\cdot \mid \Gamma, x : \tau$
Meta-level type environment	Ψ	::=	$\cdot \mid \Psi, x : \tau^m$
Unification variable renaming	$\mathcal{R}$	::=	$\cdot \mid \mathcal{R}, \kappa \mapsto \alpha_n$
Variable renaming binding	Σ	::=	$\cdot \mid \Sigma, \ell \mapsto \mathcal{R}$

Constraint generation for the object language $\boxed{\Gamma; \Psi \vdash e : \tau \mid C; \Sigma}$

$$\text{(T-Abs1)}\ \frac{\nexists \tau^s.(\tau^s = e^m) \quad \Gamma; \Psi \vdash e^m : \textbf{type} \mid C_1; \Sigma_1 \quad \Gamma, x : \alpha; \Psi \vdash e : \tau \mid C_2; \Sigma_2}{\Gamma; \Psi \vdash \lambda x : (e^m)^\alpha.\ e : \alpha \to \tau \mid C_1 \cup C_2; \Sigma_1, \Sigma_2}$$

$$\text{(T-Abs2)}\ \frac{\Gamma, x : \tau^s; \Psi \vdash e : \tau \mid C; \Sigma}{\Gamma; \Psi \vdash \lambda x : \tau^s.\ e : \tau^s \to \tau \mid C; \Sigma}$$

$$\text{(T-Esp)}\ \frac{\Gamma; \Psi \vdash e^m : \exists \overline{\kappa}.(\textbf{code}\ \tau) \mid C; \Sigma \quad \overline{\alpha_n} = \mathsf{gen\_vars}(\overline{\kappa}) \quad C' = \{\overline{\alpha_n}/\overline{\kappa}\}C \quad \Sigma' = \Sigma, \ell \mapsto \overline{\kappa \mapsto \alpha_n}}{\Gamma; \Psi \vdash\ (\sim e^m)^{[\ell]} : \{\overline{\alpha_n}/\overline{\kappa}\}\tau \mid C'; \Sigma'}$$

Constraint generation for the meta language $\boxed{\Gamma; \Psi \vdash e^m : \tau^m \mid C; \Sigma}$

$$\text{(M-Typeof)}\ \frac{\Gamma; \Psi \vdash e^m : \exists \overline{\kappa}.(\textbf{code}\ \tau) \mid C; \Sigma}{\Gamma; \Psi \vdash \textbf{typeof}\ e^m : \textbf{type} \mid C; \Sigma}$$

$$\text{(M-Code)}\ \frac{\Gamma; \Psi \vdash e : \tau \mid C; \Sigma \quad \overline{\kappa} = uv(\tau) \cup uv(C)}{\Gamma; \Psi \vdash \prec e \succ : \exists \overline{\kappa}.(\textbf{code}\ \tau) \mid C; \Sigma}$$

$$\text{(M-App)}\ \frac{\Gamma; \Psi \vdash e_1^m : \tau_1^m \to \tau_3^m \mid C_1; \Sigma_1 \quad \Gamma; \Psi \vdash e_2^m : \tau_2^m \mid C_2; \Sigma_2 \quad C = C_1 \cup C_2 \cup \{\tau_1^m = \tau_2^m\}}{\Gamma; \Psi \vdash e_1^m\ e_2^m : \tau_3^m \mid C; \Sigma_1, \Sigma_2}$$

$$\text{(M-If)}\ \frac{\begin{array}{c}\Gamma; \Psi \vdash e_1^m : \textbf{bool} \mid C_1; \Sigma_1 \quad \Gamma; \Psi \vdash e_2^m : \tau_2^m \mid C_2; \Sigma_2 \quad \Gamma; \Psi \vdash e_3^m : \tau_3^m \mid C_3; \Sigma_3 \\ \tau^m = \mathcal{UB}^m(\tau_2^m, \tau_3^m) \quad \Sigma = \Sigma_1, \Sigma_2, \Sigma_3 \\ C_4 = \{\alpha = \tau_2^m\} \cup \{\beta = \tau_3^m\} \cup \{\delta = \tau^m\} \\ C = C_1 \cup C_2 \cup C_3 \cup C_4\end{array}}{\Gamma; \Psi \vdash (\textbf{if}\ e_1^m\ \textbf{then}\ (e_2^m)^\alpha\ \textbf{else}\ (e_3^m)^\beta)^\delta : \tau^m \mid C; \Sigma}$$

Figure 10. Selected constraint generation rules

Constraint generation version of the type system is defined in Figure 10 (we omit some of the constraint generation rules). The typing judgment $\Gamma; \Psi \vdash e : \tau \mid C; \Sigma$ is for the object language and the typing judgement $\Gamma; \Psi \vdash e^m : \tau^m \mid C; \Sigma$ is for the meta language. While we type-check an expression, we also generate a set of type constraints C and a set of mappings from labels to sets of unification variable renames Σ. In the figure, uv is the function that collects all the unification variables appearing in an object type, a metatype, or a set of constraints. Function $\mathsf{gen\_vars}(\overline{\kappa})$ generates a sequence of fresh unification variables (but not some arbitrary fresh unification variables) for $\overline{\kappa}$ and it is defined as follows:

$$\begin{array}{l}\mathsf{gen\_vars}([]) = [] \\ \mathsf{gen\_vars}(\alpha, \overline{\kappa'}) = (\alpha_n, \mathsf{gen\_vars}(\overline{\kappa'}))\ \textbf{where}\ n\ \text{is fresh} \\ \mathsf{gen\_vars}(\alpha_n, \overline{\kappa'}) = (\alpha_{n'}, \mathsf{gen\_vars}(\overline{\kappa'}))\ \textbf{where}\ n'\ \text{is fresh}\end{array}$$

For rule T-Abs1, we learn that in a term abstraction, if a variable is typed by some complex type expression, then we use a fresh unification variable α as its replacement. So, when type-checking the body, the compiler will generate a set of constraints containing those fresh unification variables. In contrast, if all variables are simply-typed (see rule T-Abs2), we instead use the type-checking approach for simply-typed lambda calculus that allows us to produce more precise typing. In rule M-If, we use function $\mathcal{UB}^m$ to calculate a upper bound of two metatypes if they have one, but the upper bound is not necessarily the least upper bound (we have not found a good algorithm to calculate the least upper bound of two metatypes). Following is the definition of function $\mathcal{UB}^m$ and the idea is to perform unification for metatypes and antiunification [14] for object types. When running into antiunification, it returns a fresh unification variable for two types that cannot be unified further.

$$\begin{array}{l}\mathcal{UB}^m(\gamma,\gamma) = \gamma \\ \mathcal{UB}^m(\textbf{type},\textbf{type}) = \textbf{type} \\ \mathcal{UB}^m(\tau_1^m \to \tau_2^m, \tau_3^m \to \tau_4^m) = \\ \quad \mathsf{T} := \mathcal{UB}^m(\tau_1^m, \tau_3^m);\ \mathsf{S} := \mathcal{UB}^m(\tau_2^m, \tau_4^m); \\ \quad \textbf{if}\ \mathsf{T} = \mathsf{fail}\ \textbf{or}\ \mathsf{S} = \mathsf{fail} \\ \quad\quad \textbf{return}\ \mathsf{fail} \\ \quad \textbf{return}\ \mathsf{T} \to \mathsf{S} \\ \mathcal{UB}^m(\exists \overline{\kappa_s}.(\textbf{code}\ \tau_1), \exists \overline{\kappa_t}.(\textbf{code}\ \tau_2)) = \\ \quad (\tau,\ \overline{\kappa}) := \mathcal{UB}^o(\tau_1,\tau_2);\ \textbf{return}\ \exists \overline{\kappa}.(\textbf{code}\ \tau) \\ \mathcal{UB}^m(\_,\_) = \mathsf{fail}\end{array}$$

$$\begin{array}{l}\mathcal{UB}^o(\mathsf{x},\mathsf{x}) = (\mathsf{x},[]) \\ \mathcal{UB}^o(\gamma,\gamma) = (\gamma,[]) \\ \mathcal{UB}^o(\tau_1 \to \tau_2, \tau_3 \to \tau_4) = \\ \quad (\tau_5, \overline{\kappa_s}) := \mathcal{UB}^o(\tau_1,\tau_3);\ (\tau_6, \overline{\kappa_t}) := \mathcal{UB}^o(\tau_2,\tau_4); \\ \quad \textbf{return}\ (\tau_5 \to \tau_6, (\overline{\kappa_s}, \overline{\kappa_t})) \\ \mathcal{UB}^o(\_,\_) = \\ \quad \mathsf{generate\ fresh}\ \alpha;\ \textbf{return}\ (\alpha,\ [\alpha])\end{array}$$

The selected evaluation contexts and reduction rules are presented in Figure 11. Because typing information is required for the *typeof* operation, we design the evaluation contexts that are accompanied by object-level typing environment Γ. They are E_Γ and E_Γ^m, which are mutually recursive. The typing environment extends when evaluating under a term abstraction. In Figure 11, we use some helper functions, which we will explain next. Function $\{v^m/x\}_\Sigma(\cdot)$ substitutes free meta variable x appearing in a meta or an object expression for meta value v^m. In the following, we give the definition of meta variable substitution in a code splice, which shows that we must rename the meta variables in the substituter:

$$\begin{array}{l}\{v^m/x\}_\Sigma((\sim e^m)^{\overline{\ell}}) = \\ \quad v^{m\prime} := \mathsf{rename}^m(\Sigma, \overline{\ell}, v^m);\ \textbf{return}\ (\sim \{v^{m\prime}/x\}_\Sigma(e^m))^{\overline{\ell}}\end{array}$$

Function $\mathsf{mapping}(\Sigma, \overline{\ell}, \kappa)$ substitutes unification variable κ for the mapped unification variable if κ has mapping(s) in Σ. Sometimes, the function performs a sequence of substitutions. Suppose $\overline{\ell}$ is ℓ_1, ℓ_2 and in mapping environment Σ, ℓ_1 has mapping $\kappa \mapsto \kappa'$ and ℓ_2 has mapping $\kappa' \mapsto \kappa''$. So, κ is firstly mapped to κ' and finally to κ''. Function $\mathsf{rename}^m(\Sigma, \overline{\ell}, e^m)$ renames the unification variables in meta expression e^m and function $\mathsf{rename}^o(\Sigma, \overline{\ell}, e)$ renames the unification variables in object expression e. The following shows part of the definitions of these functions.

$\mathsf{mapping}(\Sigma,[],\kappa) = \kappa$

$\mathsf{mapping}(\Sigma,(\ell,\overline{\ell'}),\kappa) = \mathsf{mapping}(\Sigma,\overline{\ell'},\kappa')$
$\quad$ **where** $\kappa' = \begin{cases} \kappa', & \ell \mapsto \mathcal{R} \in \Sigma \ \wedge \ \kappa \mapsto \kappa' \in \mathcal{R}; \\ \kappa, & \text{otherwise.} \end{cases}$

$\mathsf{rename}^m(\Sigma,\overline{\ell},\prec e \succ) =$
$\quad e' := \mathsf{rename}^o(\Sigma,\overline{\ell},e)$; **return** $\prec e' \succ$
$\mathsf{rename}^m(\Sigma,\overline{\ell},(\textbf{if } e_1^m \textbf{ then } (e_2^m)^{\kappa_1} \textbf{ else } (e_3^m)^{\kappa_2})^{\kappa_3}) =$
$\quad e_1^{m\prime} := \mathsf{rename}^m(\Sigma,\overline{\ell},e_1^m)$; $\kappa_1{}' := \mathsf{mapping}(\Sigma,\overline{\ell},\kappa_1)$;
$\quad e_2^{m\prime} := \mathsf{rename}^m(\Sigma,\overline{\ell},e_2^m)$; $\kappa_2{}' := \mathsf{mapping}(\Sigma,\overline{\ell},\kappa_2)$;
$\quad e_3^{m\prime} := \mathsf{rename}^m(\Sigma,\overline{\ell},e_3^m)$; $\kappa_3{}' := \mathsf{mapping}(\Sigma,\overline{\ell},\kappa_3)$;
$\quad$ **return** $(\textbf{if } e_1^{m\prime} \textbf{ then } (e_2^{m\prime})^{\kappa_1{}'} \textbf{ else } (e_3^{m\prime})^{\kappa_2{}'})^{\kappa_3{}'}$

$\mathsf{rename}^o(\Sigma,\overline{\ell},\lambda x : (e^m)^\kappa.e) =$
$\quad e^{m\prime} := \mathsf{rename}^m(\Sigma,\overline{\ell},e^m)$; $e' := \mathsf{rename}^o(\Sigma,\overline{\ell},e)$;
$\quad \kappa' := \mathsf{mapping}(\Sigma,\overline{\ell},\kappa)$; **return** $\lambda x : (e^{m\prime})^{\kappa'}.e'$
$\mathsf{rename}^o(\Sigma,\overline{\ell},(\sim e^m)^{\overline{\ell'}}) =$
$\quad e^{m\prime} := \mathsf{rename}^m(\Sigma,\overline{\ell},e^m)$; **return** $(\sim e^{m\prime})^{\overline{\ell'},\overline{\ell}}$

5. Examples

In this section, we provide some simple examples to show how our unification-based implementation works and moreover to show the implementation can report type errors earlier with more precise error messages, that is pointing to the place where type error originates. We suppose our kernel calculus is extended with meta-level let binding and object-level pair constructor.

Example One. In many cases, the type of an object code fragment depends on the environment in which it is used, i.e. the code fragment in λx : **type**. $\prec\lambda$y : x. y$\succ$. We can learn it from our constraint generation rule for code escape, which generates a local constraint set for each code splice by renaming the unification variables into fresh ones. However, consider the following metaprogram, where the type of f is independent from its use and after escaped, it cannot be applied to an integer.

let meta f = $\prec\lambda$x : (**if true then bool else int**)$^\alpha$. x$\succ$ **in**
... // more let bindings
((∼ f)$^{[\ell_1]}$ **false**, (∼ f)$^{[\ell_2]}$ 0)

For Garcia's type system, the type error is caught after type-checking the generated code but our type system can catch it earlier. According to rule M-CODE in Figure 10, f has type $\exists\alpha$. **code** $(\alpha \to \alpha)$. Suppose α is renamed into α_1 at the first code splice and renamed into α_2 at the second. After type-checking, we have the type constraint set which contains $\alpha_1 = \textbf{bool}$ and $\alpha_2 = \textbf{int}$. During meta evaluation, when the type of x is evaluated into type constant **bool**, we unify the constraint set after it is extended with $\alpha = \textbf{bool}$. Suppose we use the substitution-based unification. Because substituting α for **bool** also substitutes α_ns for any n, we get the type conflict, $\textbf{bool} = \textbf{int}$, and the meta evaluation terminates (see rule EM-TC-FAIL). The error message points to the type of x, the type expression attached by α, and could say that the type of x is evaluated into **bool** but used as **int**.

Example Two. Let us revisit the example presented in section one.

let meta f_0 =
λt : **type**.
let meta id =
λx : **bool**. $\prec\lambda$x : (**if** x **then** t **else bool**)$^\alpha$. x$\succ$
in
$\prec(\sim$ (id **false**))$^{[\ell_1]}$ 1$\succ$
in

Evaluation context with meta language hole

$$E_\Gamma^m ::= \square_\emptyset^m \mid E_\Gamma^m[\textbf{typeof } \square]_\Gamma \mid E_\Gamma[\lambda x : (\square)^\kappa.e]_\Gamma \mid E_\Gamma[\sim \square] \mid \cdots$$

Evaluation context with object language hole

$$E_\Gamma ::= \square_\emptyset \mid E_\Gamma^m[\prec\square\succ]_\Gamma \mid E_\Gamma[\lambda x : \gamma.\ \square]_{\Gamma,x:\gamma} \mid \cdots$$

Meta reduction rules $\quad \boxed{\Gamma;\Sigma \vdash e^m \mid C \longrightarrow^m e^{m\prime} \text{ or } \mathbf{error} \mid C'}$

$$(\textsc{Typeof})\ \frac{\Gamma \vdash v^m : \textbf{code}\ \tau^s \mid C;\Sigma}{\Gamma;\Sigma \vdash \textbf{typeof}\ v^m \mid C \longrightarrow^m \tau^s \mid C}$$

$$(\textsc{Em-App})\ \Gamma;\Sigma \vdash (\lambda x : \tau^m.e^m)\ v^m \mid C \longrightarrow^m \{v^m/x\}_\Sigma(e^m) \mid C$$

$$(\textsc{Em-TCoerce})\ \frac{\text{unify}\ (C \cup \{\kappa_1 = \kappa_2\}) = C'}{\Gamma;\Sigma \vdash ((e^m)^{\kappa_1})^{\kappa_2} \mid C \longrightarrow^m e^m \mid C'}$$

$$(\textsc{Em-TC-Fail})\ \frac{\text{unify}\ (C \cup \{\kappa_1 = \kappa_2\}) = \text{fail}}{\Gamma;\Sigma \vdash ((e^m)^{\kappa_1})^{\kappa_2} \mid C \longrightarrow^m \mathbf{error} \mid C}$$

Object reduction rules $\quad \boxed{\Gamma;\Sigma \vdash e \mid C \longrightarrow e' \text{ or } \mathbf{error} \mid C'}$

$$(\textsc{Ec-Abs})\ \frac{\text{unify}\ (C \cup \{\kappa = \gamma\}) = C'}{\Gamma;\Sigma \vdash \lambda x : (\gamma)^\kappa.e \mid C \longrightarrow \lambda x : \gamma.\ e \mid C'}$$

$$(\textsc{Ec-Abs-Fail})\ \frac{\text{unify}\ (C \cup \{\kappa = \gamma\}) = \text{fail}}{\Gamma;\Sigma \vdash \lambda x : (\gamma)^\kappa.\ e \mid C \longrightarrow \mathbf{error} \mid C}$$

Single-step evaluation rules for E_Γ^m

$$\frac{\Gamma;\Sigma \vdash e^m \mid C \longrightarrow^m e^{m\prime} \mid C'}{E_\Gamma^m[e^m] \mid C;\Sigma \longmapsto E_\Gamma^m[e^{m\prime}] \mid C';\Sigma}$$

$$\frac{\Gamma;\Sigma \vdash e^m \mid C \longrightarrow^m \mathbf{error} \mid C}{E_\Gamma^m[e^m] \mid C;\Sigma \longmapsto \mathbf{error} \mid C;\Sigma}$$

Single-step evaluation rules for E_Γ

$$\frac{\Gamma;\Sigma \vdash e \mid C \longrightarrow e' \mid C'}{E_\Gamma[e] \mid C;\Sigma \longmapsto E_\Gamma[e'] \mid C';\Sigma}$$

$$\frac{\Gamma;\Sigma \vdash e \mid C \longrightarrow \mathbf{error} \mid C}{E_\Gamma[e] \mid C;\Sigma \longmapsto \mathbf{error} \mid C;\Sigma}$$

Figure 11. Selected evaluation contexts and reduction rules

let meta $\mathsf{f}_1 = \prec(\sim (\mathsf{f}_0\ \textbf{int}))^{[\ell_2]}\succ$ **in**
...

Garcia's type system reports the type error after the entire meta evaluation but ideally we would like it to report the error just after f_0 is applied into **int** and the error message points to the place inside the definition of f_0, not generated code. According to the constraint generation rules, id has type **bool**→ $\exists\alpha$. **code** $(\alpha \to \alpha)$. At ℓ_1, suppose α is renamed to α_1 and the type of $\prec(\sim$ (id **false**))$^{[\ell_1]}$ 1$\succ$ could be some fresh unification variable β and the generated constraint set, written C_1, is equal to $\{\alpha_1 \to \alpha_1 = \textbf{int} \to \beta\}$. So, f_0 has type **type** $\to \exists\alpha_1,\beta$. (**code** β) and the constraint set is still C_1. At ℓ_2, suppose α_1 is renamed to α_2 and β is renamed to β_1. So, $(\sim (\mathsf{f}_0\ \textbf{int}))^{[\ell_2]}$ has type β_1 and the constraint set, written C_2, is up-

dated into $\{\alpha_2 \to \alpha_2 = \textbf{int} \to \beta_1\}$ by $\{\alpha_2/\alpha_1\}\{\beta_1/\beta\}C_1$. During meta evaluation, (f_0 **int**) is evaluated into

```
let meta id =
  λx : bool. ≺λx : (if x then int else bool)^α. x≻
in
  ≺(∼ (id false))^[ℓ1,ℓ2] 1≻
```

and please note that the label attached to $\sim$ (id **false**) is changed into ℓ_1, ℓ_2. The above program is further evaluated into

```
≺(∼ (λ x : bool. ≺λx : (if x then int else bool)^α2. x≻
      false))^[ℓ1,ℓ2] 1≻
```

and please note that unification variable α is renamed to α_2 as α is first renamed to α_1 by ℓ_1 and α_1 is then renamed to α_2 by ℓ_2. The above program is further evaluated into

```
≺(∼ (≺λ x : (if false then int else bool)^α2. x≻))^[ℓ1,ℓ2] 1≻
```

and further evaluated into

```
≺(∼ (≺λ x : (bool)^α2. x≻))^[ℓ1,ℓ2] 1≻
```

Now, we put the constraint $\alpha_2 = \textbf{bool}$ into constraint set C_2. Of course, we can discover the type conflict, **bool** = **int**, by unifying the new constraint set and because of the type conflict, the meta valuation terminates. The type error message points to the type of object variable x, which is desired.

6. Related Work

There are several threads of related work concerning the type-checking of the expressions whose types may depend on computation.

C++ templates and Template Haskell [5] are the alternative metaprogramming languages, but their type systems have the similar issue with type error tracking. For C++ templates, object-level fragments are type-checked before template instantiation. So, the C++ type system can catch some type errors earlier than Garcia's type system: it will reject the code ≺**true** + 1≻. However, templates are not fully type-checked before instantiation: (1) type-checking is not performed at the variables whose types depend on template type variables; (2) type-checking is not performed at the places of code generation. Here we use the example that frequently used by other papers about metaprogramming: the power function that performs partial evaluation at compile time. The below is the power function template in C++. We mis-define the function: powN has the argument of type **string** instead of **int** and it has no base case (when N = 0).

```
template<int N>
int powN(string m) { return powN<N−1>(m) ∗ m; };

template int powN<20>(string);
```

Ideally, we would like the C++ compiler to report the type error without doing any instantiation (**string** does not support multiplication). On the contrary, the compiler goes into an infinite loop of instantiation instead of reporting the error. For Template Haskell, object-level expressions are type-checked before meta evaluation but their types are hidden at the meta level. This mechanism enables the Haskell compiler to accept more expressive metaprograms such as the generic_add function presented previously. When type-checking a top-level splice (here refers to those splices not appearing inside a pair of quasi-quotes [| |]), say $e (dollar sign means splice), the Haskell compiler has to first meta-evaluate e and then type-check the evaluation result. This strategy may not only cause the compiler to catch type errors late (it also goes into a loop without reporting the error for the powN function) but also report imprecise messages (pointing to a wrong place). Consider the following case:

```
a = [| "A" |];
b = [| "B" |];
ab = [| $a ∗ $b |];    // error originates at here
...
s = $ab;        // top level splice; error message points here
```

In the above program, the type error is that **string** does not support multiplication, which refers to the 3rd line. However, the Haskell compiler returns the error message that points to the top level splice, the place could be far away from where the error originates.

Some programming languages use dependent types to describe those programs whose types are dependent on terms. Cayenne [15] supports the manipulation of types as normal data. It has dependent types, which contain the functions that generate types, but usually those functions have to be manually written. Type-checking for Cayenne programs involves computations, which may overlap with program computation. So, the type system may take huge computation overhead. In contrast, our type system takes advantage of meta evaluation such that type-checking and meta-evaluation interleave. The type system of Cayenne is modular, which enables us to write safe programs, but it restricts the set of program we can write. In contrast, our type system ensures the expressiveness of metaprograms.

Some other dependently typed programming languages, such as DML [16], ATS [17], and Concoqtion [18], have to use some external proof assistant tools (i.e. theorem provers). Concoqtion extends from MetaOCaml, which has meta-level expressions embedded in types (alternatively called indexed types). It can write expressive programs with pattern-matching (their types are dependent on datatype constructors) using the variant of GADTs [19]. GADTs are regarded as a limited version of dependent types. Though we can use GADTs to simulate if-expressions returning values of different types, in a way very similar to how one would do that using dependent types in Cayenne, yet it may have huge overhead of type-level computation. The type system of Concoqtion is implemented using the theorem prover Coq, which handles the type-level computation during type-checking. For such type system, the ease of programming is weakened as programmers may need to write proofs.

Some program analysis tools also provide solutions on how to type-check dynamically-typed programs. Mix [20] is a program analysis tool that mixes type-checking (for statically typed program blocks) with symbolic execution (for dynamically typed program blocks). With an SMT solver, it performs path sensitive analysis on a dynamically typed if-expression. It symbolically executes both of the branches and ensures that both of the execution paths do not generate type errors. Of course, SMT solving can be undecidable when the size of a program block for symbolic execution is too large.

7. Conclusion

Because type-reflective metaprograms have type dependencies, applying the MetaML-style modular type-checking approach to type-check those programs could compromise the flexibility of metaprogramming. In this paper, we design a new type-checking approach, incremental type checking, that can detect type errors in a type-reflective metaprogram as early as possible without sacrificing flexibility. Our approach is more efficient compared with other approaches by avoiding unnecessary meta evaluation overhead.

In the future, we would like to extend Garcia's calculus with object-oriented features, such as class reflection, i.e. iteration over class methods and class fields, class generation that generates derived classes from unknown base classes, etc. Class reflection has already developed in Java as an important and new feature. In C++, templates themselves are a useful tool for class generation but they do not have the power to inquire the inner structure of a class. Also,

class reflection at the meta level is not often explored and discussed by previous papers. So, it would be interesting to discover how metaprogramming can benefit class reflection and it would also be meaningful to develop a calculus that would surpass the power of C++ templates for class reflection.

Acknowledgments

This work was supported by the NSF under grant CCF-0702362.

References

[1] David Abrahams and Aleksey Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond*. Addison-Wesley Professional, 2004.

[2] Andrei Alexandrescu. *Modern C++ design: generic programming and design patterns applied.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[3] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative programming: methods, tools, and applications*. ACM Press/Addison-Wesley, 2000.

[4] Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. In *PEPM '97: Proceedings of the 1997 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 203–217, New York, NY, USA, 1997. ACM Press.

[5] Tim Sheard and Simon Peyton Jones. Template metaprogramming for Haskell. In *ACM SIGPLAN Haskell Workshop 02*, pages 1–16, 2002.

[6] Ronald Garcia and Andrew Lumsdaine. Toward foundations for type-reflective metaprogramming. In *GPCE '09: Proceedings of the eighth international conference on Generative programming and component engineering*, pages 25–34, New York, NY, USA, 2009. ACM.

[7] Ronald Garcia. *Static Computation and Reflection*. PhD thesis, Indiana University, September 2008.

[8] Taha Walid. Multi-stage programming: Its theory and applications. Technical report, 1999.

[9] Walid Taha and Michael Florentin Nielsen. Environment classifiers. In *POPL '03: Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 26–37, New York, NY, USA, 2003. ACM Press.

[10] Guy L. Steele, Jr. *Common LISP: the language (2nd ed.)*. Digital Press, Newton, MA, USA, 1990.

[11] Jeremy G. Siek and Manish Vachharajani. Gradual typing with unification-based inference. In *Dynamic Languages Symposium*, July 2008.

[12] Jeremy Siek and Walid Taha. Typing for objects. In *European Conference on Object-Oriented Programming*, 2007.

[13] Philip Wadler and Robert Bruce Findler. Well-typed programs can't be blamed. In *In workshop on Scheme and functional programming*, pages 15–26, 2007.

[14] Ian Lynagh. Typing template haskell: Soft types. In *GPCE W2: Second MetaOCaml Workshop*, 2005.

[15] Lennart Augustsson. Cayenne—a language with dependent types. In *ICFP '98: Proceedings of the third ACM SIGPLAN international conference on Functional programming*, New York, NY, USA. ACM.

[16] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 214–227, 1999.

[17] Hongwei Xi. Applied type system. *In post-workshop Proceedings of TYPES, Lecture Notes in Computer Science*, 3085, 2004.

[18] Seth Fogarty, Emir Pasalic, Jeremy Siek, and Walid Taha. Concoqtion: indexed types now. *In Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, 2007.

[19] Hongwei Xi, Chiyan Chen, and Gang Chen. Guarded recursive datatype constructors. In *Proceedings of the 30th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 224–235, New Orleans, January 2003.

[20] Khoo Yit Phang, Bor-Yuh Evan Chang, and Jeffrey S. Foster. Mixing type checking and symbolic execution. In *ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, 2010.