Early Detection of Type Errors in C++ Templates [*]

Sheng Chen
School of EECS, Oregon State University
chensh@eecs.oregonstate.edu

Martin Erwig
School of EECS, Oregon State University
erwig@eecs.oregonstate.edu

Abstract

Current C++ implementations typecheck templates in two phases: Before instantiation, those parts of the template are checked that do not depend on template parameters, while the checking of the remaining parts is delayed until template instantiation time when the template arguments become available. This approach is problematic because it causes two major usability problems. First, it prevents library developers to provide guarantees about the type correctness for modules involving templates. Second, it can lead, through the incorrect use of template functions, to inscrutable error messages. Moreover, errors are often reported far away from the source of the program fault.

To address this problem, we have developed a type system for Garcia's type-reflective calculus that allows a more precise characterization of types and thus a better utilization of type information within template definitions. This type system allows the static detection of many type errors that could previously only be detected after template instantiation. The additional precision and earlier detection time is achieved through the use of so-called "choice types" and corresponding typing rules that support the static reasoning about underspecified template types. The main contribution of this paper is a guarantee of the type safety of C++ templates (general definitions with specializations) since we can show that well-typed templates only generate well-typed object programs.

Categories and Subject Descriptors F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs–Type structure; D.3.1 [*Programming Languages*]: Formal Definition and Theory

Keywords Metaprogramming, type systems, type reflection, choice types, C++ Templates

1. Introduction

C++ templates help construct safer programs by shifting runtime error checking to static type checking. For example, with templates C++ container classes do not have to rely on the notorious `void*` type, and they can guarantee that only type-correct data can be stored in a specific container.

However, type-safety guarantees about the template programs before they are instantiated are limited since only expressions that do not depend on any template parameter are type checked at the definition site. In the following example, there is a type error in the expression `return powN<N-1>(m)*m` because `powN<N-1>(m)` is of type `int` and `m` is of type `string`. However, the statement `return powN<N-1>(m)*m` will not be type checked until the function `powN` is instantiated since the expression `powN<N-1>(m)` uses the template parameter `N`. Therefore, the detection of the type error will be unnecessarily delayed.

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

Type checking of templates happens in two phases. When the template definition is first parsed, expressions that do not involve any template parameters are type checked, while those that depend on template parameters, directly or indirectly, are stored as abstract syntax trees without any associated type information. When the template is instantiated, template parameters are replaced with concrete arguments, and full type checking is performed.[1] This model of type checking causes several usability problems.

First, library providers cannot guarantee that their code is type correct. Second, misuse of template functions results in large, inscrutable error messages. Third, since many type errors in template programs are not captured until programs are instantiated, they are often reported far away from where they really occur.

The language feature of *concepts* has been proposed to address these usability issues [9, 15]. Unfortunately, however, concepts face some problems. First, they incur a burden for programmers, forcing them to develop concepts manually. Second, as discussed in Section 6, concepts do not provide a satisfying solution for the typing problem.

A different route was taken by Garcia and Lumsdaine [14] who presented a core calculus that captures the fundamental capabilities while avoiding many shortcoming of C++ metaprogramming. This calculus provides the foundation for further study of the typing problem. Figure 1 shows this calculus.

The calculus mainly consists of two parts. An *object language* that expresses the computation performed during runtime and a *metalanguage* to manipulate object language expressions, providing a means for static computation.

The calculus provides well-known features for metaprogramming [34, 35]. For example, the *splice* expression $\sim e^m$ escapes to the compile-time computation, that is, the computation of e^m is performed at compile time; the bracket expression $\prec e \succ$ injects a piece of object code into metacode; the *lift* expression $\%e^m$ first evaluates e^m at compile time and then injects the resulting value into object code.

At the same time, the calculus provides the full machinery for constructing and querying object types that are essential to C++

[*] This work is supported by the National Science Foundation under the grants CCF-0917092, CCF-1219165, and IIS-1314384.

PEPM '14, January 20–21, 2014, San Diego, CA, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2619-3/14/01…$15.00.
http://dx.doi.org/10.1145/2543728.2543731.

[1] Depending on the compiler implementation, some type checking may be delayed to link time.

γ	constant types (**int**, **float**, **string**)
x	meta-level variables
c	value and function constants

object types

$\tau \quad ::= \quad \gamma \mid \tau \to \tau$

object language

$e \quad ::= \quad x \mid c \mid \lambda x\colon e^m.e \mid e\ e \mid {\sim}e^m \mid \textbf{let}\ x = e\ \textbf{in}\ e \mid$
$\qquad \textbf{let meta}\ x = e^m\ \textbf{in}\ e \mid \textbf{if}\ e\ \textbf{then}\ e\ \textbf{else}\ e$

meta types

$\tau^m \quad ::= \quad \gamma \mid \textbf{code} \mid \textbf{type} \mid \tau^m \to \tau^m$

meta language

$e^m \quad ::= \quad x \mid c \mid c^m \mid \gamma \mid \lambda x\colon \tau^m.e^m \mid e^m\ e^m \mid \prec e \succ \mid$
$\qquad \%e^m \mid \textbf{if}\ e^m\ \textbf{then}\ e^m\ \textbf{else}\ e^m \mid \textbf{let}\ x = e^m\ \textbf{in}\ e^m$

$c^m \quad ::= \quad \to \mid \gamma^? \mid \to^? \mid =_\tau \mid \textbf{dom} \mid \textbf{cod} \mid \textbf{typeof}$

Figure 1: Garcia's calculus for C++ Metaprogramming

Metaprogramming (C++ MP). The type constructor $\to$ takes two types and constructs an arrow type. The type destructors **dom** and **cod** return the argument type and result type, respectively, of an arrow type. The unary predicates $\gamma^?$ and $\to^?$ determine if the argument expression e^m represents the type γ or an arrow type, respectively. We say an expression e^m represents type γ if e^m evaluates, through metareduction, to γ. The binary predicate $=_\tau$ decides if two expressions represent the same type. Finally, **typeof** allows metaprograms to query the type of a piece of object code.

Like C++ templates, Garcia's type system only assures that the generated object programs are well formed but not well typed. However, looking back at the `powN` example, we can observe that although the expression `powN<N-1>(m)` depends on the template parameter `N`, we are still able to find out that the expression is of type **int**, which is enough to help us detect the type error in the program because we know that `*` does not accept an **int** value and a **string** value.[2]

Miao and Siek [22] exploit this kind of type information to detect type errors earlier. Employing the idea of gradual typing [29, 31], they use existential types to characterize metaexpressions. The type system collects type information as metaprograms are instantiated and metaevaluations are making progress. It reports the type error when a conflicting use of a type is detected.

In essence, existential types approximate the type information for metaexpressions, that is, some important information is lost, which causes some type errors to be detected later than necessary. For example, consider the following program, presented in [22], where f_0 is a metaprogram that takes a type argument t and defines an identity function *id*. Based on the non-type template parameter x, *id* has the type **code** $(t \to t)$ when x is **true** or has the type **code** (**bool** $\to$ **bool**) when x is **false**.

```
let meta $f_0$ =
    $\lambda t$: type.
      let meta $id$ =
        $\lambda x$: bool. $\prec \lambda y$: if $x$ then $t$ else bool.$y \succ$
      in
        $\prec ({\sim}(id\ \textbf{false}))\ 1^{①} \succ$
in
    let meta $f_1 = \prec {\sim}(f_0\ \textbf{int})^{②} \succ$ in
      ...
```

We know that at position ① the expression *id* **false** has the type **code** (**bool** $\to$ **bool**) and the expression ${\sim}$(*id* **false**) has the type **bool** $\to$ **bool**. Thus, we can catch the type error there because the type of the argument, which is **bool**, is conflicting with the type of `1`, which is **int**. However, Garcia's type system does not catch the error until the metaprogram is instantiated; Miao and Siek's type system detects the error at position ②. The reason for this imprecision is that the type for *id* is approximated to **bool** $\to \exists\alpha.$**code** $(\alpha \to \alpha)$, which misses the information that when the template argument to *id* is **false**, the type of *id* is **code** (**bool** $\to$ **bool**). Thus when ${\sim}$(*id* **false**) is applied to an **int** value, the type error goes undetected at ①.

From the analysis we can learn that it is possible to catch type errors earlier if we make better use of available type and value information. However, a corresponding type checking method faces many challenges. For example, how to precisely represent the types for metaexpressions, and how to reason with uncertain type information? In the example, what should be the type for the metafunction *id*? Given the fact that **false** is a static value, how do we deduce the type for *id* **false**? Previous type systems assign only one type to each expression. However, for metaprograms this is generally insufficient. For example, the *id* function has two different types, depending on the instantiation argument. This kind of situation is prevalent in C++ template programs, for example, in implementing the traits technique [1].

We address these challenges by using *choices* [10] of values and types to encapsulate all potential types for metaexpressions. We construct and eliminate choices as the type checking process unfolds. Whenever needed, we use variational type unification to solve type equality constraints among types [6].

The two possible types for the *id* function can be represented by the type **code** $(D\langle t, \textbf{bool}\rangle \to D\langle t, \textbf{bool}\rangle)$, where the two choice types are synchronized by the use of the common name D. The dependency of the chosen type on the boolean value can be encoded in a dependent type $D\langle \textbf{true}, \textbf{false}\rangle \to \textbf{code}\ (D\langle t, \textbf{bool}\rangle \to D\langle t, \textbf{bool}\rangle)$ where the dependency is expressed by a choice between two values using again the same name D. This type expresses that the type of *id* depends on the instantiation argument. If it is instantiated with **true**, the type is **code** $(t \to t)$, and **code** (**bool** $\to$ **bool**) otherwise. At position ①, since the instantiation argument is **false**, we infer that the expression ${\sim}$(*id* **false**) has the type **bool** $\to$ **bool**. We can therefore catch the type error where it occurs.

The main contribution of this paper is the design of a modular type system for generic C++ template definitions with specializations. Through the use of choice types we can represent uncertain type information for metaexpressions. Together with very fine-grained types that represent static computations, we can precisely reason about the types of metaprograms. Since our type system separates the type checking of the definitions and the use of metafunctions, we can type check object programs without generating them. Consequently, the type system ensures that well-typed metaprograms only generate well-typed object programs.

The rest of the paper is organized as follows. In Section 2, we present the necessary background for formalizing the type system. We define the type system in Section 3, where we also discuss the properties of the type system. A sound and complete implementation of the type system is presented in Section 4. In Section 5 we evaluate our type system by comparing its behaviors with that of other type systems and by investigating its expressiveness. We discuss related work in Section 6 and conclude the paper in Section 7, where we also discuss directions for future research.

2. Background

This section introduces some technical background for presenting the type system for C++ templates. Specifically, in Section 2.1 we present the concept of choices, and in Section 2.2 we introduce metareduction, which is used to relate the properties of different type systems.

[2] More precisely, the string class does not have a `*` operator overloaded to have **int** as its first parameter type.

$$\textsc{Choice}\ \frac{\tau_1 \equiv \tau_1' \qquad \tau_2 \equiv \tau_2'}{D\langle \tau_1, \tau_2 \rangle \equiv D\langle \tau_1', \tau_2' \rangle} \qquad \textsc{Fun}\ \frac{\tau_l' \equiv \tau_r' \qquad \tau_l \equiv \tau_r}{\tau_l' \to \tau_l \equiv \tau_r' \to \tau_r}$$

$$\textsc{F-C}\quad D\langle \tau_1, \tau_2 \rangle \to D\langle \tau_1', \tau_2' \rangle \equiv D\langle \tau_1 \to \tau_1', \tau_2 \to \tau_2' \rangle$$

$$\textsc{C-Idemp}\ \frac{\tau_1 \equiv \tau \qquad \tau_2 \equiv \tau}{D\langle \tau_1, \tau_2 \rangle \equiv \tau} \qquad \textsc{C-C-Merge1}\quad D\langle D\langle \tau_1, \tau_2 \rangle, \tau_3 \rangle \equiv D\langle \tau_1, \tau_3 \rangle$$

$$\textsc{C-C-Swap1}\quad D'\langle D\langle \tau_1, \tau_2 \rangle, \tau_3 \rangle \equiv D\langle D'\langle \tau_1, \tau_3 \rangle, D'\langle \tau_2, \tau_3 \rangle \rangle$$

Figure 2: Variational type equivalence

2.1 Choices

The choice calculus was introduced in [10] as a generic variation representation for software. It provides named choices to represent variation points in programs (and other tree-structured artifacts). As an example, consider the following variational expression that applies one of two functions to a constant.

$$A\langle \texttt{odd}, \texttt{inc} \rangle\ \texttt{3}$$

Here the choice $A\langle\rangle$ denotes a named variation point. We can generate two variants from this expression, `odd 3` and `inc 3`, depending on which alternative of the choice is selected. Different choices tagged by the same name will be synchronized, that is, the expression $A\langle \texttt{odd}, \texttt{inc} \rangle\ A\langle 3, 5 \rangle$ represents only two expressions `odd 3` and `inc 5`, while differently named choices are independent of one another, that is, $A\langle \texttt{odd}, \texttt{inc} \rangle\ B\langle 3, 5 \rangle$ would generate four plain expressions, resulting from combining each function with each argument.

The types of variational expressions can as well be variational. For example, under the typing rules developed in [7], the above example expression has the choice type $A\langle \mathbf{bool}, \mathbf{int} \rangle$.

The name of a choice is called a *dimension* and is introduced through a binding construct that also defines *tags* to make selections in choices. In our example we can add a dimension declaration as follows.

$$\mathbf{dim}\ A\langle a, b \rangle\ \mathbf{in}\ A\langle \texttt{odd}, \texttt{inc} \rangle\ \texttt{3}$$

This dimension declaration allows us to select `odd` using the tag $A.a$. In the following we will employ the choice calculus to represent alternatives in types through choices. Moreover, tags for s-electing specific variants will be represented by choices of values that serve as type indices.

Syntactically different types and values can be equivalent, for example, the type $A\langle \mathbf{int}, \mathbf{int} \rangle$ is equivalent to **int** since whatever decision made about choice $A\langle\rangle$, the result is always **int**. We write $\tau_1 \equiv \tau_2$ to express that the types τ_1 and τ_2 are equivalent. Figure 2 lists a subset of the type equivalence rules developed in [7]. For brevity, we have omitted the standard rules for reflexivity, symmetry, and transitivity that turn $\equiv$ into an equivalence relationship, and the two rules C-C-MERGE2 and C-C-SWAP2 that work analogously to the shown rules with the second alternative. Most of the rules are straightforward. Note that the rule C-C-MERGE1 allows us to introduce or eliminate the dead alternative τ_2, which can't be reached with neither $D.1$ nor $D.2$. The rule C-C-SWAP1 enables us to reorder the choice nesting by swapping choices.

An important result about typing choice expressions is that the typing relationship is preserved over selection, that is, the typing process commutes with selections. If the variational expression e is of the type t, then the type of an expression derived from e through a specific selection can be derived from t with the same selection. For example, the choice expression $A\langle \texttt{odd}, \texttt{inc} \rangle\ \texttt{3}$ has the type $A\langle \mathbf{bool}, \mathbf{int} \rangle$. If we select the first alternative from choice A, then we know in advance that `odd 3` has the type **bool**.

type indices	σ	$::= \gamma \mid \sigma \to \sigma \mid \alpha$
object types	τ	$::= \sigma \mid D\langle \overline{\tau} \rangle$
meta values	ξ	$::= \tau \mid c \mid D\langle \overline{c} \rangle$
meta annot.	δ	$::= \mathbf{code}\ \tau \mid \mathbf{type} \mid \eta$
meta types	ϕ	$::= \mathbf{code}\ \tau \mid \mathbf{type}_\tau \mid \mathbf{type}^x_{D\langle \overline{\sigma}, \varepsilon \rangle} \mid \eta_c \mid \eta^x_{D\langle \overline{c}, \varepsilon \rangle} \mid D\langle \overline{\phi} \rangle \mid \phi \to \phi$
choice indices	t	$::= c \mid \sigma \mid \varepsilon$
proper object code	e^o	$::= x \mid c \mid \lambda x \colon \sigma . e^o \mid e^o\ e^o$
normal forms	v^m	$::= c \mid \sigma \mid \lambda x \colon \delta . e^m \mid \prec e^o \succ$

Figure 3: Definitions of Types and related concepts

2.2 Meta Reduction

Garcia [14] defines a set of local rewriting rules for reducing metaredexes to metavalues in the form of $e_1^m \to_{E^m} e_2^m$. For example, $\mathbf{dom}\ (\tau_1 \to \tau_2) \to_{E^m} \tau_1$. Based on that relation, the metareduction relation can be represented as follows

$$\textsc{meta-eval}\ \frac{e_1^m \to_{E^m} e_2^m}{E^m[e_1^m] \longmapsto E^m[e_2^m]}$$

where E^m represents a metaevaluation context that can be filled with a metaexpression. If e^m is not object code, then it can be subject to reduction in only one way because that e^m can be decomposed uniquely into a context and a redex [14]. Essentially, this relation models template instantiation and the object program generation process.

3. Type System

This section presents the basic features of the type system that facilitates the detection of type errors without generating object programs.

Section 3.1 presents the types for typing both object programs and metaprograms. A crucial aspect of the type system is that we use choice types to represent uncertain type information. In Sections 3.2 and 3.3, we discuss the core parts of the type system, namely reasoning about the type representation along the typing process. In Section 3.4, we investigate some properties of the type system.

3.1 Type syntax

Figure 3 specifies the types for object programs and the metaprograms. We use γ to range over constant types, such as **int** or **bool**, and α to range over type variables. Except for choice types $D\langle \overline{\tau} \rangle$, the types for typing object programs are similar to those for the lambda calculus [24]. A choice type $D\langle \overline{\tau} \rangle$ represents an uncertain type for object programs, to be employed in cases when abstractions are annotated by metaexpressions.

We use δ to describe the types of template parameters, that is, the annotations attached to meta-abstractions. C++ support-s both type parameters and non-type parameters in defining templates [32]. Non-type template parameters can be integral or enumeration types (η) or pointers to functions ($\mathbf{code}\ \tau$).[3]

To precisely represent the types for metaexpressions, we define very fine-grained types so that the mapping from expressions to types will cause minimal information loss. For example, instead

[3] C++ also allows non-type parameters to be pointers to objects, pointers to members, and references to functions and objects. These can be handled similarly; we omit the discussion for simplicity.

of **type** we use types like $\textbf{type}_{\textbf{int}}$, $\textbf{type}_{\textbf{bool}}$, and so on. Likewise, instead of using **int**, which can only represent the type of all integer values, we use type-level integers and booleans such as $\textbf{int}_1$, $\textbf{bool}_{\textbf{false}}$, and so on. Note that the information needed to specialize, for example, **type** by **int** or **bool**, **int** by 1, or **bool** by **false**, are all available at compile time, which makes the fine-grained type representation available for type checking.

Types can also be indexed by choices—choices of values (represented by $\eta^x_{D\langle\overline{c},\varepsilon\rangle}$) and choices of types (represented by $\textbf{type}^x_{D\langle\overline{\sigma},\varepsilon\rangle}$). In the following discussion we focus on the latter kind. The case for value-indexed types is analogous. A function type with $\textbf{type}^x_{D\langle\overline{\sigma},\varepsilon\rangle}$ as its argument type can express both universally quantified types and dependent types, expressed by choice-type indices. The meta-level variable x (see Figure 1) serves as the placeholder for the types or values passed to the metafunctions. We can view these as universally quantified type variables as we can pass any types to metafunctions with type template parameters.

The type index $D\langle\overline{\sigma},\varepsilon\rangle$ expresses a dependency (it may be dropped when the return type is independent of the values of the argument). Note that all the indices must be of type **type**. Moreover, we assume them to be unique. The special tag ε matches any value. With the type definitions, we can use $\textbf{type}^x \to \textbf{type}_x$ to represent the type $\forall x.x$, and the type of the function *id* from Section 1 is as follows.

$$\textbf{bool}^x_{D\langle\textbf{true},\textbf{false}\rangle} \to \textbf{code}\ (D\langle t,\textbf{bool}\rangle \to D\langle t,\textbf{bool}\rangle)$$

Note that we can also express more complex types such as the following.

$$\textbf{type}^x_{D\langle\textbf{bool},\textbf{int},\varepsilon\rangle} \to D\langle\textbf{type}_{\textbf{int}},\textbf{type}_{\textbf{bool}},\textbf{type}_x\rangle \tag{1}$$

This type characterizes the metafunctions that accept and return object types. More precisely, when the input is **int** the output is **bool** and vice versa, and for all other types, the output is the same as the input.

While the choice types in [6, 7] are global, the choice types in this paper are scoped, and the subscript $D\langle\overline{t}\rangle$ in $\delta^x_{D\langle\overline{t}\rangle} \to \phi$ serves as a dimension declaration [10]. (We let δ range over η and **type**.) The declared tags will act as selectors for choices only in ϕ. Since all choices are introduced during compile time, we can ensure that they are all unique. This restriction significantly simplifies the definition of tag selection.

We let t range over tags (that is, type indices), which, besides constant values and types, also includes default tag ε that matches any value.

The tag selection operation $\lfloor\phi\rfloor_{D.t}$, given the type ϕ and a selector $D.t$, is defined by traversing the tree representation of ϕ and making appropriate selections. Conceptually, it is realized in two phases. First, use the choice D in the selector to search for a dimension declaration, and when found, use t to decide which alternative to select. When no tag matches, the alternative corresponding to tag ε will be taken. Assume the ith alternative is chosen in this phase. Second, traverse the rest of the tree representation of the type, and when the current node is a choice with the name D, the node is replaced by its ith alternative. The process recursively descends to all the children of the current node and makes selections. In the following, we present the cases in which the traversal hits the choice for which the selection is intended.

Formally, we define the tag selection process as follows.

$$\begin{aligned}
\lfloor\delta_{D\langle t_1,\cdots,t_i,\cdots,t_n,\varepsilon\rangle} \to \phi\rfloor_{D.t_i} &= \lfloor\phi\rfloor_{D.i} \\
\lfloor\delta_{D\langle t_1,\cdots,t_i,\cdots,t_n,\varepsilon\rangle} \to \phi\rfloor_{D.t} &= \lfloor\phi\rfloor_{D.n+1} && \forall i\colon t \neq t_i \\
\lfloor\phi_1 \to \phi_2\rfloor_{D.i} &= \lfloor\phi_1\rfloor_{D.i} \to \lfloor\phi_2\rfloor_{D.i} \\
\lfloor D\langle\phi_1,\cdots,\phi_i,\cdots,\phi_n\rangle\rfloor_{D.i} &= \phi_i
\end{aligned}$$

T-ABS
$$\frac{\Gamma;\Psi;\Delta\vdash_m e^m : \phi \qquad \phi \equiv \textbf{type}_\tau \qquad \Gamma,(x,\tau);\Psi;\Delta\vdash_o e : \tau'}{\Gamma;\Psi;\Delta\vdash_o \lambda x : e^m.e : \tau \to \tau'}$$

T-APP
$$\frac{\Gamma;\Psi;\Delta\vdash_o e_1 : \tau_1 \qquad \Gamma;\Psi;\Delta\vdash_o e_2 : \tau_2 \qquad \tau_1 \equiv \tau_2 \to \tau}{\Gamma;\Psi;\Delta\vdash_o e_1\ e_2 : \tau}$$

T-ESP
$$\frac{\Gamma;\Psi;\Delta\vdash_m e^m : \textbf{code}\ \tau}{\Gamma;\Psi;\Delta\vdash_o \sim e^m : \tau}$$

T-IF
$$\frac{\begin{array}{c}\Gamma;\Psi;\Delta\vdash_o e_1 : \textbf{bool} \\ \Gamma;\Psi;\Delta\vdash_o e_2 : \tau_2 \qquad \Gamma;\Psi;\Delta\vdash_o e_3 : \tau_3 \qquad \tau_2 \equiv \tau_3\end{array}}{\Gamma;\Psi;\Delta_1\vdash_o \textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 : \tau_2}$$

Figure 4: Typing rules for object language

We observe that there is no superscript x associated with the metatypes since the substitution of x occurs before selection. For example, assume a metafunction whose type is expression (1) is applied to the type **char**. Then x is substituted with **char** and expression (1) becomes the following.

$$\textbf{type}_{D\langle\textbf{bool},\textbf{int},\varepsilon\rangle} \to D\langle\textbf{type}_{\textbf{int}},\textbf{type}_{\textbf{bool}},\textbf{type}_{\textbf{char}}\rangle$$

Next, we make a selection of this expression with the selector $D.\textbf{char}$. We first decide that the third alternative of choice D will be chosen, and then we select third alternative from the return type, resulting in the type $\textbf{type}_{\textbf{char}}$.

Given the extended type definitions, the type equivalence relation has to be augmented by the following straightforward rules (which also hold for n-ary choices).

$$\begin{aligned}
\textbf{code}\ D\langle\tau_1,\tau_2\rangle &\equiv D\langle\textbf{code}\ \tau_1,\textbf{code}\ \tau_2\rangle \\
\textbf{type}_{D\langle\tau_1,\tau_2\rangle} &\equiv D\langle\textbf{type}_{\tau_1},\textbf{type}_{\tau_2}\rangle
\end{aligned}$$

3.2 Typing rules for the object language

Figure 4 presents a subset of typing rules for object programs. Typing rules for other constructs are similar to thoses in type systems for lambda calculi [24] and are omitted here.

The typing judgment has the form $\Gamma;\Psi;\Delta\vdash_o e : \tau$, with environments Γ and Ψ storing type bindings for object variables and metavariables, respectively. The choice environment Δ is a mapping from metavariables to choices of tags. For example, $\Delta = \{(x, D\langle\textbf{true},\textbf{false}\rangle)\}$ maps x to a choice D between **true** and **false**. The environment Δ controls how choice types are constructed, which is a core part of typing metaexpressions. We discuss this in more depth when we present the typing rule for metaconditionals.

We need Ψ and Δ for typing object programs because metaexpressions can be used as type annotations for function parameters. Also, the escape expression $\sim e^m$ evaluates e^m to an object value, which is injected into the object program at compile time. To type such an expression, we have to first type e^m, which will access Ψ and Δ.

The abstractions in object code are more expressive than those of the simply-typed lambda calculus (STLC) [4] since the type annotations can be any metaexpression. Note that each object is itself a metaexpression. Thus, to type an abstraction we first type the metaexpression e^m and check if its type is equivalent to $\textbf{type}_\tau$ for some object type τ. If so, we bind the variable x to type τ and type check the body of the abstraction with the extended assumption. If

$$\textsc{M-ConsT}\quad \Gamma;\Psi;\Delta \vdash_m \gamma : \mathbf{type}_\gamma \qquad \textsc{M-ConsV}\quad \frac{c \text{ is of type } \eta}{\Gamma;\Psi;\Delta \vdash_m c : \eta_c} \qquad \textsc{M-Var}\quad \frac{(x,\phi) \in \Psi}{\Gamma;\Psi;\Delta \vdash x : \phi}$$

$$\textsc{M-Typeof}\quad \frac{\Gamma;\Psi;\Delta \vdash_m e^m : \mathbf{code}\ \tau}{\Gamma;\Psi;\Delta \vdash_m \mathbf{typeof}\ e^m : \mathbf{type}_\tau} \qquad \textsc{M-Code}\quad \frac{\Gamma;\Psi;\Delta \vdash_o e : \tau}{\Gamma;\Psi;\Delta \vdash_m \prec e \succ : \mathbf{code}\ \tau}$$

$$\textsc{M-Dom}\quad \frac{\Gamma;\Psi;\Delta \vdash_m e^m : \mathbf{type}_{\tau_1 \to \tau_2}}{\Gamma;\Psi;\Delta \vdash_m \mathbf{dom}\ e^m : \mathbf{type}_{\tau_1}}$$

$$\textsc{M-Teq}\quad \frac{\Gamma;\Psi;\Delta \vdash_m e_1^m : \mathbf{type}_{\tau_1} \qquad \Gamma;\Psi;\Delta \vdash_m e_2^m : \mathbf{type}_{\tau_2} \\ r = (\tau_1 \equiv \tau_2 \Rightarrow \mathbf{true}; \_ \Rightarrow \mathbf{false})}{\Gamma;\Psi;\Delta \vdash_m e_1^m =_\tau e_2^m : \mathbf{bool}_r}$$

$$\textsc{M-If}\quad \frac{(x, A\langle \bar{t}\rangle) \in \Delta \qquad \Gamma;\Psi;\Delta \vdash_m x : \delta_x \\ \Gamma;\Psi;\Delta \vdash_m v : \delta_v \qquad \Gamma;\Psi;\Delta \vdash_m e_2^m : \phi_2 \qquad \Gamma;\Psi;\Delta \vdash_m e_3^m : \phi_3}{\Gamma;\Psi;\Delta \vdash_m \mathbf{if}\ \ x = v\ \ \mathbf{then}\ \ e_2^m\ \ \mathbf{else}\ \ e_3^m : A\langle \phi_2 \rangle \Diamond \phi_3}$$

$$\textsc{M-IfTrue}\quad \frac{\Gamma;\Psi;\Delta \vdash_m e_1^m : \mathbf{bool_{true}} \qquad \Gamma;\Psi;\Delta \vdash_m e_2^m : \phi_2}{\Gamma;\Psi;\Delta \vdash_m \mathbf{if}\ \ e_1^m\ \ \mathbf{then}\ \ e_2^m\ \ \mathbf{else}\ \ e_3^m : \phi_2}$$

$$\textsc{M-IfFalse}\quad \frac{\Gamma;\Psi;\Delta \vdash_m e_1^m : \mathbf{bool_{false}} \qquad \Gamma;\Psi;\Delta \vdash_m e_3^m : \phi_3}{\Gamma;\Psi;\Delta \vdash_m \mathbf{if}\ \ e_1^m\ \ \mathbf{then}\ \ e_2^m\ \ \mathbf{else}\ \ e_3^m : \phi_3}$$

Figure 5: Typing rules for basic metaconstructors

under this assumption the body has type τ', then the abstraction has the type $\tau \to \tau'$.

There is a difference between typing applications in STLC and applications in object programs. While in STLC the argument type of the function must be syntactically identical to the type of the argument, the typing rule T-App only requires that the two types be equivalent as defined by rules in Figure 2. In fact, we can further relax this restriction by only requiring that the type of the function be equivalent to an arrow type whose argument type is the same as the type of the argument. If this is the case, then the result type is the return type of the arrow type.

For a metaexpression e^m to be successfully spliced into some other object expression, it must be evaluated to some value that has an object type. Thus, for the expression $\sim e^m$ to be well typed, the expression e^m must have the type **code** τ for some τ. We use $\Gamma;\Psi;\Delta \vdash_m e^m : \mathbf{code}\ \tau$ to achieve this. If this is the case, the typing result is τ. The rule T-If types the conditionals in the object program. If both branches have equivalent types, then the conditional has that type. Although the rule T-If itself is simple, it is needed as the counterpart for the corresponding typing rule M-if for metaconditionals.

3.3 Typing rules for the metalanguage

Since the typing rules for metaprograms have to deal with metacomputations, they are more involved. We break the presentation of the typing rules into two parts. Figure 5 shows the typing rules for basic metaconstructors and conditionals, and Figure 6 shows the rules for abstractions and applications.

For any constant type γ, such as **int** and **bool**, the rule M-ConsT states that it has the type $\mathbf{type}_\gamma$. The rule M-ConsV assigns types to constants of integral or enumeration types. The rule M-Var retrieves the type of a metavariable x from the metatyping environment Ψ. For a type query to be successful, the queried expression e^m must have the type **code** τ for some τ. In this case, the type of the whole expression is $\mathbf{type}_\tau$. For the expression $\prec e \succ$ to be well typed, the object code e must be well typed. The rule M-Code expresses this relation. Since **dom** extracts the argument type of an arrow type, **dom** e^m will succeed only when e^m evaluates to an object function type $\tau_1 \to \tau_2$. In this case, the type of the expression is τ_1. The rule M-Dom describes this.

The rule M-Teq is slightly more complicated. Since $=_\tau$ decides if two metaexpressions represent the same object type, we first have to check if both the operands have object types. If so, we further determine if the two types are equivalent since the metaexpressions can represent potentially many types. If two types are equivalent, then the result will be **true**. Thus, the result type is $\mathbf{bool_{true}}$. Otherwise, the result type is $\mathbf{bool_{false}}$.

In the typing rule we make use of a syntactic abbreviation for expressing conditional values in premises that essentially allows us to combine two inference rules in one. The expression $(c \Rightarrow e; c' \Rightarrow e')$ yields e if the condition c is true, and it yields e' if condition c' is true. We sometimes use $\_$ as a shorthand for 'true' for c' to express that the condition for $\_$ is always satisfied.

Unlike the typing rule for the object **if** statement, which requires that both **then** and **else** branches to have the same type, the meta **if** statement does not have this requirement. Given the statement **if** e_1^m **then** e_2^m **else** e_3^m, the result type is the type of e_2^m when the condition e_1^m is satisfied and the type of e_3^m otherwise. Thus, the result type is dependent on the condition. We use choice types to express such dependencies.

However, the type definition in Figure 3 allows tags to be only type indices and constant values. Moreover, tag selection is defined by equality checking of tags. Therefore, we only support conditions of the form $x = v$ where v is a metavalue (that is, any object type or any enumerable value). Together with the default tag ε, this limited form of comparison already allows us to type C++ generic definitions with specializations.

In fact, C++ Metaprogramming doesn't provide any metalevel **if** statements directly. Rather, it supports conditional operations through (partial) template specializations. Consider, for example the following program. We first give a generic definition stating that values of all types shouldn't be copied through low-level memory operations for the purpose of safety. We then define two specializations for the types **bool** and **int** such that their values may be copied through some faster memory operations due to their memory representations.

```
template<class Tp>
struct __type_traits{
    typedef _false_t fast_cp;
};

template<> struct __type_traits<bool>{
    typedef _true_t fast_cp;
};

template<> struct __type_traits<int>{
    typedef _true_t fast_cp;
};
```

We can translate the above program and its specializations into Garcia's calculus as follows. We observe that with the current definition of types and tag selection, together with the default tag ε, we can handle the generic definitions and their specializations.

```
__type_traits = $\lambda x$: type.
  if $x$ = bool then _true_t
  else if $x$ = int then _true_t
      else _false_t
```

In the rule M-IF, we first determine the choice type we will create from Δ, which controls how choice types are created as we mentioned earlier. We then retrieve the types for the two branches and then use the $\lozenge$ operation to merge two types into a choice type, which is formally defined as follows.

$$D\langle\overline{\phi_n}\rangle \lozenge D\langle\overline{\phi_m}\rangle = D\langle\overline{\phi_n},\overline{\phi_m}\rangle$$
$$D\langle\overline{\phi_n}\rangle \lozenge \phi = D\langle\overline{\phi_n},\phi\rangle$$

This operation essentially merges the corresponding alternatives when both are choice types and have the same name. Otherwise, the second type is treated as the last alternative of the first type, which must be a choice type.

To simplify the presentation of typing rules, we assume that the ordering of tags in Δ for a specific variable x matches the ordering it is compared with in the program. For example, before typing the body of the above `__type_traits` program, Δ contains $(x, A\langle\mathbf{bool},\mathbf{int},\varepsilon\rangle)$. Note that x is first compared with **bool** and thus it comes first in the tag list. In Section 4, we show an implementation that computes Δs in this manner.

This assumption doesn't limit our type system. We can lift this assumption by designing a more involved M-IF rule. Given the condition $x = v$, we not only retrieve the choice name bound to x as we are doing now, but also find the index in the tag list for v and use that index to decide where the type of the **then** branch should appear in the result type of the conditional.

For the `__type_traits` example, we assume $(x, A\langle\mathbf{bool},\mathbf{int},\varepsilon\rangle) \in \Delta$. Then after applying the rule M-IF twice, we obtain the following result type.

$$\phi_{\texttt{tt}} = A\langle\mathbf{type}_{\texttt{\_true\_t}},\mathbf{type}_{\texttt{\_true\_t}},\mathbf{type}_{\texttt{\_false\_t}}\rangle$$

The rule M-IF deals with the case that the condition doesn't have a specific type. When the condition has the type $\mathbf{bool_{true}}$ or $\mathbf{bool_{false}}$, we no longer have to build up choice types because we already know which branch will be taken. We only need to ensure that the relevant branch is well typed. There is no requirement on the other branch. We use rules M-IFTRUE and M-IFFALSE to deal with these two special cases.

We now turn our attention to the typing rules for meta-abstractions and meta-applications, which are presented in Figure 6. To type metafunctions (rule M-ABS1), we first extend Ψ with the assumption for the parameter, and we extend Δ with the choice and tags assumption for the parameter. Then we type the body under the extended assumptions. Note the minor difference between the annotation to the abstraction variable x, which is δ, and the type assumption for x in Ψ, which is δ_x. We need this more precise representation for x to type the function body because, as we have seen before, we not only remember the types of expressions, but also their values.

After typing the body, we need to distinguish between two different situations: (a) the result type has different representations according to different values of the parameter, and (b) the result type can be uniformly represented involving the parameter. The type of the body ϕ_2 contains this information. We use the auxiliary function $chcs(\phi)$ to get the dimension names in ϕ without corresponding declarations. For example, $chcs(D\langle\mathbf{type_{bool}},\mathbf{type_{int}}\rangle) = \{D\}$. If dimension D is in $chcs(\phi_2)$, then the parameter value is scrutinized, and we build up a choice type. Otherwise, the return type has a uniform representation based on the parameter.

For case (a), consider the `__type_traits` example introduced earlier. We know that if x binds to $A\langle\mathbf{bool},\mathbf{int},\varepsilon\rangle$, then the result type of the **if** expression is $\phi_{\texttt{tt}}$. Based on the rule M-ABS1, the abstraction has the following type.

$$\mathbf{type}^x_{A\langle\mathbf{bool},\mathbf{int},\varepsilon\rangle} \to A\langle\mathbf{type}_{\texttt{\_true\_t}},\mathbf{type}_{\texttt{\_true\_t}},\mathbf{type}_{\texttt{\_false\_t}}\rangle$$

$$\textsc{M-Abs1}\quad \frac{\begin{array}{c}\Gamma;\Psi,(x,\delta_x);\Delta,(x,D\langle\bar{t}\rangle)\vdash_m e^m:\phi_2\\ \phi_1=(D\in chcs(\phi_2)\Rightarrow\delta^x_{D\langle\bar{t}\rangle};\_\Rightarrow\delta^x)\end{array}}{\Gamma;\Psi;\Delta\vdash_m \lambda x:\delta.e^m:\phi_1\to\phi_2}$$

$$\textsc{M-Abs2}\quad \frac{\Gamma;\Psi,(x,\mathbf{code}\ \tau);\Delta\vdash_m e^m:\phi}{\Gamma;\Psi;\Delta\vdash_m \lambda x:\mathbf{code}\ \tau.e^m:\mathbf{code}\ \tau\to\phi}$$

$$\textsc{M-App1}\quad \frac{\Gamma;\Psi;\Delta\vdash_m e_1^m:\delta_\xi\to\phi \qquad \Gamma;\Psi;\Delta\vdash_m e_2^m:\delta_\xi}{\Gamma;\Psi;\Delta\vdash_m e_1^m\ e_2^m:\phi}$$

$$\textsc{M-App2}\quad \frac{\begin{array}{c}\Gamma;\Psi;\Delta\vdash_m e_1^m:\delta^x\to\phi_1 \qquad \Gamma;\Psi;\Delta\vdash_m e_2^m:\phi\\ \xi_1=(\phi=\delta_y\Rightarrow y;\phi\equiv\delta_\xi\Rightarrow\xi)\end{array}}{\Gamma;\Psi;\Delta\vdash_m e_1^m\ e_2^m:[\xi_1/x]\phi_1}$$

$$\textsc{M-App3}\quad \frac{\Gamma;\Psi;\Delta\vdash_m e_1^m:\delta^x_{D\langle\bar{t}\rangle}\to\phi \qquad \Gamma;\Psi;\Delta\vdash_m e_2^m:\delta_{t_i}}{\Gamma;\Psi;\Delta\vdash_m e_1^m\ e_2^m:\lfloor[t_i/x]\phi\rfloor_{D.i}}$$

$$\textsc{M-App4}\quad \frac{\Gamma;\Psi;\Delta\vdash_m e_1^m:\phi' \qquad \Gamma;\Psi;\Delta\vdash_m e_2^m:\delta_{C\langle\bar{\xi}\rangle} \qquad \phi'=\delta^x_{D\langle\bar{t}\rangle}\to\phi}{\Gamma;\Psi;\Delta\vdash_m e_1^m\ e_2^m:C\langle\overline{\lfloor[\xi_i/x]\phi'\rfloor_{D.\xi_i}}\rangle}$$

$$\textsc{M-App5}\quad \frac{\begin{array}{c}\Gamma;\Psi;\Delta\vdash_m e_1^m:\delta^x_{D\langle\bar{t}\rangle}\to\phi\\ \Gamma;\Psi;\Delta\vdash_m e_2^m:\delta_y \qquad (y,C\langle\bar{t}\rangle)\in\Delta\end{array}}{\Gamma;\Psi;\Delta\vdash_m e_1^m\ e_2^m:C\langle\overline{\lfloor\phi\rfloor_{D.i}}\rangle}$$

$$\textsc{M-App6}\quad \frac{\begin{array}{c}\Gamma;\Psi;\Delta\vdash_m e_1^m:\mathbf{code}\ \tau_1\to\phi\\ \Gamma;\Psi;\Delta\vdash e_2^m:\mathbf{code}\ \tau_2 \qquad \tau_1\equiv\tau_2\end{array}}{\Gamma;\Psi;\Delta\vdash e_1^m\ e_2^m:\phi}$$

Figure 6: Typing rules for meta-abstraction and meta-application

For case (b), consider the following expression.

$$mf = \lambda x:\mathbf{type}.\prec \lambda y:x.y \succ$$

The body of the outer lambda expression has the type $\mathbf{code}\ (x \to x)$, and the type for mf is $\mathbf{type}^x \to \mathbf{code}\ (x \to x)$, which precisely captures the type for the function mf. The argument type denotes that the function only accepts object types, and for any object type x, the result type is $\mathbf{code}\ (x \to x)$.

Based on the forms of the argument type of the function and the type of the argument, there are many cases for metafunction applications. If the argument type matches the type of the argument, including the meta-annotation and the value, the result type of the application is the return type of the function (rule M-APP1). If the argument type has the form δ^x, then for any argument of the type δ, we substitute the value of the argument for x in the return type of the function (rule M-APP2). Continuing with the mf example, if the metaexpression g has the type $D\langle\mathbf{type_{int}},\mathbf{type_{bool}}\rangle$, then the application $mf\ g$ has the type $\mathbf{code}\ (D\langle\mathbf{int},\mathbf{bool}\rangle \to D\langle\mathbf{int},\mathbf{bool}\rangle)$. Note the type equivalence relation in M-APP2. In this example, we have $D\langle\mathbf{type_{int}},\mathbf{type_{bool}}\rangle \equiv \mathbf{type}_{D\langle\mathbf{int},\mathbf{bool}\rangle}$.

The typing relation becomes more interesting when the return type of the function depends on the value to the function. In this case, we have to distinguish three different cases: (a) the argument evaluates to a single value that selects a specific return type, (b) the argument evaluates to some unknown value, and the return type becomes dependent on the value of the argument, and (c) the argument evaluates to a variable, and the dependency structure transfers to the argument variable and the new return type.

We represent each case with a specific rule. Rule M-APP3 deals with case (a). We first substitute the value t_i for x in the return type ϕ and select the ith alternative in the substituted return type. For the function `__type_traits`, if the metaexpression g has the type $\mathbf{type}_{\mathbf{int}}$, then the second alternative of the return type, `_true_t`, will become the result type of the application `__type_traits` g.

Rule M-APP4 deals with case (b). When the argument is also a choice between different alternatives, then the result type is a choice depending on the argument. We know that under each different argument, we have a particular return type, and we put them all as alternatives in the argument choice. Note the minor difference between the tag selections in rules M-APP3 and M-APP4. In rule M-APP3, we use index i to get the ith alternative of the return type, while in M-APP4, we use each ξ_i to select from the whole function type. This difference originates from the fact that we don't know which alternative will be chosen in the return type for each ξ_i in advance. Again, for the `__type_traits` function, assuming that g has the type $D\langle\mathbf{type}_{\mathbf{int}},\mathbf{type}_{\mathbf{bool}}\rangle$, the type of the meta-application `__type_traits` g is $D\langle$`_true_t`, `_true_t`$\rangle$.

Of course, the argument expression does not necessarily reduce to a value or a choice—it could be a variable (M-APP5). The overall idea to typing the application in this case is to make the function type depend on the new variable. To realize this, we eliminate the choice in D and create a choice in C, which is the choice y is bound to in Δ. Assuming that the tags in C have the same ordering as those in D, we create of result type by simply replacing D of C in the result type ϕ. To avoid the introduction of a new notation, we select ϕ with each index $D.i$ and these types as alternatives for choice C. Consider, for example, the following expression.

$$\lambda y:\mathbf{type}.\texttt{\_\_type\_traits}\ y$$

Assuming $(y, C\langle\mathbf{int},\mathbf{bool},\varepsilon\rangle) \in \Delta$, the expression `__type_traits` y has the following choice type.

$$C\langle\mathbf{type}_{\texttt{\_true\_t}},\mathbf{type}_{\texttt{\_true\_t}},\mathbf{type}_{\texttt{\_false\_t}}\rangle$$

According to rule M-ABS1, we obtain the following type for the metafunction.

$$\mathbf{type}^{y}_{C\langle\mathbf{bool},\mathbf{int},\varepsilon\rangle} \to C\langle\mathbf{type}_{\texttt{\_true\_t}},\mathbf{type}_{\texttt{\_true\_t}},\mathbf{type}_{\texttt{\_false\_t}}\rangle$$

Note that a metaprogram can manipulate and produce object code as function result, which is handled by the rules M-ABS2 and M-APP6.

Although the typing of abstractions and applications is spread across many different rules, the fact that the premises don't overlap ensures non-ambiguity, that is, it is always possible to decide which rule to apply based on the form of the types. This facilitates a simple implementation of the type system.

Given the typing rules in Figures 4, 5, and 6, the table shown in Figure 7 illustrates how to assign types to terms for the example presented in the introduction, repeated here for convenience.

```
$\mathbf{let\ meta}\ f_0 =$
    $\lambda t:\mathbf{type}.$
        $\mathbf{let\ meta}\ id =$
            $\lambda x:\mathbf{bool}.^{⑥} \prec \lambda y:\mathbf{if}\ {}^{③}x\ \mathbf{then}\ t^{①}\ \mathbf{else\ bool}^{②}.^{④}y \succ^{⑤}$
        $\mathbf{in}$
            $\prec (\sim^{⑧}(id\ \mathbf{false})^{⑦})\ 1^{⑨} \succ$
$\mathbf{in}$
    $\mathbf{let\ meta}\ f_1 = \prec \sim(f_0\ \mathbf{int}) \succ \mathbf{in}$
        ...
```

Pos	Rule	Type
①	M-VAR	$\mathbf{type}_t$
②	M-CONST	$\mathbf{type}_{\mathbf{bool}}$
③	M-IF	$D\langle\mathbf{type}_t,\mathbf{type}_{\mathbf{bool}}\rangle$ [4]
④	T-ABS	$D\langle t,\mathbf{bool}\rangle \to D\langle t,\mathbf{bool}\rangle$
⑤	M-CODE	$\mathbf{code}\ (D\langle t,\mathbf{bool}\rangle \to D\langle t,\mathbf{bool}\rangle)$
⑥	M-ABS1	$\mathbf{bool}^{x}_{D\langle\mathbf{true},\mathbf{false}\rangle} \to \mathbf{code}\ (D\langle t,\mathbf{bool}\rangle \to D\langle t,\mathbf{bool}\rangle)$
⑦	M-APP3	$\mathbf{code}\ (\mathbf{bool} \to \mathbf{bool})$
⑧	T-ESP	$\mathbf{bool} \to \mathbf{bool}$
⑨	T-APP	type error

Figure 7: Typing of introductory example

As we can see, our type system is able to catch the type error in the definition of the metafunction.

3.4 Properties of the type system

We first show that the type system is sound by relating the typing rules to the reduction relation introduced in Section 2.2. This property can be proved by showing that the type system has the preservation and progress properties.

For the following theorem, we say that an expression e is "proper" if it doesn't contain metaexpressions. Specifically, annotations of object abstractions are object types.

THEOREM 1 (Progress).

- *If e_1 is closed and $\Gamma;\Psi;\Delta \vdash_o e:\tau$, then e_1 is a proper object code, or there is some e_2 such that $e_1 \longmapsto e_2$.*
- *If e^m_1 is closed and $\Gamma;\Psi;\Delta \vdash_m e^m_1:\phi$, then e^m_1 is in normal form, or there is some e^m_2 such that $e^m_1 \longmapsto e^m_2$.*

PROOF The proof is based on an induction of the typing derivations. Observe that, although the type definitions are sophisticated, the fact still holds that if e^m is of type $\phi_1 \to \phi_2$ then e^m is an abstraction. □

THEOREM 2 (Preservation).

- *If $\Gamma;\Psi;\Delta \vdash_o e_1:\tau$ and $e_1 \longmapsto e_2$, then $\Gamma;\Psi;\Delta \vdash_o e_2:\tau$.*
- *If $\Gamma;\Psi;\Delta \vdash_m e^m_1:\phi$ and $e^m_1 \longmapsto e^m_2$, then $\Gamma;\Psi;\Delta \vdash_m e^m_2:\phi$.*

The proof of this theorem is significantly more involved than that for STLC [24] because of the complicated types and typing rules. The proof is based on an induction over the typing derivation. The proof strongly relies on the following lemmas, whose proofs follow a standard process and will be omitted here.

LEMMA 1 (Substitution).
If $\Gamma;\Psi,(x,\delta_x);\Delta \vdash_m e^m:\phi$ and $\Gamma;\Psi;\Delta \vdash_m e^m_1:\delta_\xi$, then $\Gamma;\Psi;\Delta \vdash_m [e^m_1/x]e^m:[\xi/x]\phi$.

LEMMA 2 (Selection).
If $\Gamma;\Psi;\Delta,(x,D\langle\bar{t}\rangle) \vdash_m e^m:\phi$, then $\Gamma;\Psi;\Delta \vdash_m [t_i/x]e^m:\lfloor\phi\rfloor_{D.t_i}$ where $t_i \neq \varepsilon$.

PROOF SKETCH OF THEOREM 2 With Lemma 1 and Lemma 2, we can prove M-APP1 through M-APP5 based on an induction over typing derivations. The proof for the other rules follow a standard process. □

Combining Theorems 1 and 2, we obtain the following safety property.

THEOREM 3 (Safety).

- *If $\Gamma;\Psi;\Delta \vdash_o e_1:\tau$ and $e_1 \longmapsto^* e_2$, then $\Gamma;\Psi;\Delta \vdash_o e_2:\tau$ and either e_2 is a proper object code, or there is some e_3 such that $e_2 \longmapsto e_3$.*

[4] Assume x binds to $D\langle\mathbf{true},\mathbf{false}\rangle$ in Δ.

- *If* $\Gamma;\Psi;\Delta \vdash_m e_1^m : \phi$ *and* $e_1^m \longmapsto^* e_2^m$, *then* $\Gamma;\Psi;\Delta \vdash_m e_2^m : \phi$ *and either* e_2^m *is a normal form, or there is some* e_3^m *such that* $e_2^m \longmapsto e_3^m$.

In the theorem, $e_1^m \longmapsto^* e_2^m$ denotes the reduction of e_1^m to e_2^m in an arbitrary number of steps.

There is an important difference between Theorem 1 and the similar theorems in [14, 22]. While well-typed metaprograms may get stuck in previous systems, this will never happen in our type system. *All well-typed metaprograms will reduce to metavalues.* This fact makes our type system more strict and enables the earlier detection of type errors.

Another important difference involves the type preservation theorem. In [22] the type for a metaprogram changes as metaevaluation progresses. In contrast, in our type system a metaexpression will retain its type. From this property it follows that if a metaexpression evaluates to an injected object program, we can type that object program without having to generate it. This observation is captured in the following lemma.

LEMMA 3 (Early Typing). *If* e^m *is a closed metaexpression with* $e^m \longmapsto^* \prec e \succ$ *and* $\Gamma;\Psi;\Delta \vdash_m e^m : \textbf{code}\ \tau$, *then* $\Gamma;\Psi;\Delta \vdash_o e : \tau$.

PROOF Follows from Theorem 3 with M-CODE being the last rule in the typing derivation. □

For open metaexpressions there is an intimate connection between the typing relation and the reduction relation in our type system, which is expressed in the following theorem.

THEOREM 4 (Instantiation Preserves Typing). *Given* e_1^m *and* e_2^m *with* $\Gamma;\Psi;\Delta \vdash_m e_1^m : \phi_1$ *and* $\Gamma;\Psi;\Delta \vdash_m e_2^m : \phi_2$. *If* $e_1^m\ e_2^m \longmapsto^* e_3^m$ *with* $\Gamma;\Psi;\Delta \vdash_m e_3^m : \phi_3$, *then* $\Gamma;\Psi;\Delta \vdash_m e_1^m\ e_2^m : \phi'$ *with* $\phi_3 \equiv \phi'$.

PROOF Take e_1^m to be $e_1^m\ e_2^m$ and e_2^m to be e_3^m in Theorem 3. □

The fact that we can type individual expressions and put them together helps to provide a modular type system. As a corollary, we obtain the guarantee that well-typed metaprograms can only generate well-typed object programs.

COROLLARY 1. *Given any well-typed metaprogram* e_1^m, *for any instantiation argument* e_2^m *whose type matches the argument type of* e_1^m *the generated object program* $e_1^m\ e_2^m$ *is well typed.*

This is a desirable property that helps compilers to delimit the source of type errors and helps address many usability problems with C++ templates.

4. Implementation

This section presents a unification-based implementation of the type system. An implementation both performs type checking and constructs Δs that control how choice types are created. Although object expressions are annotated with type information, type inference is sometimes needed to perform full type checking. For example, given the following expression

$$f = \lambda t\colon \textbf{type}.\prec (\lambda x\colon t.x)\ \texttt{1} \succ$$

the parameter x has the type t, for which we only know it must be a type. The inner abstraction thus has the type $t \to t$. Applying it to `1` requires us to solve the unification problem between t and **int**, the type of `1`. Here we get a unifier $\{t \mapsto \textbf{int}\}$. However, in general we need a variational unification algorithm because, as we have seen before, type annotations can be variational. We use the unification algorithm developed in [7] to unify variational types, which is proved to be sound, complete, and most general. More specifically, we write $\theta = \mathit{vunify}(\tau_1, \tau_2)$ to denote the unifier for τ_1 and τ_2. If the unification succeeds, θ stores the most general unifier.

In the above example, the type variable t is unified to **int**, which implies that the instantiation argument to f should only be **int**. Instantiations with other arguments will lead to type errors in generated object programs. This phenomenon indicates that we provide a way to potentially infer the constraints of template parameters. We may extend the type system to deal with a richer set of constraint forms, and thus support to infer concepts, rather than specify concepts for parameters.

For brevity, Figure 8 presents the implementation of only the most important rules. The implementation for other rules can be derived similarly. We use *infero* and *inferm* to implement the typing relation for the object language and metalanguage, respectively. Both functions have the same signature except that *infero* returns object types τ while *inferm* returns metatypes ϕ. Each function takes four arguments, Γ, Ψ, Δ, and an expression e or e^m, and returns a result type, a unifier, and a Δ, which records how variables are compared with values.

In the implementation, we use several auxiliary functions to decompose metatypes. In particular, for a given type $\phi = \delta^x_{D\langle \bar{t} \rangle}$, we use $\urcorner\, \phi$, $\uparrow \phi$, and $\downarrow \phi$ to get δ, x, and $D\langle \bar{t} \rangle$, respectively. The expression assert(*cond*) terminates the inference process when the condition *cond* doesn't hold. Moreover, Δ is handled qwuite differently in the implementation of the typing rules. When specifying rules, we assume variables are already bound to choices and corresponding tags. However, in the implementation we need to generate them. There are two places where Δ is updated.

First, when we encounter a condition of the form $x = v$ in a conditional, we update the binding information for x. This is realized by adding $(x, C\langle v, \varepsilon \rangle)$ to Δ when $x \notin \textbf{dom}(\Delta)$, where C is a fresh dimension name. Otherwise, if $(x, C\langle \bar{t}, \varepsilon \rangle) \in \Delta$, we extend $C\langle \bar{t}, \varepsilon \rangle$ to $C\langle \bar{t}, v, \varepsilon \rangle$. Note that we always keep the default tag ε at the end of the tag list. For the `__type_traits` example introduced in Section 2.2, Δ first has no binding information for the variable x. When $x = \textbf{bool}$ is visited, $(x, A\langle \textbf{bool}, \varepsilon \rangle)$ is added to Δ, where we assume A is the fresh choice. Later when $x = \textbf{int}$ is encountered, $(x, A\langle \textbf{bool}, \varepsilon \rangle)$ changes to $(x, A\langle \textbf{bool}, \textbf{int}, \varepsilon \rangle)$.

Second, when a function whose argument type is a choice is applied to a variable in the metalanguage, we have to extend Δ to add binding information for the variable. In Figure 8, this is handled by the code with the comment M-APP5. To realize this, we generate a fresh choice (C in the implementation) and associate the variable (y) with the new choice and the tags ($\bar{t}$) in the argument type. We implement this by extending Δ_2 with the binding information $(y, C\langle \bar{t} \rangle)$. Note that there should be no binding information for y in Δ_2 yet. Otherwise, the type of y would not be of the form δ_y but of some more specific form $\delta_{A\langle \bar{\xi} \rangle}$. Consider, for example the expression $\lambda y : \textbf{type}.$`__type_traits` y. The subexpression `__type_traits` y has the following choice type.

$$C\langle \textbf{type}_{\texttt{\_true\_t}}, \textbf{type}_{\texttt{\_true\_t}}, \textbf{type}_{\texttt{\_false\_t}} \rangle$$

C is a fresh dimension name, and the resulting choice environment contains the binding information $(y, C\langle \textbf{bool}, \textbf{int}, \varepsilon \rangle)$.

The implementation for meta-abstractions has also changed. The condition for deciding whether the argument type contains a choice differs from that in the typing rules. If, after typing the body, Δ contains binding information for the parameter x, then the argument type contains a choice. Based on the implementation, the expression $\lambda y : \textbf{type}.$`__type_traits` y has the type

$$\textbf{type}^y_{C\langle \textbf{bool}, \textbf{int}, \varepsilon \rangle} \to C\langle \textbf{type}_{\texttt{\_true\_t}}, \textbf{type}_{\texttt{\_true\_t}}, \textbf{type}_{\texttt{\_false\_t}} \rangle$$

The implementation is correct in the sense that it is both sound and complete with respect to the type system. We capture these properties in following theorems.

```
$infero\colon \Gamma \times \Psi \times \Delta \times e \to \tau \times \theta \times \Delta$
$infero(\Gamma, \Psi, \Delta, \lambda x\colon e^m.e) =$
  $(\phi, \theta, \Delta_1) \leftarrow inferm(\Gamma, \Psi, \Delta, e^m)$
  assert( $\Lsh\phi = \mathbf{type}$ )
  $\tau \leftarrow \downarrow\phi$
  $(\tau', \theta', \Delta_2) \leftarrow infero(\theta(\Gamma, (x, \tau)), \Psi, \Delta_1, e)$
  return $(\theta'(\tau' \to \tau), \theta', \Delta_2)$
$infero(\Gamma, \Psi, \Delta, e_1\ e_2) =$
  $(\tau_1, \theta_1, \Delta_1) \leftarrow infero(\Gamma, \Psi, \Delta, e_1)$
  $(\tau_2, \theta_2, \Delta_2) \leftarrow infero(\Lambda, \theta_1\Gamma, \Psi, \Delta_1, e_2)$
  $\theta \leftarrow vunify(\theta_2\tau_1, \tau_2 \to a)$        {- a is a fresh type variable -}
  return $(\sigma a, \theta \circ \theta_2 \circ \theta_1, \Delta_2)$


$inferm\colon \Gamma \times \Psi \times \Delta \times e^m \to \phi \times \theta \times \Delta$
$inferm(\Gamma, \Psi, \Delta, \lambda x\colon \delta.e^m) =$
  $(\phi, \theta, \Delta_1) \leftarrow inferm(\Gamma, (\Psi, (x, \delta_x)), \Delta)$
  if $(x, D\langle\bar{t}\rangle) \in \Delta_1$
    $\phi' = \delta^x_{D\langle\bar{t}\rangle} \to \phi$
  else
    $\phi' = \delta^x \to \phi$
  return $(\phi', \theta, \Delta_1)$
$inferm(\Gamma, \Psi, \Delta, \mathbf{if}\ \ x = v\ \ \mathbf{then}\ \ e^m_2\ \ \mathbf{else}\ \ e^m_3) =$
  $(\phi_x, \_, \_) \leftarrow inferm(\Gamma, \Psi, \Delta, x)$
  $(\phi_v, \_, \_) \leftarrow inferm(\Gamma, \Psi, \Delta, v)$
  assert( $\Lsh\phi_x = \Lsh\phi_v$ )
  if $(x, D\langle\bar{t}, \varepsilon\rangle) \in \Delta$
    $\Delta_1 = \Delta \backslash \{(x, D\langle\bar{t}, \varepsilon\rangle)\} \cup \{(x, D\langle\bar{t}, v, \varepsilon\rangle)\}$
  else
    $\Delta_1 = \Delta \cup \{(x, D\langle v, \varepsilon\rangle)\}$              {- D is a fresh choice -}
  $(\phi_2, \theta_2, \Delta_2) \leftarrow inferm(\Gamma, \Psi, \Delta_1, e^m_2)$
  $(\phi_3, \theta_3, \Delta_3) \leftarrow inferm(\theta_2\Gamma, \Psi, \Delta_2, e^m_3)$
  return $(D\langle\phi_2\rangle \lozenge \phi_3, \theta_3 \circ \theta_2, \Delta_3)$
$inferm(\Gamma, \Psi, \Delta, e^m_1\ e^m_2) =$
  $(\phi \to \phi_1, \theta_1, \Delta_1) \leftarrow inferm(\Gamma, \Psi, \Delta, e^m_1)$
  $(\phi_2, \theta_2, \Delta_2) \leftarrow inferm(\theta_1\Gamma, \Psi, \Delta_1, e^m_2)$
  if $\phi \to \phi_1 = \phi_2$                                  {- M-APP1 -}
    return $(\phi_1, \theta_2 \circ \theta_1, \Delta_2)$
  assert( $\Lsh\phi = \Lsh\phi_2$ )
  $x \leftarrow \uparrow\phi$
  if $\downarrow\phi$ is empty                                 {- M-APP2 -}
    return $([\phi_2/x]\phi_1, \theta_2 \circ \theta_1, \Delta_2)$
  $D\langle\bar{t}\rangle \leftarrow \downarrow\phi$
  if $\downarrow\phi_2$ is a value $t_i$                         {- M-APP3 -}
    return $(\lfloor[t_i/x](\phi \to \phi_1)\rfloor_{D.t_i}, \theta_2 \circ \theta_1, \Delta_2)$
  if $\downarrow\phi_2$ is $C\langle\overline{u}\rangle$                              {- M-APP4 -}
    return $(C\langle\overline{\lfloor[u_i/x](\phi \to \phi_1)\rfloor_{D.u_i}}\rangle, \theta_2 \circ \theta_1, \Delta_2)$
  if $\downarrow\phi_2$ is a variable $y$                       {- M-APP5 -}
    $\Delta_3 \leftarrow \Delta_2 \cup \{(y, C\langle\bar{t}\rangle)\}$               {- C is a fresh choice -}
    return $(C\langle\overline{\lfloor\phi_1\rfloor_{D.i}}\rangle, \theta_2 \circ \theta_1, \Delta_3)$
```

Figure 8: An implementation of the type system

THEOREM 5 (Inference algorithm is sound).

- *If* $(\tau, \theta, \Delta) = infero(\Gamma, \Psi, \varnothing, e)$*, then* $\theta\Gamma; \Psi; \Delta \vdash_o e : \tau$.
- *If* $(\phi, \theta, \Delta) = inferm(\Gamma, \Psi, \varnothing, e^m)$*, then* $\theta\Gamma; \Psi; \Delta \vdash_m e^m : \phi$.

THEOREM 6 (Inference algorithm is complete and principal).

- *If* $\theta\Gamma; \Psi; \Delta \vdash_o e : \tau$*, then* $(\tau', \theta', \Delta') = infero(\Gamma, \Psi, \varnothing)$ *and* $\theta = \theta_1 \circ \theta'$ *for some mapping* θ_1*,* $\tau = \theta_2\tau'$ *for some* θ_2 *and* $\Delta = \Delta'$ *if the fresh dimension for x is D in* Δ' *when* $(x, D\langle\bar{t}\rangle) \in \Delta$.
- *If* $\theta\Gamma; \Psi; \Delta \vdash_m e : \phi$*, then* $(\phi, \theta', \Delta') = inferm(\Gamma, \Psi, \varnothing)$ *and* $\theta = \theta_1 \circ \theta'$ *for some mapping* θ_1 *and* $\Delta = \Delta'$ *if the fresh dimension for x is D in* Δ' *when* $(x, D\langle\bar{t}\rangle) \in \Delta$.

We have implemented a prototype of the inference algorithm in Haskell. We performed an initial evaluation of our approach by type checking part of the Standard Template Library (STL).[5] We first manually translated the source code from the files `type_traits.h`, `stl_iterator_base.h`, and `stl_algo.h` and their dependent header files into Garcia's calculus extended by other basic types. Since our type system doesn't support partial specializations, we omitted the code introduced by the compilation conditional `__STL_CLASS_PARTIAL_SPECIALIZATION`. We also simplified loop statements to **if** statements in the translated source code. The translation results in about 3000 lines of source code. Type checking each function is finished within several tenth seconds, which demonstrates the efficiency of our approach. Our type checking result doesn't reveal any type error in the source code, which implies that STL without partial specializations shouldn't be blamed as sources for type errors.

5. Discussion

In this section, we investigate the expressiveness of our type system and its ability to guarantee the type safety of object programs.

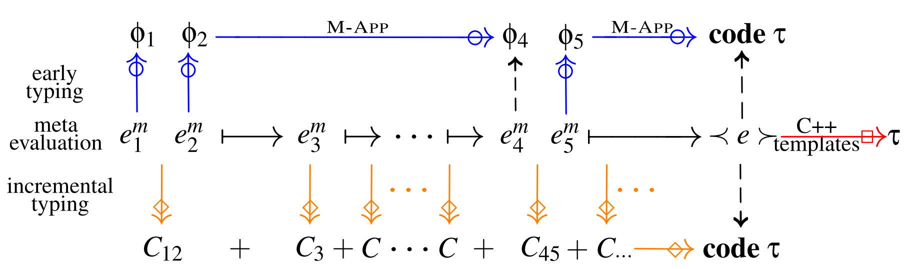
There are two principally different approaches to type checking meta-applications of the form $e^m_1\ e^m_2$ that evaluate to an object program. On the one hand, one can evaluate the application to generate an object program, which is then type checked. Both, the C++ standard and Garcia's work [14] follow this approach. Alternatively, one can try to type check the metaexpressions e^m_1 and e^m_2 and obtain the type for the object program through the application rule for $e^m_1\ e^m_2$. The type system presented in this paper realizes this approach.

If $e^m_1\ e^m_2$ doesn't evaluate to an object program, the first approach is unable to type it whereas the second approach can still determine its type. Miao and Siek's approach falls in between these two approaches. Figure 9a illustrates the typing processes for the different type systems for typing the metaexpression $e^m_1\ e^m_2\ e^m_5$. In the figure, we use blue arrows (also annotated with circles), orange arrows (also annotated with diamonds) and red arrows (also annotated with squares) to denote the typing relation of our type system, Miao and Siek's type system, and C++ templates, respectively. Dashed arrows denote that the typing relation holds but no real computation is needed. This means there are two ways to type, for example, e^m_4. On the one hand, we can use the blue arrows to obtain the types for e^m_1 and e^m_2 and derive the result through the M-APP rule. On the other hand, we can obtain e^m_4 through metareduction, and then type e^m_4 to get ϕ_4. The dashed arrows express that the results obtained through our approach are correct and can be achieved earlier without metareduction. Finally, we use $\mapsto$ arrows to represent individual steps of the metaevaluation.

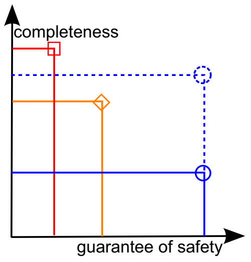
Our type system types each subexpression once and assembles the results employing the application rules twice. In the standard C++ implementation, the typing doesn't happen until the object program has been generated. Miao and Siek's type system will collect initial type constraints from the expression $e^m_1\ e^m_2$ and also additional constraints from each intermediate result. All type constraints will be combined to compute the type for the generated program.

The employed type checking strategy has a huge impact on when type errors are caught and where they are reported. For example, we can envision that in $e^m_1\ e^m_2\ e^m_5$, e^m_1 is a library function and e^m_2 and e^m_5 are the instantiation arguments of e^m_1. Based on

[5] `http://www.sgi.com/tech/stl/index.html`



(a) The typing process for the expression $e_1^m\ e_2^m\ e_5^m$



(b) The completeness/safety trade-off

Figure 9: A comparison of different type systems

Theorem 4 and Corollary 1, our type system can detect the potential type errors in e_1^m. When e_1^m is well typed, we can simply check if the types of e_2^m and e_5^m match the argument type of e_1^m to detect type errors. Thus, type errors within the library will not leak into the client side, and we can easily determine the location of type errors. The C++ templates approach represents the other extreme of handling type errors, where very little type checking of the library code is done, effectively delegating much of type error detection to generated programs. Thus, it is very hard to precisely locate the source of type errors. Library functions are always blamed for type errors, although they are usually caused by the client programs. A problem with Miao and Siek's type system is that it is hard to reason about the detection of type errors, because type checking happens incrementally. It is not clear when there is enough type information to detect type errors. Although it usually detects the type errors earlier than the C++ templates approach, the approach is unlikely to detect type errors without invoking any metaevaluations.

In comparison with other type systems, our type system has the following advantages. First, it improves the usability of C++ templates. It can guarantee the type safety of library code, locate type errors more precisely, and thus offers the opportunity to generate better error messages. Second, our type system is more efficient than other approaches. Since metafunctions are usually complicated, the need for repeated type checking is a potential source of inefficiency. While our type system needs to type each metafunction only once, other type systems have to type them whenever they are instantiated and object programs are generated.

A drawback of our type system is that it lacks the completeness property. There are two main limitations that prevent our type system from being complete. First, as already discussed in Section 3.3, when typing metaconditionals we only consider conditions of the form $x = v$. Thus, we can only deal with generic definitions and template specializations, but not partial specializations [32]. As an example, consider the following metafunction, which decides whether a given type is a pointer type.

$$\texttt{is\_pointer} = \lambda x\colon \textbf{type}.\textbf{if}\ \texttt{isPtr}(x)\ \textbf{then true else false}$$

We assume that `isPtr` is a function to decide, at compile time, whether or not a type is a pointer. This is, in fact, the case for any C++ compiler. Such a predicate must also be available to our type system if we want to extend the type definition to represent pointer types. To support such partial specializations, we have to extend tag selection to predicates. Currently, we only support tag selection with exact matching and default tags. With an extension by a predicate `isPtr`, we could assign the following type to the above metafunction.

$$\textbf{type}^x_{D\langle\texttt{isPtr(x)},\varepsilon\rangle} \to D\langle\textbf{bool}_{\textbf{true}}, \textbf{bool}_{\textbf{false}}\rangle$$

Second, we use types to represent static computations. Since the C++ template system is Turing complete, we obviously can't represent all the functions. Consider, for example, the following metafunction that removes all pointers from types. Here we assume `rmPtr` is a function available to C++ compilers, and thus also to our type system.

$$\begin{aligned}&\texttt{rm\_pointer} = \lambda x\colon \textbf{type}.\\ &\quad \textbf{if}\ \texttt{isPtr}(x)\ \textbf{then}\ \texttt{rm\_pointer}(\texttt{rmPtr}(x))\ \textbf{else}\ x\end{aligned}$$

Although we know that this function terminates and can potentially represent it using the following type, it's unclear about how to construct this type.

$$\textbf{type}^x_{D\langle\texttt{isPtr(x)},\varepsilon\rangle} \to D\langle\textbf{type}_{\texttt{rmAllPtr}(x)}, \textbf{type}_x\rangle$$

We leave this as a challenge for future work.

Figure 9b visualizes the trade-off between completeness and type-safety guarantees for different type systems. Note that we use the same coloring scheme as in Figure 9a. While C++ templates can deal with all template definitions, they provide the weakest safety guarantees. Our current type system can only handle template generic definitions with specializations, and is thus the least complete, but it provides the strongest safety guarantees. Miao and Siek's type system lies between the two. Note, however, that with extended support for predicate-based tag selection, we can substantially expand the expressiveness of our approach. Such an extension promises to cover the area enclosed by the dashed blue lines.

6. Related Work

We will first discuss the related work that is directly concerned with the typing problem for C++ templates. After that we also address the question of metaprogram safety and usability. Since we use values in our type system, we additionally discuss briefly type systems that involve dependent typing of some sort.

Checking C++ Templates The problem of type checking C++ templates has received quite a lot of attention [9, 14, 15, 19, 22, 26, 28]. To improve the typing of C++ templates, the two-phases type checking model of C++ has to be adjusted [32]. One way to tackle this problem is to turn parameter-dependent expressions into parameter-independent ones as much as possible so that more type checking can be performed during the first phase. Concepts [9, 15] fall under this approach. Concepts attached to templates express the requirements of the template parameters, which give type checker more information about template parameters. For example, if the template parameter `T` has the concept *LessThanComparable*, then the type checker knows that the use of the operator `<` is permitted.

Algorithm specializations in C++ pose a challenge for modular type checking of C++ templates [19]. By reducing the type checking of generics with constraints problem to F^G, Siek and Lumsdaine [30] showed that modular type checking can be achieved. However, integrating specializations will break this property. To achieve modular type checking, we either have to eliminate the use of specialization, which precludes the selection of most appropriate implementation, or restricts the implementation of specializations, which dramatically impedes the expressiveness. Neither approach seems to be promising. On the other hand, with choice types, our type system doesn't impose any particular requirements on the definitions of the specializations with respect to generic definitions. Moreover, our type system is modular. This is an important step towards Stroustrup's goal of enforcing type safety of C++ programs [33].

For a more rigorous study of C++ templates metaprogramming, Garcia and Lumsdaine [14] analyzed the capabilities of C++ and proposed a calculus for modeling metaprogramming, focusing on type reflection. However, like C++ the calculus only guarantees that the produced object programs are well-formed but not well-typed.

Miao and Siek improved this situation using incremental typing [22], with the ideas from gradual typing [29, 31]. Unlike C++ compilers that separate metaevaluation and type checking, these phases are instead intertwined in their type system. Whenever one more step of metaevaluation has been completed, newly available type constraints are merged into the constraints previously collected. This process continues until a type error is caught or the metaevaluation is complete. The advantage of this approach is that the type error may be caught earlier.

The most important difference between their work and ours is the representation of uncertain type information and its ramifications. When an expression has potentially several types, they use an existential type to subsume these, while we put all the types into a choice. Their approach can lead to important information loss, which might delay the detection of type errors. An example that illustrates the difference to our approach was given in Section 1.

The other difference is that choice types are usually carried over to be part of the final types for expressions whereas existential types are gradually eliminated and replaced with more specific types as metaevaluation proceeds. This has two implications. (A) The presence of existential types precludes the type soundness property of the type system because, although an expression may be assigned a type, later metaevaluation may get stuck. In contrast, this doesn't happen in our type system. Well-typed programs will not get stuck. (B) Miao and Siek's type system can't ensure that an object program is well typed until metaevaluation is finished. Therefore, metaprograms will be type checked whenever they are instantiated. In contrast, our type system can determine the type of generated object programs without actually generating and type checking it, because a metaprogram is type checked once, and based on that type, the types for generated object programs can be derived.

Metaprogramming Safety and Usability There is a tension between expressiveness and safety, such as type safety of metaprogramming [36]. C++ template metaprograming [1, 3] and Template Haskell [27] represent the position that supports maximum expressiveness. C++ templates provide static type reflection [13] by allowing metaprograms to inspect type structures through specializations. Template Haskell allows the interpreter to be invoked to generate object code that is spliced into the program currently being compiled. However, either approach can provide very few type safety guarantees for the generated object programs. MetaML [35] and MacroML [12] represent the other extreme by ensuring that generated objects are well typed at the expense of limited expressiveness for the metaprogramming system.

Our approach averts the shortcoming of C++ template metaprogramming by designing a type system for C++ templates that guarantees type safety of object programs without the need to generate them, which helps move C++ closer to an ideal model for metaprogramming: ensuring type safety without sacrificing expressiveness.

Along with increasing recognition of C++ templates, usability problems have become apparent. There have been several approaches addressing this issue from different angles. For example, Pataki et al. [23] extended STL with iterator adaptors to solve the iterator invalidation problem and to check that the preconditions of certain algorithms are met (for example, that the range passed to `lower_bound` is sorted) . By viewing compiler actions as runtime computations, Porkoláb et al. [25] developed a framework named *Templight* to debug C++ templates. The tool enables programmers to set breakpoints along the instantiation chain and inspect metaprogram information.

Outside of the C++ community, SafeGen [17] and MorphJ [16] support a mechanism for generating new classes from existing classes. These systems ensure that generated programs are well typed through a mechanism similar to C++ concepts. Inoue and Taha [18] theoretically studied the problems involved in verifying multi-staged programs.

Dependent Types Choice types and type indices used in this paper are similar to types found in dependent type systems [2, 11, 20, 21], and so one question is whether one could implement the type system with one of the dependently typed programming languages. However, these approaches suffer from some shortcoming that make them less than ideal for that task. In Cayenne [2], type checking programs always involves runtime computation, which can significantly slow down the type checker, which does not happen in this type system. To get programs type checked in [11, 21], the programmers usually have to provide fancy type annotations or provide proof terms, which is often too much of a burden for programmers, as pointed out by Chlipala [8]. Finally, it seems possible to use GADTs [20], whose constructors may return different types, to simulate the **if** statements in this calculus. However, GADTs require that all branches' return types refine the data type being defined, which can make type information less precise as happens in Miao and Siek's type system [22].

Applications of Choice Types Choice types can be employed for facilitating the efficient type checking for a family of related programs. We have demonstrated this aspect in [6, 7]. Another use of choice types that was presented in this paper is the encoding of uncertainty about type information. As a different application of this aspect, we have developed a method of *counter-factual typing* to precisely locate type errors and suggest type or expression changes to fix type errors [5]. The fundamental idea of counter-factual typing is to assume that principally any constant or variable can be the potential cause for a type errors. Consequently we can assign each such constant or variable a choice type, whose first alternative contains the type under normal type inference and whose second alternative contains the type that the context requires it to be to make the program well typed. When the inference process terminates, constants or variables whose corresponding choices have different alternatives may be blamed for type errors. Combined with a few simple heuristics, counter-factual typing provides precise and concise feedback about type errors.

7. Conclusions

In metaprogramming there is a critical tension between ensuring the type safety of generated object programs and the expressiveness of metaprograms. In this paper, we have developed a type system for a subset of the C++ template system that provides a type safety guarantee without sacrificing expressiveness. We have achieved this by using choice types to represent uncertain type information of metaexpressions. At the same time, we have used very fine-grained types to represent the types involving static computations and have defined a set of typing rules to reason about the uncertain type information. The most significant properties of our type system are that it can type object programs without generating them and that instantiation preserves the typing relation, that is, we can derive the type for metaprograms by only looking at the types of the functions and their arguments. This makes our type system modular.

There are many potential directions for future research. First, we can extend the tag selection of choice types to accommodate the representation of type dependencies with regard to template partial specializations. We would also like to know which instantiation arguments will lead to type-correct object programs for metaprograms that are ill-typed. This will also help to infer the constraints of template parameters, which is currently done with concepts. However, concepts are only informally attached to metaprograms, and they are not verified against the metaprograms. We hope to also address this issue.

Acknowledgments

We thank anonymous PEPM reviewers whose comments improved the presentation of this paper.

References

[1] D. Abrahams and A. Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond (C++ in Depth Series)*. Addison-Wesley Professional, 2004.

[2] L. Augustsson. Cayenne-a language with dependent types. In *ACM Int. Conf. on Functional Programming*, pages 239–250, 1998.

[3] M. H. Austern. *Generic Programming and the STL: Using and Extending the C++ Standard Template Library*. Addison-Wesley Longman, 1998.

[4] H. Barendregt, S. Abramsky, D. M. Gabbay, T. S. E. Maibaum, and H. P. Barendregt. Lambda calculi with types. In *Handbook of Logic in Computer Science*, pages 117–309. Oxford University Press, 1992.

[5] S. Chen and M. Erwig. Counter-Factual Typing for Debugging Type Errors. In *ACM Symp. on Principles of Programming Languages*, 2014. to appear.

[6] S. Chen, M. Erwig, and E. Walkingshaw. An Error-Tolerant Type System for Variational Lambda Calculus. In *ACM Int. Conf. on Functional Programming*, pages 29–40, 2012.

[7] S. Chen, M. Erwig, and E. Walkingshaw. Extending Type Inference to Variational Programs. *ACM Trans. Prog. Lang. Syst.*, 2013. To appear.

[8] A. Chlipala. Ur: statically-typed metaprogramming with type-level record computation. In *ACM Conf. on Programming Languages Design and Implementation*, pages 122–133, 2010.

[9] G. Dos Reis and B. Stroustrup. Specifying C++ concepts. *SIGPLAN Notices*, 41(1):295–308, Jan. 2006.

[10] M. Erwig and E. Walkingshaw. The Choice Calculus: A Representation for Software Variation. *ACM Trans. on Software Engineering and Methodology*, 21(1):6:1–6:27, 2011.

[11] S. Fogarty, E. Pasalic, J. Siek, and W. Taha. Concoqtion: indexed types now! In *Symp. on Partial Evaluation and Program Manipulation*, pages 112–121, 2007.

[12] S. E. Ganz, A. Sabry, and W. Taha. Macros as multi-stage computations: type-safe, generative, binding macros in macroml. *SIGPLAN Notices*, 36(10):74–85, Oct. 2001.

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

[14] R. Garcia and A. Lumsdaine. Toward foundations for type-reflective metaprogramming. In *Int. Conf. on Generative Programming and Component Engineering*, pages 25–34, 2009.

[15] D. Gregor, J. Järvi, J. Siek, B. Stroustrup, G. Dos Reis, and A. Lumsdaine. Concepts: linguistic support for generic programming in C++. In *ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, pages 291–310, 2006.

[16] S. S. Huang and Y. Smaragdakis. Morphing: Structurally Shaping a Class by Reflecting on Others. *ACM Trans. Program. Lang. Syst.*, 33(2):6:1–6:44, Feb. 2011.

[17] S. S. Huang, D. Zook, and Y. Smaragdakis. Statically Safe Program Generation with Safegen. In *Int. Conf. on Generative Programming and Component Engineering*, pages 309–326, 2005.

[18] J. Inoue and W. Taha. Reasoning about multi-stage programs. In *European Symposium on Programming*, volume 7211 of *Lecture Notes in Computer Science*, pages 357–376. 2012.

[19] J. Järvi, D. Gregor, J. Willcock, A. Lumsdaine, and J. Siek. Algorithm specialization in generic programming: challenges of constrained generics in C++. In *ACM Conf. on Programming Languages Design and Implementation*, pages 272–282, 2006.

[20] P. Johann and N. Ghani. Foundations for structured programming with gadts. In *ACM Symp. on Principles of Programming Languages*, pages 297–308, 2008.

[21] C. T. McBride. Agda-curious?: an exploration of programming with dependent types. In *ACM Int. Conf. on Functional Programming*, pages 1–2, 2012.

[22] W. Miao and J. G. Siek. Incremental type-checking for type-reflective metaprograms. In *Int. Conf. on Generative Programming and Component Engineering*, pages 167–176, 2010.

[23] N. Pataki, Z. Szgyi, and G. Dévai. Measuring the overhead of c++ standard template library safe variants. *Electron. Notes Theor. Comput. Sci.*, 264(5):71–83, July 2011.

[24] B. C. Pierce. *Types and programming languages*. MIT Press, 2002.

[25] Z. Porkoláb, J. Mihalicza, and A. Sipos. Debugging C++ template metaprograms. In *Int. Conf. on Generative Programming and Component Engineering*, pages 255–264, 2006.

[26] G. D. Reis and B. Stroustrup. A Formalism for C++. Technical report, ISO/IEC SC22/JTC1/WG21, October 2005.

[27] T. Sheard and S. L. Peyton Jones. Template Metaprogramming for Haskell. pages 1–16, 2002.

[28] J. Siek and W. Taha. A semantic analysis of C++ templates. In *European Conf. on Object-Oriented Programming*, pages 304–327, 2006.

[29] J. Siek and W. Taha. Gradual typing for objects. In *European Conf. on Object-Oriented Programming*, pages 2–27, 2007.

[30] J. G. Siek and A. Lumsdaine. Essential language support for generic programming. In *ACM Conf. on Programming Languages Design and Implementation*, pages 73–84, 2005.

[31] J. G. Siek and M. Vachharajani. Gradual typing with unification-based inference. In *Symposium on Dynamic languages*, pages 7:1–7:12, 2008.

[32] B. Stroustrup. *The design and evolution of C++*. ACM Press, 1994.

[33] B. Stroustrup. Evolving a language in and for the real world: C++ 1991-2006. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, HOPL III, pages 4–1–4–59, New York, NY, USA, 2007. ACM.

[34] W. Taha and M. F. Nielsen. Environment classifiers. In *ACM Symp. on Principles of Programming Languages*, pages 26–37, 2003.

[35] W. Taha and T. Sheard. Multi-stage programming with explicit annotations. In *Symp. on Partial Evaluation and Program Manipulation*, pages 203–217, 1997.

[36] T. L. Veldhuizen. Tradeoffs in metaprogramming. In *Workshop on Partial Evaluation and Program Manipulation*, pages 150–159, 2006.