A Semantic Analysis of C++ Templates

Jeremy Siek and Walid Taha
Jeremy.G.Siek@rice.edu, taha@rice.edu

Rice University,
Houston, TX 77005, USA

Abstract. Templates are a powerful but poorly understood feature of the C++ language. Their *syntax* resembles the parameterized classes of other languages (e.g., of Java). But because C++ supports template specialization, their *semantics* is quite different from that of parameterized classes. Template specialization provides a Turing-complete sub-language within C++ that executes at compile-time. Programmers put this power to many uses. For example, templates are a popular tool for writing program generators.

The C++ Standard defines the semantics of templates using natural language, so it is prone to misinterpretation. The meta-theoretic properties of C++ templates have not been studied, so the semantics of templates has not been systematically checked for errors. In this paper we present the first formal account of C++ templates including some of the more complex aspects, such as template partial specialization. We validate our semantics by proving type safety and verify the proof with the Isabelle proof assistant. Our formalization reveals two interesting issues in the C++ Standard: the first is a problem with member instantiation and the second concerns the generation of unnecessary template specializations.

1 Introduction

We start with a review of C++ templates, demonstrating their use with some basic examples. We then review more advanced uses of templates to perform compile-time computations and to write program generators. The introduction ends with an overview of the technical contributions of this paper.

1.1 Template Basics

The following definition is an example of a *class template* that defines a container parameterized on the element type T and length n.

```
template<class T, int n>
class buffer {
  T data[n];
public:
  void set(int i, T v) { data[i] = v; }
  T get(int i) { return data[i]; }
};
```

A *template specialization* provides an alternate implementation of a template for concrete template arguments. For example, the above template is not space-efficient when T=**bool** because a **bool** may be larger than a single bit in C++. The following specialization for element type **bool** uses a compressed representation, dedicating a single bit for each element.

```
template<int n>
class buffer<bool, n> {
  int data[(n + BITS_PER_WORD − 1)/BITS_PER_WORD];
public:
  void set(int i, bool v) { /* complicated bit masking */ }
  T get(int i) { /* complicated bit masking */ }
};
```

The above definition is called a *partial template specialization* because there is still a template parameter left. We refer to <**bool**,n> as the *specialization pattern*. The following is an example of a *full specialization*:

```
template<>
class buffer<bool, 0> {
public:
  void set(int i, bool v) { throw out_of_range(); }
  T get(int i) { throw out_of_range(); }
};
```

When a template is used, C++ performs *pattern matching* between the template arguments and the specialization patterns to determine which specialization to use, or whether a specialization needs to be generated from a class template or a partial specialization. Consider the following program that defines three objects.

```
int main() {
  buffer<int,3> buf1;
  buffer<bool,3> buf2;
  buffer<bool,0> buf3;
}
```

The type buffer<**int**,3> matches neither of the specializations for buffer, so C++ will generate a specialization from the original buffer template by substituting **int** for T and 3 for n. This automatic generation is called *implicit instantiation*. The resulting template specialization is shown below.

```
template<>
class buffer<int,3> {
  int data[3];
public:
  void set(int i, int v);
  int get(int i);
};
```

Note that definitions (implementations) of the members set and get were not generated. Only the declarations of the members were generated. The definition

of a member is generated only if the member is called. So, for example, the following code

```
buf1.get(2);
```

causes C++ to generate a member definition for buffer<**int**,3>::get.

```
template<>
int buffer<int,3>::get(int i) { return data[i]; }
```

The above is an example of a member defined separately from a class.

The type, buffer<**bool**,3> matches the partial specialization, but not full specialization of buffer, so C++ will generate a specialization from the partial specialization.

Last but not least, the type buffer<**bool**,0> type matches the full specialization of buffer, so C++ does not need to generate a specialization.

1.2 Compile-time programming with templates

A small compile-time computation is performed in the buffer<**bool**,n> specialization to compute the length of the data array:

```
int data[(n + BITS_PER_WORD − 1)/BITS_PER_WORD];
```

The ability to pass values as template parameters and to evaluate expressions at compile time provides considerable computational power. For example, the following power template computes the exponent x^n at compile-time.

```
template<int x, int n> struct power {
  static const int r = x ∗ power<x, n − 1>::r;
};
template<int x> struct power<x, 0> {
  static const int r = 1;
};
int array[power<3, 2>::r];
```

The **static** keyword means the member is associated with the class and not with each object. The **const** keyword means the member is immutable.

There are limitations, however, to what kinds of expressions C++ will evaluated at compile time: only arithmetic expressions on built-in integer-like types. There are similar restrictions on the kinds of values that may be passed as template parameters. For example, a list value can not be passed as a template parameter. Fortunately, it is possible to encode data structures as types which can be then be passed as template parameters. The following creates a type encoding for cons-lists.

```
template<class Head, class Tail> struct Cons { };
struct Nil { };
typedef Cons< int, Cons< float, Nil > list_of_types;
```

In general, templates can be used to encode algebraic datatypes [5] such as those found in SML [8] and Haskell [1].

This paper omits value template parameters and compile-time expressions because they are technically redundant: we can encode computations on integer-like values as computations on types. For example, the following types encode natural numbers.

```
template<class T> struct succ { };
struct zero { };
typedef succ<zero> one;
typedef succ< succ<zero> > two;
```

The following is the power template reformulated for types. The definition of mult is left as an exercise for the reader.[1]

```
template<class x, class n> struct power { };

template<class x, class p> struct power<x, succ<p> > {
  typedef mult<x, power<x, p>::r>::r r;
};
template<class x> struct power<x, zero> {
  typedef one r;
};
```

1.3 Metaprogramming with templates

The combination of templates and member functions enables compile-time program generation in C++, often referred to as *template metaprogramming* [3, 4, 17]. Member functions can be used to represent run-time program fragments while templates provide the ability to compose and select fragments. We revisit the power example, but this time as a staged metaprogram that takes n as a compile-time parameter and generates a program with a run-time parameter x.

```
template<class n> struct power {};

template<class p>
struct power< succ<p> > {
  static int f(int x){ return x * power<p>::f(x); }
};
template<> struct power<zero> {
  static int f(int x) { return 1; }
};
int main(int argc, char* argv[]) {
  return power<two>::f(atoi(argv[1])); // command-line input
}
```

The bodies of functions, such as in main and f, contain run-time code. Type expressions, such as power<two> and power<p> represent escapes from the run-time code back into the compile-time level. The power metaprogram is recursive

[1] A C++ expert will notice missing **typename** keywords in our examples. We do this intentionally to avoid confusing readers unfamiliar with C++ with syntactic clutter.

but the generated program is not. The generated program has a static call tree of height 3. An optimizing C++ compiler is likely to simplify the generated program to the following one, but such optimization is not required by the C++ Standard. The compiler is required to preserve the call-by-value semantics when it performs function inlining.

```
int main(int argc, char* argv[]) {
  int x = atoi(argv[1]);
  return x * x;
}
```

The **inline** keyword of C++ does not force inlining. It is only a suggestion to the compiler. The performance of the generated programs is therefore brittle and non-portable. Compilers rarely publicize the details of their inlining algorithm, and the algorithms are heuristic in nature and hard to predict. Furthermore, the inlining algorithm can vary dramatically from one compiler to the next. See [6] for an alternative approach based on macros that guarantees inlining.

The subset of C++ we study in the paper includes just enough features to exhibit both the compile time and run time computations needed to write template metaprograms.

1.4 Contributions

We present the first formal account of C++ templates. We identify a small subset of C++ called C++.T and give a semantics to C++.T by defining:

1. template lookup (Section 3.1)
2. type evaluation (Section 3.2),
3. expression evaluation and well-typed expressions (Section 4), and
4. template instantiation (Section 5).

C++.T includes the partial specialization feature of C++, so template lookup is nontrivial. To maintain a clear focus, C++.T does not include features of C++ that are orthogonal to templates, such as statements, imperative assignment, and object-oriented features such as inheritance.

A C++.T program is "valid" if and only if the template instantiation process succeeds. This definition is unusual because some potentially non-terminating evaluation (type evaluation) is performed as part of determining whether a program is "valid". We show that C++.T is type safe in the sense that if template instantiation succeeds, run-time execution of the program will not encounter type errors(Theorem 1, Section 5.1). We wrote the proof in the Isar proof language [9, 19] and mechanically verified the proof using the Isabelle proof assistant [10]. Due to space considerations, we do not present the Isar proof in this paper but refer the reader to the accompanying technical report [15].

Formalizing C++.T revealed two issues with the C++ Standard:

1. The Standard's rule for member instantiation requires the point of instantiation to come too soon, possibly before the definition of the member. In our

semantics we delay member instantiation to the end of the program, which corresponds to the current practice of the GNU and Edison Design Group C++ compilers.

2. The Standard requires the permanent generation of a template specialization whenever a member is accessed. However, if such a specialization is only needed temporarily, the compiler should be allowed to discard the specialization, analogously to the way procedure activation frames are discarded when a function returns, thereby improving the space-complexity for template programs.

2 Overview of the formalization

The semantics of C++.T includes compile-time and run-time components. The compile-time components concern template lookup, type evaluation, type-checking, and template instantiation, whereas the run-time component concerns expression evaluation.

C++.T contains syntactic categories for types, expressions (or terms), and definitions. A program is a sequence of definitions. We use the metavariable τ for types, e for expressions, v for values, d for definitions, and p for programs.

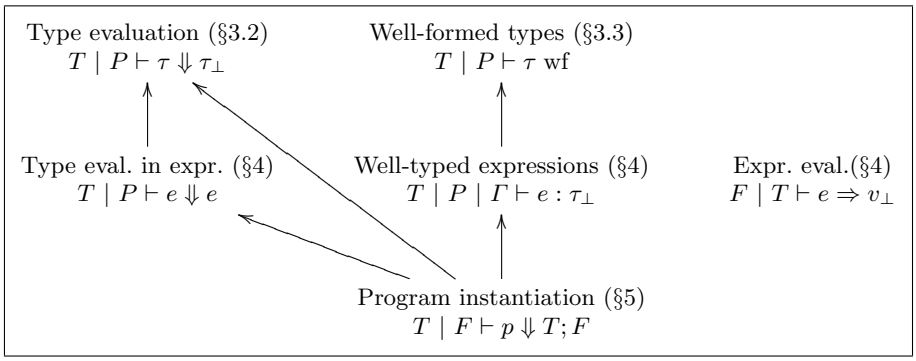

Fig. 1. Semantic judgments and their dependencies.

Fig. 1 shows the semantic judgments for C++.T. The type and expression evaluation judgments are given in a big-step operational style. The variable T stands for a set of template definitions and P for a set of in-scope type parameters. The judgment for program instantiation also serves to define *valid* C++.T programs. The program instantiation judgment recursively processes a program: the first definition in the program is processed and then this judgment is applied again to process the rest of the program. The instantiation process results in updated sets of template and function definitions. The notation $X_\perp$ stands for the disjoint union of X and $\{\perp\}$ where $\perp$ denotes an error. The notation $\lfloor x \rfloor$

injects an element of X into $X_\perp$. The variable Γ is a type assignment for term variables and F is a set of function definitions.

The main theorem, shown below, states that valid programs are type safe. The first premise says that the program successfully instantiates, producing a set of template definitions T and a set of function definitions F. The second and third premise say that the program defines an Main template with a main member function. The fourth premise says that the program terminates with an answer (either an error or a value). The conclusion is that the answer must be a value of the appropriate type, thereby ruling out the error case.

Theorem 1. *(Type Safety) If*

1. $\emptyset \mid \emptyset \mid \emptyset \vdash p \Downarrow T; F$, *and*
2. $\mathit{Main}\langle\rangle\{\mathit{main} : \mathit{int} \to \mathit{int}\} \in T$, *and*
3. $\mathit{Main}\langle\rangle\ \mathit{has}\ \mathit{main(x:int)} \to \mathit{int}\{e\} \in F$, *and*
4. $F \mid T \vdash \mathit{Main}\langle\rangle.\mathit{main(n)} \Rightarrow a$

then there exists v *such that* $a = \lfloor v \rfloor$ *and* $T \mid \emptyset \mid \emptyset \vdash v : \mathit{int}$.

(The type safety theorem is proved in Section 5.) The lemmas in the following sections lead up to this type safety result.

3 Types and Templates

The syntax of types and templates is defined by the following grammar. We use an abstract syntax for the sake of conciseness and readability for those unfamiliar with C++ template syntax.

Abstract syntax of types and templates $\tau \in \mathbb{T}$ $\mathcal{T}$

Type variables	α	$\in$	TyVar
Template names	t	$\in$	TmName
Member names	m, f, a	$\in$	MemName
Type expressions	$\tau \in \mathbb{T}$	$::=$	$\alpha \mid \tau.a \mid t\langle\tau_1..\tau_n\rangle \mid \tau \to \tau \mid \texttt{int}$
Member kind	$\kappa \in \mathcal{K}$	$::=$	$\texttt{fun} \mid \texttt{type}$
Templates	$\mathcal{T}$	$::=$	$t\langle\pi_1..\pi_n\rangle\ \{m : \kappa\ \tau\}$
Type patterns	$\pi \in \Pi$	$::=$	$\alpha \mid t\langle\pi_1..\pi_n\rangle \mid \pi_1 \to \pi_2 \mid \texttt{int}$
Residual types	$r \in \mathbb{R}$	$::=$	$t\langle r_1..r_n\rangle \mid r \to r \mid \texttt{int}$

A type in C++.T can be a type variable, member type access, template identifiers, function type, or `int`. We pick out two subsets of $\mathbb{T}$: type patterns Π and residual types $\mathbb{R}$. Type patterns are types without member type access. Residual types are restricted to template identifiers, functions, and `int`. When applied to closed types, type evaluation produces residual types.

The member access type expression $\tau.a$ refers to a nested type definition. The τ should refer to a template specialization with a member named a. In the concrete syntax of C++, type member access is written $\tau\texttt{::}a$. The template identifier type expression $t\langle\tau_1..\tau_n\rangle$ refers to the specialization of template t for the type arguments $\tau_1..\tau_n$. The function type expression $\tau_1 \to \tau_2$ corresponds to the C++ syntax $\tau_2\texttt{(*)(}\tau_1\texttt{)}$.

There are no variable binders in $\mathbb{T}$ so the set of free type variables (FTV) of a type is simply the collection of type variables occurring in the type. A type is *closed* if it contains no type variables.

The syntax $t\langle\pi_1..\pi_n\rangle\{m : \kappa\ \tau\}$ is used for both class templates and class template specializations. When all of the patterns are variables, the declaration is a class template: $t\langle\alpha_1..\alpha_n\rangle\{m : \kappa\ \tau\}$. When the patterns contain no type variables, then the declaration is a full specialization of a class template: $t\langle\tau_1..\tau_n\rangle\{m : \kappa\ \tau\}$ where $\mathrm{FTV}(t\langle\tau_1..\tau_n\rangle) = \emptyset$. Everything in between corresponds to partial specializations of class templates. When referring to things of the general form $t\langle\pi_1..\pi_n\rangle\{m : \kappa\ \tau\}$ we will use the term "template" even though we ought to say "template or specialization".

We restrict templates to contain just a single member to reduce clutter in the semantics. Expressiveness is not lost because a template with multiple members can always be expressed using multiple templates. In the following example, we split template A into two templates A1 and A1 and change the use of member x in A2 to A1<T>::x.

```
template<class T>
struct A {
  typedef T x;
  typedef foo<x> y;
};
```

$\Longrightarrow$

```
template<class T>
struct A1 {
  typedef T x;
};
template<class T>
struct A2 {
  typedef foo<A1<T>::x> y;
};
```

A type member is written m : `type` τ and is equivalent to the C++ syntax `typedef` τ m`;`. A member function declaration is written f : `fun` $\tau_1 \to \tau_2$ and is equivalent to the C++ syntax `static` τ_2 f`(`τ_1`);`. The definition (implementation) of a member function is written separately and the syntax for that is introduced in Section 4.

We define the following function to return the set of template names from a set of templates.

$$\mathit{names} : \mathsf{Set}\ \mathcal{T} \to \mathsf{Set}\ \mathsf{TmName}$$
$$\mathit{names}(T) \triangleq \{t \mid t\langle\pi_1..\pi_n\rangle\{m : \kappa\ \tau\} \in T\}$$

We also need the notion of when a type is defined and when a type is complete.

Definition 1.

- *A type* τ ***is defined in*** T *iff* $\exists\tau', m, \kappa, \tau''.\tau' \doteq \tau \wedge \tau'\{m : \kappa\ \tau''\} \in T$.
- *A type* τ ***is complete*** *in* T *iff* τ *is defined in* T *and* $\tau \in \mathbb{R}$.

3.1 Template lookup, matching, and ordering

As mentioned in Section 1, template lookup is non-trivial because C++ supports partial specialization. The use of a template resolves to the most specialized template that matches the given template arguments, according to Section [14.5.4.1 p1] of the C++ Standard. So our goal is to define "most specific" and "matches".

Template arguments are matched against the specialization pattern of candidate templates. In the C++ Standard, the matching is called *template argument deduction* (see Section [14.8.2.4 p1] of the C++ Standard). The following defines matching.

Definition 2. *A type τ_1* ***matches*** *a type τ_2 iff there exists a substitution S such that $S(\tau_2) = \tau_1$.*

To define "most specialized" we first need to define the "at least as specialized" relation on types. This relation is defined in terms of matching. (See Sections [14.5.4.2 p1] and [14.5.5.2 p2-5] of the C++ Standard.)

Definition 3. *If τ_1 matches τ_2, then we write $\tau_2 \leq \tau_1$ and say that τ_1 is* ***at least as specialized as*** *τ_2.*

The $\leq$ relation is a quasi-order, i.e., it is reflexive and transitive. If we identify type patterns up to renaming type variables, then we have antisymmetry and the $\leq$ relation is a partial order.

Proposition 1. *If $\tau_1 \leq \tau_2$ and $\tau_2 \leq \tau_1$ then there exists a variable renaming S such that $S(\tau_2) = \tau_1$. A renaming is a substitution that maps variables to variables and is injective.*

We use the following notation $\doteq$ for type pattern equivalence and use it to define a notion of duplicate template definitions.

Definition 4.

- $\pi_1 \doteq \pi_2 \triangleq \pi_1 \leq \pi_2$ *and* $\pi_2 \leq \pi_1$.
- *Template* $\pi_1\{m_1 : \kappa_1\ \tau_1'\}$ *and template* $\pi_2\{m_2 : \kappa_2\ \tau_2'\}$ *are* ***duplicates*** *if* $\pi_1 \doteq \pi_2$.
- *There* ***are no duplicates in*** *T if no two templates in T are duplicates of one another.*

We extend the $\leq$ type relation to templates as follows:

Definition 5. $\pi_1\{m_1 : \kappa_1\ \tau_1\} \leq \pi_2\{m_2 : \kappa_2\ \tau_2\} \triangleq \pi_1 \leq \pi_2$

This extension is a partial order on the set of template definitions in a valid program because we do not allow duplicate templates (duplicates would cause antisymmetry to fail).

Definition 6. *Given a set of template definitions T and the ordering $\leq$, the* ***most specific template****, if it exists, is the greatest element of T, written* $\max T$.

We define the following *lookup* function to capture the rule that the use of a template resolves to the most specific template that matches the given template arguments.

$$\begin{aligned} &\mathit{lookup} : \mathsf{Set}\ \mathcal{T} \times \mathbb{T} \to \mathcal{D}_\perp \\ &\qquad \mathit{lookup}(T, \tau) \triangleq \begin{cases} \lfloor \max\{\pi\{m\} \in T \mid \pi \leq \tau\} \rfloor & \text{if the max exists} \\ \perp & \text{otherwise} \end{cases} \end{aligned}$$

The *inst* function maps a set of templates and a type to the template specialization obtained by instantiating the best matching template from a set of templates.

$$\begin{aligned} &\mathit{inst} : \mathsf{Set}\ \mathcal{T} \times \mathbb{T} \to \mathcal{D}_\perp \\ &\qquad \mathit{inst}(T, \tau) \triangleq \begin{cases} \lfloor \tau\{m : \kappa\ S(\tau')\} \rfloor & \text{where } \mathit{lookup}(T, \tau) = \lfloor \pi\{m : \kappa\ \tau'\} \rfloor \\ & \text{and } S(\pi) = \tau \\ \perp & \text{otherwise} \end{cases} \end{aligned}$$

Definition 7. *A set of types N* instantiates to *a set of templates T* in $T' \triangleq$ *for each type $\tau \in N$, $\mathit{inst}(T', \tau) = \lfloor \tau\{m : \kappa\ \tau'\} \rfloor$ iff $\tau\{m : \kappa\ \tau'\} \in T$.*

The following *lookupmem* function maps from a set of template, a type, and a member name to a type.

$$\begin{aligned} &\mathit{lookupmem} : \mathsf{Set}\ \mathcal{T} \times \mathbb{T} \times \mathsf{MemName} \to (\mathcal{K} \times \mathbb{T})_\perp \\ &\qquad \mathit{lookupmem}(T, \tau, m) \triangleq \begin{cases} \lfloor (\kappa, \tau') \rfloor & \mathit{inst}(T, \tau) = \lfloor \tau\{m : \kappa\ \tau'\} \rfloor \\ \perp & \text{otherwise} \end{cases} \end{aligned}$$

Next we show that member lookup produces closed types. For this lemma we need to define the free type variables of a set of template definitions.

Definition 8. $\mathit{FTV}(T) \triangleq \{\alpha \mid \pi\{m : \kappa\ \tau\} \in T \wedge \alpha \in \mathit{FTV}(\tau) - \mathit{FTV}(\pi)\}$

Lemma 1. *(Member lookup produces closed types.) If $\mathit{lookupmem}(T, \tau, m) = \lfloor (\kappa, \tau') \rfloor$ and $\mathit{FTV}(\tau) = \emptyset$ and $\mathit{FTV}(T) = \emptyset$ then $\mathit{FTV}(\tau') = \emptyset$*

Proof. A straightforward use of Proposition 4.

3.2 Type evaluation

The rules for type evaluation are complicated by the need to evaluate types underneath type variable binders. In the following example, the type `A<A<int>::u>` is underneath the binder for `T` but it must be evaluated to `A<float>`.

```
template<class T>                          template<class T>
struct A {                                 struct A {
  typedef float u;                           typedef T u;
};                                         };
template<class T>                    ⟹    template<class T>
struct B {                                 struct B {
  static int foo(A<A<int>::u> x)             static int foo(A<float> x)
      { return x; }                              { return x; }
};                                         };
```

The need for evaluation under variable binders is driven by the rules for determining the point of instantiation for a template. Section [14.6.4.1 p3] of the C++ Standard [7] says that the point of instantiation for a specialization precedes the first declaration that contains a use of the specialization, unless the enclosing declaration is a template and the use is dependent on the template parameters. In that case the point of instantiation is immediately before the point of instantiation for the enclosing template. In the above example, the type `A<A<int>::u>` is in an instantiation context and does not depend on the template parameter `T`. So we need to instantiate `A<A<int>::u>`, but it must first evaluate to `A<float>` so that we can check whether this type was already instantiated.

The evaluation rules for type expressions are defined below. The rule (C-VarT) says a type variable α evaluates to itself provided α is in scope. Rule (C-MemT1) defines type member access $\tau.a$ analogous to a function call. First evaluate τ. If the result is of the form $t\langle\tau_1..\tau_n\rangle$ and has no free variables, lookup the type definition τ' for member a. The substitution of type arguments $\tau_1..\tau_n$ for template parameters is performed as part of lookup. The member type is evaluated to τ'' and that is the result. An alternative design would perform the lookup and substitution whenever a template identifier such as $t\langle\tau_1..\tau_n\rangle$ is evaluated. We choose to delay the lookup and instantiation to the last possible moment to better reflect the on-demand nature of C++ instantiation.

Rule (C-MemT2) handles the case when the τ in $\tau.a$ evaluates to a type τ' with free variables. In this case the result is just $\tau'.a$. We write explicit rules to capture and propagate all errors so that we can distinguish between a nonterminating type expression and an error. The rest of the rules are straightforward; they simply evaluate the nested types and put the type back together.

Several of the type evaluation rules test if a type contains free variables, which is not a constant-time operation. However, an implementation of type evaluation could keep track of whether types contain any free variables by returning a boolean value in addition to the resulting type.

Type evaluation. $T \mid P \vdash \tau \Downarrow \tau'_\perp$

(C-VarT) $$\frac{\alpha \in P}{T \mid P \vdash \alpha \Downarrow \lfloor \alpha \rfloor}$$

(C-MemT1) $$\frac{T \mid P \vdash \tau \Downarrow \lfloor t\langle \tau_1..\tau_n \rangle \rfloor \quad \bigcup_i \mathrm{FTV}(\tau_i) = \emptyset \quad \mathit{lookupmem}(T, t\langle \tau_1..\tau_n \rangle, a) = \lfloor \mathtt{type}\ \tau' \rfloor \quad T \mid P \vdash \tau' \Downarrow \lfloor \tau'' \rfloor}{T \mid P \vdash \tau.a \Downarrow \lfloor \tau'' \rfloor}$$

(C-MemT2) $$\frac{T \mid P \vdash \tau \Downarrow \lfloor \tau' \rfloor \quad \mathrm{FTV}(\tau') \neq \emptyset}{T \mid P \vdash \tau.a \Downarrow \lfloor \tau'.a \rfloor}$$

(C-TmT) $$\frac{t \in \mathit{names}(T) \qquad \forall i \in 1..n.\ T \mid P \vdash \tau_i \Downarrow \lfloor \tau'_i \rfloor}{T \mid P \vdash t\langle \tau_1..\tau_n \rangle \Downarrow \lfloor t\langle \tau'_1..\tau'_n \rangle \rfloor}$$

(C-ArrowT) $$\frac{T \mid P \vdash \tau_1 \Downarrow \lfloor \tau'_1 \rfloor \quad T \mid P \vdash \tau_2 \Downarrow \lfloor \tau'_2 \rfloor}{T \mid P \vdash (\tau_1 \to \tau_2) \Downarrow \lfloor \tau'_1 \to \tau'_2 \rfloor}$$

(C-IntT) $$\frac{}{T \mid P \vdash \mathtt{int} \Downarrow \lfloor \mathtt{int} \rfloor}$$

Error propagation rules

$$\frac{\alpha \notin P}{T \mid P \vdash \alpha \Downarrow \perp} \qquad \frac{T \mid P \vdash \tau \Downarrow \lfloor t\langle \tau_1..\tau_n \rangle \rfloor \quad \bigcup_i \mathrm{FTV}(\tau_i) = \emptyset \quad \mathit{lookupmem}(T, t\langle \tau_1..\tau_n \rangle, a) = \perp}{T \mid P \vdash \tau.a \Downarrow \perp} \qquad \frac{T \mid P \vdash \tau \Downarrow \lfloor \tau' \rfloor \quad \mathrm{FTV}(\tau') = \emptyset \quad \neg \exists t \tau_1..\tau_n.\tau' = t\langle \tau_1..\tau_n \rangle}{T \mid P \vdash \tau.m \Downarrow \perp} \qquad \frac{T \vdash \tau \Downarrow \perp}{T \vdash \tau.m \Downarrow \perp}$$

$$\frac{t \notin T}{T \mid P \vdash t\langle \tau_1..\tau_n \rangle \Downarrow \perp} \qquad \frac{\exists i \in 1..n.\ T \vdash \tau_i \Downarrow \perp}{T \vdash t\langle \tau_1..\tau_n \rangle \Downarrow \perp} \qquad \frac{\exists i \in 1, 2.\ T \vdash \tau_i \Downarrow \perp}{T \vdash (\tau_1 \to \tau_2) \Downarrow \perp}$$

Proposition 2. *(Properties of type evaluation)*

1. *If* $T \mid P \vdash \tau \Downarrow \lfloor \tau' \rfloor$ *then* $FTV(\tau') \subseteq P$.
2. *If* $T \mid \emptyset \vdash \tau \Downarrow \lfloor \tau' \rfloor$ *then* $\tau' \in \mathbb{R}$.
3. $\exists T P \tau.\ \neg \exists a.\ T \mid P \vdash \tau \Downarrow a$. *For example, let* $T = \{A\langle\alpha\rangle\{x : \mathtt{type}\ A\langle A\langle\alpha\rangle\rangle.x\},\}$, $P = \emptyset$, *and* $\tau = \mathtt{A}\langle \mathtt{int} \rangle.x$. *Then type evaluation does not terminate and so no such a exists.*

3.3 Well-formed types

Well-formed types are types that do not contain out-of-scope type parameters or use undefined template names. The following is the definition of well-formed types.

Well-formed types $T \mid P \vdash \tau \text{ wf}$

$$\frac{\alpha \in P}{T \mid P \vdash \alpha \text{ wf}} \qquad \frac{T \mid P \vdash \tau \text{ wf}}{T \mid P \vdash \tau.a \text{ wf}} \qquad \frac{t \in \mathit{names}(T) \quad \forall i \in 1..n.\ T \mid P \vdash \tau_i \text{ wf}}{T \mid P \vdash t\langle \tau_1..\tau_n \rangle \text{ wf}}$$

$$\frac{T \mid P \vdash \tau_1 \text{ wf} \quad T \mid P \vdash \tau_2 \text{ wf}}{T \mid P \vdash \tau_1 \to \tau_2 \text{ wf}} \qquad T \mid P \vdash \texttt{int} \text{ wf}$$

Proposition 3. *(Properties of well-formed types)*

1. *If* $T \mid P \vdash \tau$ wf *then* $FTV(\tau) \subseteq P$.
2. *If* $T \mid P \vdash \tau$ wf *and* $T \subseteq T'$ *then* $T' \mid P \vdash \tau$ wf.

3.4 Type substitution

A *substitution* is a function mapping type variables to types that acts like the identity function on most of its domain except for a finite number of elements. Substitutions are extended to types with the following definition. The line $S(\alpha) = S(\alpha)$ may look strange, but is not in fact a circular definition. We are given S that is a function on type variables and building a function, also called S, on type expression. The α on the left hand side is viewed as a type expression whereas the α on the right is viewed as a type variable.

Simultaneous substitution on types. $S(\tau) \in \mathbb{T}$

$$\begin{aligned} S(\alpha) &= S(\alpha) \\ S(\tau_1 \to \tau_2) &= S(\tau_1) \to S(\tau_2) \\ S(t\langle \tau_1..\tau_n \rangle) &= t\langle S(\tau_1)..S(\tau_n) \rangle \\ S(\tau.a) &= S(\tau).a \\ S(\texttt{int}) &= \texttt{int} \end{aligned}$$

Proposition 4. $FTV(S(\tau)) = \bigcup_{\alpha \in FTV(\tau)} FTV(S(\alpha))$

Proof. By induction on the structure of τ.

4 Expressions and Functions

The expressions of C++.T include variables, integers, object creation, static member function access, and function application. (In C++ the syntax for static member access is $\tau\texttt{::}f$ and the syntax for object creation is $\tau\texttt{()}$.)

Abstract syntax of expressions. $e \in \mathcal{E}$ $\mathcal{F}$

Expressions	$e \in \mathcal{E} ::= x \mid n \mid \texttt{obj}\ \tau \mid \tau.f \mid e\ e$
Values	$v \in \mathcal{V} ::= n \mid \texttt{obj}\ \tau \mid \tau.f$
Member functions	$\mathcal{F} ::= t\langle\pi_1..\pi_n\rangle\ \texttt{has}\ f(x\texttt{:}\tau) \to \tau\{e\}$

The definition of a static member function has of the form $\tau\ \texttt{has}\ f(x : \tau_1) \to \tau_2\{e\}$. The type τ is the owner of the function and f is the name of the function. The function has a parameter x of type τ_1 and return type τ_2. The expression e is the body of the function.

During the instantiation process, all the types occurring in an expression are evaluated.

Definition 9. *(Type evaluation inside an expression)* $T \mid P \vdash e \Downarrow e'$ *iff every type* τ *occurring in expression* e *is replaced with* τ' *where* $T \mid P \vdash \tau \Downarrow \lfloor\tau'\rfloor$ *to produce expression* e'*.*

Substitution of expressions for expression variables is defined below. There are no variable binders inside expressions, so substitution is straightforward. We also extend type-substitution to expressions.

Substitution on expressions $e[y := e] \in \mathcal{E}$ $S(e) \in \mathcal{E}$

$$x[y := e] = \begin{cases} e & y = x \\ x & \text{otherwise} \end{cases}$$
$$(e_1\ e_2)[y := e] = e_1[y := e]\ e_2[y := e]$$
$$\tau.f[y := e] = \tau.f$$
$$\texttt{obj}\ \tau[y := e] = \texttt{obj}\ \tau$$
$$n[y := e] = n$$

$$S(x) = x$$
$$S(e_1\ e_2) = S(e_1)\ S(e_2)$$
$$S(\tau.f) = S(\tau).f$$
$$S(\texttt{obj}\ \tau) = \texttt{obj}\ S(\tau)$$
$$S(n) = n$$

A big-step operational semantics for the run-time evaluation of expressions is defined by a judgment of the form $F \mid T \vdash e \Rightarrow v_\perp$. The main computational rule is (R-App), which evaluates a function application expression. The expression e_1 evaluates to a member function expression $\tau.f$ and the operand e_2 evaluates to e_2'. The body of the member function $\tau.f$ is found in F. The argument e_2' is substituted for parameter x in the body e, which is then evaluated. The parameter and return types are required to be complete types because C++ has

pass-by value semantics: we need to know the layout of the types to perform the copy.

Similarly, in the (R-Obj) rule, the type of the object must be complete so that we know how to construct the object. The semantics includes error propagation rules so that we can distinguish between non-termination and errors. The (R-AppE1) rule states that a function application errors if either e_1 or e_2 evaluates to an error. Strictly speaking, this would force an implementation of C++.T to interleave the evaluation of e_1 and e_2 so that non-termination of either would not prevent encountering the error. Preferably we would allow for a sequential implementation but that is difficult to express with a big step semantics.

Run-time evaluation. $F \mid T \vdash e \Rightarrow v_\bot$

(R-Int)
$$\frac{}{F \mid T \vdash n \Rightarrow \lfloor n \rfloor}$$

(R-Obj)
$$\frac{\tau \text{ is complete in } T}{F \mid T \vdash \texttt{obj}\ \tau \Rightarrow \lfloor \texttt{obj}\ \tau \rfloor}$$

(R-Mem)
$$\frac{}{F \mid T \vdash \tau.f \Rightarrow \lfloor \tau.f \rfloor}$$

(R-App)
$$\frac{\begin{array}{c} F \mid T \vdash e_1 \Rightarrow \lfloor \tau.f \rfloor \quad F \mid T \vdash e_2 \Rightarrow \lfloor e_2' \rfloor \quad \tau\ \texttt{has}\ f(x{:}\tau_1) \to \tau_2\{e\} \in F \\ \tau_1 \text{ and } \tau_2 \text{ are complete in } T \quad F \mid T \vdash e[x := e_2'] \Rightarrow \lfloor e' \rfloor \end{array}}{F \mid T \vdash e_1\ e_2 \Rightarrow \lfloor e' \rfloor}$$

Error propagation rules

(R-AppE1)
$$\frac{\exists i \in 1,2.\ F \mid T \vdash e_i \Rightarrow \bot}{F \mid T \vdash e_1\ e_2 \Rightarrow \bot} \qquad \frac{\begin{array}{c} F \mid T \vdash e_1 \Rightarrow \lfloor \tau.f \rfloor \\ \neg \exists x, \tau_1, \tau_2, e.\ \tau\ \texttt{has}\ f(x{:}\tau_1) \to \tau_2\{e\} \in F \end{array}}{F \mid T \vdash e_1\ e_2 \Rightarrow \bot}$$

$$\frac{\begin{array}{c} F \mid T \vdash e_1 \Rightarrow \lfloor \tau.f \rfloor \quad F \mid T \vdash e_2 \Rightarrow \lfloor e_2' \rfloor \\ \tau\ \texttt{has}\ f(x{:}\tau_1) \to \tau_2\{e\} \in F \\ F \mid T \vdash e_1[x := e_2'] \Rightarrow \bot \end{array}}{F \mid T \vdash e_1\ e_2 \Rightarrow \bot} \qquad \frac{\begin{array}{c} F \mid T \vdash e_1 \Rightarrow \lfloor \tau.f \rfloor \\ \tau\ \texttt{has}\ f(x{:}\tau_1) \to \tau_2\{e\} \in F \\ \exists i \in 1,2.\ \tau_i \text{ is not complete in } T \end{array}}{F \mid T \vdash e_1\ e_2 \Rightarrow \bot}$$

$$\frac{}{F \mid T \vdash x \Rightarrow \bot} \qquad \frac{\tau \text{ is not complete in } T}{F \mid T \vdash \texttt{obj}\ \tau \Rightarrow \bot}$$

Proposition 5. *If* $F \mid T \vdash e \Rightarrow \lfloor e' \rfloor$ *then* $e' \in \mathcal{V}$.

Proof. By induction on evaluation.

The definition of well-typed expressions is defined by a judgment of the form $T \mid P \mid \Gamma \vdash e : \tau_\perp$. This typing judgment is used to type check expressions in the body of member functions of templates and specializations. If an expression contains a type that contains type variables, the type of the expression cannot be determined and is assigned the type $\perp$. *This does not indicate a type error.* When the member function is instantiated the type variables are replaced by closed types and the body is type checked again (See Section 5).

Well-typed expressions. $T \mid P \mid \Gamma \vdash e : \tau_\perp$

(T-Var1)
$$\frac{x : \tau \in \Gamma \quad \mathrm{FTV}(\tau) = \emptyset}{T \mid P \mid \Gamma \vdash x : \lfloor \tau' \rfloor}$$

(T-Var2)
$$\frac{x : \tau \in \Gamma \quad \mathrm{FTV}(\tau) \neq \emptyset}{T \mid P \mid \Gamma \vdash x : \perp}$$

(T-Int)
$$\frac{}{T \mid P \mid \Gamma \vdash n : \lfloor \texttt{int} \rfloor}$$

(T-Obj1)
$$\frac{t\langle \tau_1..\tau_n \rangle \text{ is complete in } T}{T \mid P \mid \Gamma \vdash \texttt{obj}\ t\langle \tau_1..\tau_n \rangle : \lfloor t\langle \tau_1..\tau_n \rangle \rfloor}$$

(T-Obj2)
$$\frac{\mathrm{FTV}(\tau) \neq \emptyset}{T \mid P \mid \Gamma \vdash \texttt{obj}\ \tau : \perp}$$

(T-Mem1)
$$\frac{T \mid P \vdash \tau \text{ wf} \quad \mathrm{FTV}(\tau) = \emptyset \quad \tau\{f : \texttt{fun}\ \tau'\} \in T}{T \mid P \mid \Gamma \vdash \tau.f : \lfloor \tau' \rfloor}$$

(T-Mem2)
$$\frac{\mathrm{FTV}(\tau) \neq \emptyset}{T \mid P \mid \Gamma \vdash \tau.f : \perp}$$

(T-App1)
$$\frac{T \mid P \mid \Gamma \vdash e_1 : \tau \to \tau' \quad T \mid P \mid \Gamma \vdash e_2 : \tau}{T \mid P \mid \Gamma \vdash e_1\ e_2 : \lfloor \tau' \rfloor}$$

(T-App2)
$$\frac{T \mid P \mid \Gamma \vdash e_1 : a_1 \quad T \mid P \mid \Gamma \vdash e_2 : a_2 \quad a_1 = \perp \vee a_2 = \perp}{T \mid P \mid \Gamma \vdash e_1\ e_2 \Downarrow e_1'\ e_2' : \perp}$$

Lemma 2. *(Substitution preserves well-typed expressions) If* $T \mid \emptyset \mid x : \tau_1 \vdash e : \tau_2$ *and* $T \mid \emptyset \mid \emptyset \vdash e' : \tau_1$ *then* $T \mid \emptyset \mid \emptyset \vdash e[x := e'] : \tau_2$.

Proof. By induction on the typing judgment.

Lemma 3. *(Environment weakening for well-typed expressions) If* $T \mid P \mid \Gamma \vdash e : \tau_2$ *and* $T \subseteq T'$ *then* $T' \mid P \mid \Gamma \vdash e : \tau_2$.

Proof. By induction on the typing judgment. The cases for (T-Obj1) and (T-Mem1) use Proposition 3.

The following defines when a member function is used in an expression and when a member function is used in a set of function definitions and when a member function is defined.

Definition 10.

- $\tau.f \in e \triangleq \tau.f$ *is a subexpression of* e *and* $\tau \in \mathbb{R}$.
- $\mathit{funused}(e) \triangleq \{\tau.f \mid \tau.f \in e\}$
- $\tau.f \in F \triangleq$ *there is a member function* π **has** $f'(x : \tau_1) \to \tau_2\{e\} \in F$ *such that* $\tau.f \in e$.
- $\mathit{funused}(F) \triangleq \{\tau.f \mid \tau.f \in F\}$
- *A function* $\tau.f$ ***is defined in*** F *iff* $\exists \tau', x, \tau_1, \tau_2, e. \tau' \doteq \tau \wedge \tau'$ **has** $f(x : \tau_1) \to \tau_2\{e\} \in F$.
- $\mathit{fundef}(F) \triangleq \{\tau.f \mid \tau.f$ *is defined in* $F\}$

Lemma 4. *If* $F \mid T \vdash e \Rightarrow \lfloor e' \rfloor$ *then* $\mathit{funused}(e')$ *is a subset of* $\mathit{funused}(e) \cup \mathit{funused}(F)$.

Proof. By induction on the evaluation judgment. The case for application relies on the fact that the functions used in $e_1[x := e_2]$ are a subset of the functions used in e_1 and e_2.

Definition 11. *(Well typed function environment) We write* $T \vdash F$ *if, for all full member specializations* r **has** $f(x : \tau_1) \to \tau_2\{e\} \in F$ *(note that* $r \in \mathbb{R}$*) we have*

1. $r\{f : \mathbf{fun}\ \tau_1 \to \tau_2\} \in T$
2. $T \mid \emptyset \mid x : \tau_1 \vdash e : \tau_2$
3. τ_1 *and* τ_2 *are complete in* T

Lemma 5. *(Type safety of expression evaluation) If*

1. $T \mid \emptyset \mid \emptyset \vdash e : \lfloor \tau \rfloor$ *and*
2. $F \mid T \vdash e \Rightarrow ans$ *and*
3. *every function used in* e *and* F *is defined in* F *and*
4. $T \vdash F$ *and*
5. *there are no duplicates in* T

then there exists v *such that* $ans = \lfloor v \rfloor$ *and* $T \mid \emptyset \mid \emptyset \vdash v : \lfloor \tau \rfloor$.

Proof. By induction on the evaluation judgment. The cases for application (including the cases for error propagation) rely on the assumptions that $T \vdash F$, every function used in e and F is defined in F, and that there are no duplicates in T. Also, the application cases use Proposition **??** and Lemmas 2, and 4. The two cases for object construction rely on the requirement for a complete type in the typing rule (T-Obj1). The other cases are straightforward.

4.1 Member function processing

During program instantiation there are two places where member function definitions are processed, when a user-defined function definition is encountered and when a member function is instantiated. We abstract the member function processing into a judgment of the form $T; F \vdash \pi \texttt{ has } f(x:\tau) \to \tau\{e\} \Downarrow T'; F'$. The definition is shown below.

Process member function $T; F \vdash \pi \texttt{ has } f(x:\tau) \to \tau\{e\} \Downarrow T; F$

(MemFun)

$$\frac{\begin{array}{c} T_1 \mid \mathrm{FTV}(\pi) \vdash \tau_1 \Downarrow \lfloor \tau_1' \rfloor \quad T_1 \mid \mathrm{FTV}(\pi) \vdash \tau_2 \Downarrow \lfloor \tau_2' \rfloor \quad T_1 \mid \mathrm{FTV}(\pi) \vdash e \Downarrow \lfloor e' \rfloor \\ N = \{\tau \mid \tau \in e' \vee \tau \in \{\pi, \tau_1', \tau_2'\}\} \\ N' = \{\tau \in N \mid \tau \text{ is not defined in } T_1\} \\ N' \text{ instantiates to } T_2 \text{ in } T_1 \\ T_1 \cup T_2 \mid \mathrm{FTV}(\pi) \mid x : \tau_1' \vdash e' : a \qquad (\mathrm{FTV}(\pi) \neq \emptyset \wedge a = \bot) \vee a = \lfloor \tau_2' \rfloor \\ F_2 = \{\pi \texttt{ has } f(x:\tau_1') \to \tau_2'\{e'\}\} \end{array}}{T_1 \mid F_1 \vdash \pi \texttt{ has } f(x:\tau_1) \to \tau_2\{e\} \Downarrow T_1 \cup T_2; F_1 \cup F_2}$$

The type parameters and types in the body of the function are evaluated with $\mathrm{FTV}(\pi)$ for the in-scope type parameters. We record all of the types that need to be instantiated in the set N' and put all of the instantiations in set T_2. We type check the body of the function in an environment extended with T_2. If there are free type variables in the template pattern π, then type checking may result in $\bot$. Otherwise the type of the body must equal the return type.

5 Programs and the instantiation process

A program is a sequence of template and function definitions:

Abstract syntax of programs $p \in \mathcal{P}$

$$\begin{array}{l} \text{Definitions } d \in \mathcal{D} ::= \mathcal{T} \mid \mathcal{F} \\ \text{Programs } \quad p \in \mathcal{P} ::= d^* \end{array}$$

The program instantiation judgment, defined below, performs type evaluation, template instantiation, and type checking on each definition. The following auxiliary definitions are used in the definition of program instantiation.

Definition 12.

- $\tau.f$ ***is defined in*** F *iff* $\exists \tau' x \tau_1 \tau_2 e. \tau \doteq \tau' \wedge \tau' \texttt{ has } f(x:\tau_1) \to \tau_2\{e\} \in F$.
- *We write* $\tau \in e$ *iff* $\texttt{obj}\ \tau$ *is a subexpression of* e *and* $\tau \in \mathbb{R}$.
- *The notation* X, z *stands for* $\{z\} \cup X$ *where* $z \notin X$.

There are four rules for program instantiation:

(C-Nil): Program instantiation is finished when there are definitions for all of the functions used in the program.

(C-InstFun): Once the entire program has been processed we instantiate member functions that are used but not yet defined. and instantiates the function. We find the best matching template and the corresponding member function definition. The matching substitution S is applied to the type parameters and the body of the function. We then process the instantiated member function with rule (MemFun).

(C-Tm): For template definition, we check that the template is not already defined and then evaluate the template's member. We then insert the evaluated template into T and process the rest of the program.

(C-Fun): For member function definitions, we check that there is a template defined with a member declaration for this function. Then we check that there is not already a definition for this function. We then apply the (MemFun) rule to the member function and then process the rest of the program.

Program instantiation $T \mid F \vdash p \Downarrow T; F$

(C-Nil)
$$\frac{\mathit{funused}(F) \subseteq F}{T \mid F \vdash \epsilon \Downarrow T; F}$$

(C-InstFun)
$$\frac{\begin{array}{c}\tau.f \in \mathit{funused}(F_1) - F_1 \qquad \mathit{lookup}(T_1, \tau) = \pi\{f : \texttt{fun}\ \tau_1 \to \tau_2\} \\ \pi\ \texttt{has}\ f(x{:}\tau_1) \to \tau_2\{e\} \in F_1 \qquad S(\pi) = \tau \\ T_1; F_1 \vdash \tau\ \texttt{has}\ f(x{:}S(\tau_1)) \to S(\tau_2)\{S(e)\} \Downarrow T_2; F_2 \qquad T_2 \mid F_2 \vdash \epsilon \Downarrow T'; F'\end{array}}{T_1 \mid F_1 \vdash \epsilon \Downarrow T'; F'}$$

(C-Tm)
$$\frac{\begin{array}{c}\pi \text{ is not defined in } T \qquad T \mid \mathrm{FTV}(\pi) \vdash \tau \Downarrow \lfloor \tau' \rfloor \\ \{\pi\{m : \kappa\ \tau'\}\} \cup T \mid F \vdash p_1 \Downarrow T'; F'\end{array}}{T \mid F \vdash \pi\{m : \kappa\ \tau\} :: p_1 \Downarrow T'; F'}$$

(C-Fun)
$$\frac{\begin{array}{c}\pi.f \text{ is not defined in } F_1 \qquad \mathit{lookupmem}(T_1, \pi, f) = \lfloor (\texttt{fun}, \tau_1 \to \tau_2) \rfloor \\ T_1; F_1 \vdash \pi\ \texttt{has}\ f(x{:}\tau_1) \to \tau_2\{e\} \Downarrow T_2; F_2 \qquad T_2 \mid F_2 \vdash p_1 \Downarrow T'; F'\end{array}}{T_1 \mid F_1 \vdash \pi\ \texttt{has}\ f(x{:}\tau_1) \to \tau_2\{e\} :: p_1 \Downarrow T'; F'}$$

5.1 Type Safety

For the purposes of proving type safety, we need to show that the semantics of program instantiation establish the appropriate properties needed by Lemma 5 (Type soundness for evaluation). The following lemma captures the invariants that are maintained during program instantiation to achieve this goal.

Lemma 6. *(Type preservation for program instantiation) If* $T \mid F \vdash p \Downarrow T'; F'$, *and*

1. $T \vdash F$, *and*
2. *there are no duplicates in* T, *and*
3. $FTV(T) = \emptyset$

then

1. $funused(F') \subseteq F'$, *and*
2. $T' \vdash F'$, *and*
3. *there are no duplicates in* T', *and*
4. $FTV(T') = \emptyset$

Proof. By induction on the instantiation of p. The case for template definitions uses Proposition 2 (Properties of type evaluation) and and Lemma 3 (Environment weakening for well-typed expressions). The cases for member function definitions and member function instantiation use Proposition 2 and Lemma 3. In addition they use Lemma 1 (Member lookup produces closed types).

The proof of the type-safety theorem is a straightforward use of Lemma 6 (Type preservation for program instantiation) and Lemma 5 (Type safety of expression evaluation).

Theorem 1.(Type Safety) *If*

1. $\emptyset \mid \emptyset \mid \emptyset \vdash p \Downarrow T; F$, *and*
2. $\texttt{Main}\langle\rangle\{\texttt{main} : \texttt{int} \rightarrow \texttt{int}\} \in T$, *and*
3. $\texttt{Main}\langle\rangle\ \texttt{has main(x:int)} \rightarrow \texttt{int}\{e\} \in F$, *and*
4. $F \mid T \vdash \texttt{Main}\langle\rangle.\texttt{main(n)} \Rightarrow ans$

then there exists v *such that* $ans = \lfloor v \rfloor$ *and* $T \mid \emptyset \mid \emptyset \vdash v : \texttt{int}$.

Proof. From $\texttt{Main}\langle\rangle\{\texttt{main} : \texttt{int} \rightarrow \texttt{int}\} \in T$ we have $T \mid \emptyset \mid \emptyset \vdash \texttt{Main}\langle\rangle.\texttt{main(n)} : \texttt{int}$. By Lemma 6 we know that all the functions used in F are defined, $T \vdash F$, there are no duplicates in T, and there are no free type variables in T. From $\texttt{Main}\langle\rangle\ \texttt{has main(x:int)} \rightarrow \texttt{int}\{e\} \in F$ we know that the function used in $\texttt{Main}\langle\rangle.\texttt{main(n)}$ is defined, so we apply Lemma 5 to obtain v such that $ans = \lfloor v \rfloor$ and $T \mid \emptyset \mid \emptyset \vdash v : \texttt{int}$.

6 Discussion

The semantics defined in this paper instantiates fewer templates than what is mandated by the C++ standard. In particular, the C++ standard says that member access, such as `A<int>::u`, causes the instantiation of `A<int>`. In our semantics, the member access will obtain the definition of member `u` but it will not generate a template specialization for `A<int>`. We only generate template specializations for types that appear in residual program contexts that require

complete types: object construction and function parameters and return types. Our type-safety result shows that even though we produce fewer template specializations, we produce enough to ensure the proper run-time execution of the program. The benefit of this semantics is that the compiler is allowed to be more space efficient.

The semantics of member function instantiation is a point of some controversy. Section [14.6.4.1 p1] of the Standard says that the point of instantiation for a member function immediately follows the enclosing declaration that triggered the instantiation (with a caveat for dependent uses within templates). The problem with this rule is that uses of a member function may legally precede its definition and the definition is needed to generate the instantiation. (A use of a member function must only come after the *declaration* of the member function, which is in the template specialization.) In general, the C++ Standard is formulated to allow for compilation in a single pass, whereas the current rules for member instantiation would require two passes. Also, there is a disconnect between the Standard and the current implementations. The Edison Design Group and GNU compilers both delay the instantiation of member functions to the end of the program (or translation unit). We discussed this issue on C++ committee[2] and the opinion was that this is a defect in the C++ Standard and that instantiation of member functions should be allowed to occur at the end of the program. Therefore, C++.T places instantiations for member functions at the end of the program.

7 Related work

Recently, Stroustrup and Dos Reis proposed a formal account of the type system for C++ [11, 12]. However, they do not define the semantics of evaluation and they do not study template specialization. The focus of the work by Stroustrup and Dos Reis is to enable the type checking of template definitions separately from their uses. Siek et al. [14] also describe an extension to improve the type checking of template definitions and uses.

Wallace studied the dynamic evaluation of C++, but not the static aspects such as template instantiation [18].

C++ templates are widely used for program generation. There has been considerable research on language support for program generation, resulting in languages such as MetaOCaml [2] and Template Haskell [13]. These languages provide first-class support for program generation by including a bracket construct to delay computation, creating a piece of code, and an escape construct, that forces a computation and splices the result into the generated code. The advanced type system used in MetaOCaml guarantees that the generated code is type safe. There are no such guarantees in C++. The formal semantics defined in this paper will facilitate comparing C++ with languages such as MetaOCaml

[2] A post to the C++ committee email reflector on September 19, 2005, with a response from John Spicer.

and will help in finding ways to improve C++. The C++ Standards Committee has begun to investigate improved support for metaprogramming [16].

8 Conclusion

This paper presents a formal account of C++ templates. We identify a small subset, named C++.T that includes templates, specialization, and member functions. We define the compile-time and run-time semantics of C++.T, including type evaluation, template instantiation, and a type system. The main technical result is the proof of type safety, which states that if a program is valid (template instantiation succeeds), then run-time execution of the program will not encounter type errors. In the process of formalizing C++.T, we found two interesting issues: the C++ Standard instantiates member functions too soon and generates unnecessary template specializations.

From the point of view of language semantics and program generation research, it was interesting to see that C++.T involves a form of evaluation under variable binders at the level of types but not at the level of terms. It will be interesting to investigate how this affects the expressivity of C++ templates as a mechanism for writing program generators.

Bibliography

[1] *Haskell 98 Language and Libraries: The Revised Report*, December 2002. `http://www.haskell.org/onlinereport/index.html`.

[2] MetaOCaml: A compiled, type-safe multi-stage programming language. Available online from `http://www.metaocaml.org/`, 2004.

[3] David Abrahams and Aleksey Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2004.

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

[5] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative programming: methods, tools, and applications*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

[6] Steven E. Ganz, Amr Sabry, and Walid Taha. Macros as multi-stage computations: type-safe, generative, binding macros in MacroML. In *ICFP '01: Proceedings of the sixth ACM SIGPLAN international conference on Functional programming*, pages 74–85, New York, NY, USA, 2001. ACM Press.

[7] International Organization for Standardization. *ISO/IEC 14882:2003: Programming languages — C++*. Geneva, Switzerland, October 2003.

[8] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.

[9] Tobias Nipkow. Structured proofs in Isar/HOL. In *TYPES*, number 2646 in LNCS, 2002.

[10] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[11] Gabriel Dos Reis and Bjarne Stroustrup. A formalism for C++. Technical Report N1885=05-0145, ISO/IEC JTC1/SC22/WG21, 2005.

[12] Gabriel Dos Reis and Bjarne Stroustrup. Specifying C++ concepts. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, January 2006.

[13] Tim Sheard and Simon Peyton Jones. Template meta-programming for haskell. In *Haskell '02: Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, pages 1–16, New York, NY, USA, 2002. ACM Press.

[14] Jeremy Siek, Douglas Gregor, Ronald Garcia, Jeremiah Willcock, Jaakko Järvi, and Andrew Lumsdaine. Concepts for C++0x. Technical Report N1758=05-0018, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, January 2005.

[15] Jeremy Siek and Walid Taha. C++.T formalization in Isar. Technical report, Rice University, Houston, TX, December 2005. Available online from `http://www.cs.rice.edu/~jgs3847/publications.html`.

[16] Daveed Vandevoorde. Reflective metaprogramming in C++. Technical Report N1471/03-0054, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, April 2003.

[17] Todd Veldhuizen. Using C++ template metaprograms. *C++ Report*, 7(4):36–43, May 1995. Reprinted in C++ Gems, ed. Stanley Lippman.

[18] Charles Wallace. The semantics of the C++ programming language. pages 131–164, 1995.

[19] Markus Wenzel. *The Isabelle/Isar Reference Manual*. TU München, April 2004.