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. We give an overview of the technical contributions of this paper at the end of this section.

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]; }
};
```

* This work was supported by NSF ITR-0113569 Putting Multi-Stage Annotations to Work, Texas ATP 003604-0032-2003 Advanced Languages Techniques for Device Drivers, and NSF SOD-0439017 Synthesizing Device Drivers.

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 a call to the member appears in the program. 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.

Going back to the second line of the main function, consider the use of type buffer<bool,3>. This matches the partial specialization buffer<bool,n> but not the full specialization buffer<bool,0>, so C++ will generate a specialization from the partial specialization buffer<bool,n>.

In the third line of main, the type buffer<bool,0> type matches the full specialization of buffer, so C++ does not need to generate a specialization.

1.1 Compile-time programming with templates

The buffer<bool,n> specialization contains a small compile-time computation for 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, and types can 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 mimic algebraic datatypes [5], such as those found in SML [10] and Haskell [1]. Furthermore, templates are "open ended" so they can mimic the extensible variants of Objective Caml [9].

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;
};
```

1.2 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, 19]. 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
}
```

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

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 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:

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

Such optimization is not required by the C++ Standard. However, if the compiler performs inlining, it must preserve the call-by-value semantics. We might label function f with the **inline** keyword, but this 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 [7] 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.

1.3 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 2.2)
2. type evaluation (Section 2.3),
3. expression evaluation and well-typed expressions (Section 3), and
4. template instantiation (Section 4).

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 terminates without error. This definition is unusual because some potentially non-terminating evaluation (type evaluation) is performed as part of determining whether a program is "valid". In particular, this means that determining validity is undecidable. 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 4.3). We wrote the proof in the Isar proof language [11, 21] and mechanically verified the proof using the Isabelle proof assistant [12]. Due to space considerations, we do not present the Isar proof in this paper but refer the reader to the accompanying technical report [17].

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 template instantiation process converts a program into an expanded program with more template specializations. The Standard requires the 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 omit the specialization from the resulting program, analogously to the way procedure activation frames are discarded when a function returns, thereby improving the space-complexity for template programs.

1.4 Overview of the formalization

C++.T contains syntactic categories for types, expressions (or terms), and definitions. A program is a sequence of definitions. The semantics of C++.T includes compile-time and run-time components. The main semantic definitions for the compile-time aspects of C++.T are:

Type evaluation We define a big step evaluation function that reduces type expressions to simpler type expressions in §2.3.
Well-formed types We define when a type expression is valid in §2.4.
Type evaluation in expressions Types may occur within expressions, so we define how type evaluation is performed within expressions in §3.
Well-typed expressions We define type checking for C++.T expressions in §3.
Program instantiation The instantiation process expands a program by generating the necessary template specializations and member function specializations. Program instantiation is defined in §4. A **valid** C++.T program is one in which program instantiation completes successfully.

The run-time aspect of C++.T consists of one definition: expression evaluation, which we cover in §3. Fig. 1 shows the dependencies between the semantic definitions for C++.T.

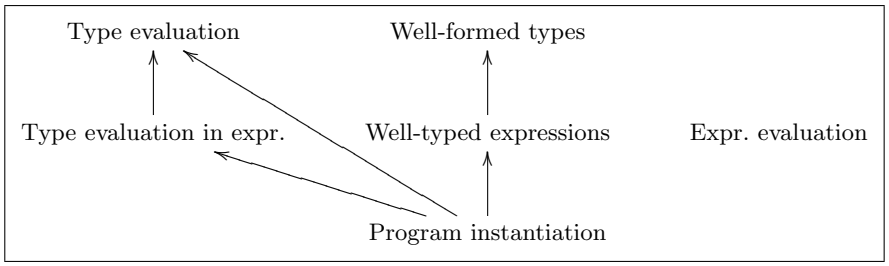

Fig. 1. Semantic definitions and their dependencies.

The main technical result is that valid programs are type safe. That is, if a program successfully instantiates, then the run-time evaluation will result in a value of the appropriate type, provided that evaluation terminates. The type safety theorem is proved in Section 4. The lemmas and definitions in the following sections lead up to this result.

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

A type in C++.T can be a type variable α, member access $\tau.a$, template identifier $t\langle\tau_1..\tau_2\rangle$, function type $\tau_1 \to \tau_2$, or `int`. There are no variable binders in $\mathbb{T}$ so the set of free type variables (FTV) of a type is simply the type variables occurring in the type. A type is **closed** if it contains no type variables. We pick out two subsets of $\mathbb{T}$: type patterns Π and residual types $\mathbb{R}$. Type patterns do not include member access types. Residual types are restricted to template identifiers, functions, and `int`. When applied to closed types, type evaluation produces residual types.

Member access $\tau.a$ refers to a type definition for a within the template specialization identified by τ. In the concrete syntax of C++, type member access is written $\tau\texttt{::}a$. A template identifier $t\langle\tau_1..\tau_n\rangle$ refers to the specialization of template t for the type arguments $\tau_1..\tau_n$. A function type $\tau_1 \to \tau_2$ corresponds to the C++ syntax $\tau_2\texttt{(*)(}\tau_1\texttt{)}$.

The abstract syntax $t\langle\pi_1..\pi_n\rangle\{m : \kappa\ \tau\}$ is used for both class templates and class template specializations, where t is the name of the template and $\langle\pi_1..\pi_n\rangle$ is the specialization pattern. The body contains the declaration of a single member $m : \kappa\ \tau$, where m is the name, κ is the kind, and τ is its type. When all of the patterns $\pi_1..\pi_n$ are variables, the declaration is a class template, such as $t\langle\alpha_1..\alpha_n\rangle\{m : \kappa\ \tau\}$. When the patterns contain no type variables, then the

declaration is a full specialization of a class template, such as $t\langle\tau_1..\tau_n\rangle\{m\!:\!\kappa\ \tau\}$ 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 A2 and change the use of x 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 implementation of a member function is written separately and the syntax for that is introduced in Section 3.

2.1 Type substitution

A *substitution* S 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. A *renaming* is an injective substitution whose codomain is restricted to type variables. We use renaming to define α-equivalence for types and the injective property is needed to disallow mapping distinct variables to the same variable. We extend substitution to types with the following definition.

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

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

The case $S(\alpha) = S\ \alpha$ in the definition of substitution on types is not a circular definition. We are given a function S on type variables and we are building a function, also called S, on type expressions.

Proposition 1. $FTV(S(\tau)) = \bigcup_{\alpha \in FTV(\tau)} FTV(S(\alpha))$

Proof. By induction on the structure of τ.

2.2 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 Standard). The following defines matching.

Definition 1. *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 2. *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 2. *If $\tau_1 \leq \tau_2$ and $\tau_2 \leq \tau_1$ then there exists a renaming S such that $S(\tau_2) = \tau_1$.*

We use the notation $\doteq$ for type pattern equivalence and use it to define a notion of duplicate template definitions. The symbol $\triangleq$ means "is defined to be".

Definition 3.

- $\pi_1 \doteq \pi_2 \triangleq \pi_1 \leq \pi_2$ *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 relation $\leq$ to templates as follows:

Definition 4. $\pi_1\{m_1 : \kappa_1\ \tau_1\} \leq \pi_2\{m_2 : \kappa_2\ \tau_2\} \triangleq \pi_1 \leq \pi_2$

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

Definition 5. *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.

$$\mathit{lookup} : \mathsf{Set}\ \mathcal{T} \times \mathbb{T} \to \mathcal{T}_\perp$$

$$\mathit{lookup}(T, \tau) \triangleq \begin{cases} \lfloor \max\{\pi\{m{:}\kappa\ \tau\} \in T \mid \pi \leq \tau\} \rfloor & \text{if the max exists} \\ \perp & \text{otherwise} \end{cases}$$

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. The condition that $S(\pi) = \tau$ fully determines the action of S on free variables in π, and because the free variables in τ' are required to be a subset of those in π, the type $S(\tau')$ is unique.

$$\mathit{inst} : \mathsf{Set}\ \mathcal{T} \times \mathbb{T} \to \mathcal{T}_\perp$$

$$\mathit{inst}(T, \tau) \triangleq \begin{cases} \lfloor \tau\{m{:}\kappa\ S(\tau')\} \rfloor & \text{where } \mathit{lookup}(T, \tau) = \lfloor \pi\{m{:}\kappa\ \tau'\} \rfloor \\ & \text{and } S(\pi) = \tau \\ \perp & \text{otherwise} \end{cases}$$

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

$$\mathit{lookupmem} : \mathsf{Set}\ \mathcal{T} \times \mathbb{T} \times \mathsf{MemName} \to (\mathcal{K} \times \mathbb{T})_\perp$$

$$\mathit{lookupmem}(T, \tau, m) \triangleq \begin{cases} \lfloor (\kappa, \tau') \rfloor & \mathit{inst}(T, \tau) = \lfloor \tau\{m{:}\kappa\ \tau'\} \rfloor \\ \perp & \text{otherwise} \end{cases}$$

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

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

Proof. A straightforward use of Proposition 1.

2.3 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 float 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 [8] says that the point of instantiation for a specialization precedes the first declaration that contains a use of the specialization in an instantiation context, unless the enclosing declaration is a template and the use is dependent on the template parameters. Otherwise 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 we must first reduce it to `A<float>` to check whether the type was already instantiated.

The type evaluation judgment has the form $T; P \vdash \tau \Downarrow \tau'$ where T is the set of in-scope template definitions and P is the set of in-scope type parameters. Type evaluation reduces type τ to τ'. The type evaluation rules do not include error handling cases because it is not important for us to distinguish between errors and non-termination of type evaluation. A C++.T program is invalid if the instantiation process does not terminate.

The evaluation rules for type expressions are defined in Fig. 2. The rule (C-VarT) says a type variable α evaluates to itself provided α is in scope. The rule (C-MemT1) that defines type member access $\tau.a$ is reminicent of 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 lookup function takes care of substituting the type arguments $\tau_1..\tau_n$ for template parameters. The member type is evaluated to τ'' and that is the result. An alternative design of the rules would be to 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 $\tau'.a$. 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.

The *names* function returns the set of template names from a set of templates.

$$names(T) \triangleq \{t \mid t\langle\pi_1..\pi_n\rangle\{m\!:\!\kappa\ \tau\} \in T\}$$

Fig. 2. Type evaluation. $T; P \vdash \tau \Downarrow \tau$

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

(C-MemT1)
$$\frac{\begin{array}{c} T; P \vdash \tau \Downarrow_n t\langle \tau_1..\tau_n \rangle \qquad \bigcup_i \mathrm{FTV}(\tau_i) = \emptyset \\ \mathit{lookupmem}(T, t\langle \tau_1..\tau_n \rangle, a) = \lfloor (\texttt{type}, \tau') \rfloor \\ T; P \vdash \tau' \Downarrow \tau'' \end{array}}{T; P \vdash \tau.a \Downarrow \tau''}$$

(C-MemT2)
$$\frac{T; P \vdash \tau \Downarrow \tau' \qquad \mathrm{FTV}(\tau') \neq \emptyset}{T; P \vdash \tau.a \Downarrow \tau'.a}$$

(C-TmT)
$$\frac{t \in \mathit{names}(T) \qquad \forall i \in 1..n.\ T; P \vdash \tau_i \Downarrow \tau_i'}{T; P \vdash t\langle \tau_1..\tau_n \rangle \Downarrow t\langle \tau_1'..\tau_n' \rangle}$$

(C-ArrowT)
$$\frac{T; P \vdash \tau_1 \Downarrow \tau_1' \qquad T; P \vdash \tau_2 \Downarrow \tau_2'}{T; P \vdash \tau_1 \to \tau_2 \Downarrow \tau_1' \to \tau_2'}$$

(C-IntT)
$$\frac{}{T; P \vdash \texttt{int} \Downarrow \texttt{int}}$$

Proposition 3. *(Properties of type evaluation)*

1. *If* $T; P \vdash \tau \Downarrow \tau'$ *then* $FTV(\tau') \subseteq P$.
2. *If* $T; \emptyset \vdash \tau \Downarrow \tau'$ *then* $\tau' \in \mathbb{R}$.

It is worth mentioning that type evaluation may diverge for some types and therefore it is impossible to build a derivation for those types (the derivation would need to be infinite). For example, let $T = \{A\langle\alpha\rangle\{x : \texttt{type}\ A\langle A\langle\alpha\rangle\rangle.x\}\}$ and $P = \emptyset$. Then we can not build an derivation for $\texttt{A}\langle\texttt{int}\rangle.x$.

2.4 Well-formed types

Well-formed types are types that do not contain out-of-scope type parameters or use undefined template names. The definition of well-formed types is given in Fig. 3. The well-formed type judgment has the form $T; P \vdash \tau$ wf where T is the set of in-scope templates and P is the set of in-scope type parameters. In the case for member access, we do not check that the member name is indeed a member of the given type τ, as that would require us to evaluate τ. The purpose

of the well-formed type judgment is not to ensure a safety property for type evaluation, it is merely to check for uses of undefined type variables or template names.

Fig. 3. Well-formed types. $T; P \vdash \tau \text{ wf}$

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

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

Proposition 4. *(Properties of well-formed types)*

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

Lemma 2. *(Well-formed types in an empty type variable environment are closed.) If* $T; \emptyset \vdash \tau$ wf *then* $FTV(\tau) = \emptyset$.

Proof. By induction on the well-formed type judgment.

3 Expressions and functions

The expressions of C++.T include variables x, integers n, object creation $\texttt{obj}\ \tau$, static member function access $\tau.f$, and function application $e_1\ e_2$. (In C++ the syntax for object creation is $\tau\texttt{()}$, static member access is $\tau\texttt{::}f$, and function application is $e_1\texttt{(}e_2\texttt{)}$.)

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

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	$\mathcal{F} ::= t\langle \pi..\pi \rangle\ \texttt{has}\ f(x\!:\!\tau) \to \tau\{e\}$
Errors	$\epsilon ::= \mathsf{Error} \mid \mathsf{Abort}$
Answers	$ans ::= v \mid \epsilon$

A static member function definition has 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.

When a member function is instantiated, all the types in the member function are evaluated, including those occurring in the body expression.

Definition 7. *(Type evaluation inside an expression)* $T; P \vdash e \Downarrow e'$ *iff every type* τ *occurring in expression* e *is replaced with* τ' *where* $T; P \vdash \tau \Downarrow \tau'$ *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 $\qquad e[y := e] \in \mathcal{E} \qquad S(e) \in \mathcal{E}$

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

The following defines when a type is defined and when a type is complete, two notions that will be used in the evaluation semantics.

Definition 8.

- *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* $\tau \in \mathbb{R}$*.*

We present a big-step semantics for the run-time evaluation of expressions with a judgment of the form $F; T \vdash e \hookrightarrow_n ans$, defined in Fig. 4. The F is a set of member function definitions and T is a set of template definitions. Evaluation produces an answer that is either a value, Error, or Abort. The n is used to limit the derivation depth by aborting when n reaches 0. This lets us distinguish between erroneous programs versus nonterminating programs and thereby strengthen the statement of type safety and ensure that all cases are handled by the evaluation rules [6].

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 results in an error 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. This is not the intended semantics, but a precise treatment is rather verbose. Our type safety result still holds for any sequential implementation because the behavior only differs on programs that are rejected by the type system.

Fig. 4. Run-time evaluation. $F; T \vdash e \hookrightarrow_n ans$

(R-Int)
$$\frac{0 < k}{F; T \vdash n \hookrightarrow_k n}$$

(R-Obj)
$$\frac{\tau \text{ is complete in } T \quad 0 < n}{F; T \vdash \texttt{obj}\ \tau \hookrightarrow_n \texttt{obj}\ \tau}$$

(R-Mem)
$$\frac{0 < n}{F; T \vdash \tau.f \hookrightarrow_n \tau.f}$$

(R-App)
$$\frac{\begin{array}{c} F; T \vdash e_1 \hookrightarrow_n \tau.f \quad F; T \vdash e_2 \hookrightarrow_n v_2 \quad \tau \texttt{ has } f(x{:}\tau_1) \to \tau_2\{e\} \in F \\ \tau_1 \text{ and } \tau_2 \text{ are complete in } T \quad F; T \vdash e[x := v_2] \hookrightarrow_n v \end{array}}{F; T \vdash e_1\ e_2 \hookrightarrow_{n+1} v}$$

Error introduction rules

$$\frac{0 < n}{F; T \vdash x \hookrightarrow_n \mathsf{Error}} \qquad \frac{\tau \text{ is not complete in } T \quad 0 < n}{F; T \vdash \texttt{obj}\ \tau \hookrightarrow_n \mathsf{Error}} \qquad \frac{}{F; T \vdash e \hookrightarrow_0 \mathsf{Abort}}$$

$$\frac{\begin{array}{c} F; T \vdash e_1 \hookrightarrow_n \tau.f \\ \neg\exists x, \tau_1, \tau_2, e.\ \tau \texttt{ has } f(x{:}\tau_1) \to \tau_2\{e\} \in F \end{array}}{F; T \vdash e_1\ e_2 \hookrightarrow_{n+1} \mathsf{Error}} \qquad \frac{\begin{array}{c} F; T \vdash e_1 \hookrightarrow_n \tau.f \\ \tau \texttt{ has } f(x{:}\tau_1) \to \tau_2\{e\} \in F \\ \exists i \in 1, 2.\ \tau_i \text{ is not complete in } T \end{array}}{F; T \vdash e_1\ e_2 \hookrightarrow_{n+1} \mathsf{Error}}$$

Error propagation rules

(R-AppE1)
$$\frac{\exists i \in 1, 2.\ F; T \vdash e_i \hookrightarrow_n \epsilon}{F; T \vdash e_1\ e_2 \hookrightarrow_{n+1} \epsilon} \qquad \frac{\begin{array}{c} F; T \vdash e_1 \hookrightarrow_n \tau.f \quad F; T \vdash e_2 \hookrightarrow_n v_2 \\ \tau \texttt{ has } f(x{:}\tau_1) \to \tau_2\{e\} \in F \\ F; T \vdash e_1[x := v_2] \hookrightarrow_n \epsilon \end{array}}{F; T \vdash e_1\ e_2 \hookrightarrow_{n+1} \epsilon}$$

Well-typed expressions are defined by a judgment of the form $T; P; \Gamma \vdash e : \tau_?$, shown in Fig. 5. This judgment is used to type check expressions in the body of member functions of templates and specializations. If an expression contains a type with a type variable, the type of the expression cannot be determined and the typing judgment returns ? (for unknown). Thus, type checking within template definitions is incomplete in the sense that it does not guarantee that instantiating a member function will result in a well-typed body. When the member function is instantiated, type variables are replaced by closed types and the body is type checked again (See Section 4).

Lemma 3. *(Substitution preserves well-typed expressions) If* $T; \emptyset; x : \tau_1 \vdash e : \tau_2$ *and* $T; \emptyset; \emptyset \vdash e' : \tau_1$ *then* $T; \emptyset; \emptyset \vdash e[x := e'] : \tau_2$.

Proof. By induction on the typing judgment.

Lemma 4. *(Environment weakening for well-typed expressions) If* $T; P; \Gamma \vdash e : \tau_2$ *and* $T \subseteq T'$ *then* $T'; P; \Gamma \vdash e : \tau_2$.

Proof. By induction on the typing judgment. The cases for (T-Obj1) and (T-Mem1) use Proposition 4.

In what follows we give a formal definition of what it means for a member function to be used in an expression and to be used in a set of function definitions.

Definition 9.

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

Lemma 5. *If* $F; T \vdash e \hookrightarrow_n e'$ *then* $funused(e')$ *is a subset of* $funused(e) \cup funused(F)$.

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 10. *(Well typed function environment) We write* $T \vdash F$ *if, for all full member specializations* r **has** $f(x : \tau_1) \rightarrow \tau_2\{e\} \in F$ *we have*

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

Fig. 5. Well-typed expressions. $T; P; \Gamma \vdash e : \tau_?$

(T-Var1)
$$\frac{x : \tau \in \Gamma \quad \mathrm{FTV}(\tau) = \emptyset}{T; P; \Gamma \vdash x : \lfloor \tau \rfloor}$$

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

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

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

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

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

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

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

(T-App2)
$$\frac{\begin{array}{c} T; P; \Gamma \vdash e_1 : a_1 \quad T; P; \Gamma \vdash e_2 : a_2 \\ a_1 = ? \vee a_2 = ? \end{array}}{T; P; \Gamma \vdash e_1\ e_2 : ?}$$

The next lemma is type safety for expression evaluation. The appearance of $F; T \vdash e \hookrightarrow_n ans$ in the conclusion of this lemma, and not as a premise, would normally be a naive mistake because not all programs terminate. However, by using n-depth evaluation, we can construct a judgment regardless of whether the program is non-terminating. Further, by placing $F; T \vdash e \hookrightarrow_n ans$ in the conclusion, this lemma proves that our evaluation rules are complete, analogous to a progress lemma for small-step semantics. We learned of this technique from Ernst et al. [6].

Lemma 6. *(Type safety of expression evaluation) If*

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

then for all n, $F; T \vdash e \hookrightarrow_n ans$ *and either* $ans = \mathsf{Abort}$ *or there exists* v *such that* $ans = v$ *and* $T; \emptyset; \emptyset \vdash v : \lfloor \tau \rfloor$.

Proof. By strong induction on n, followed by case analysis on the final derivation step in the typing judgment for e. 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 2 (Well-formed types in an empty type variable environment are closed), Lemma 3 (Substitution preserves well-typed expressions), and Lemma 5. The cases for object construction rely on the requirement for a complete type in the typing rule (T-Obj1). The other cases are straightforward.

4 Programs and the instantiation process

A program is a sequence of template and function definitions. To review, templates $\mathcal{T}$ were defined in Section 2 and functions $\mathcal{F}$ were defined in Section 3.

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

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

The program instantiation judgment has the form $T; F \vdash p \Downarrow T'; F'$, where T is a set of templates, F a set of member function definitions, and p is the program. The result of instantiation is an updated set T' of templates and of member functions F'. The program instantiation judgment performs type evaluation, template instantiation, and type checking on each definition. The following auxiliary definitions are used in the definition of program instantiation.

Definition 11.

- *We write $\tau \in e$ iff* `obj` τ *or* $\tau.f$ *is a subexpression of* e *and* $\tau \in \mathbb{R}$.
- *The notation* X, z *stands for* $\{z\} \cup X$ *where* $z \notin X$.

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 $T; F \vdash \pi \texttt{ has } f(x\!:\!\tau) \to \tau'\{e\} \Downarrow T'; F'$ with a single rule (MemFun). This rule evaluates the type expressions that occur in the member function then records all of the types that need to be instantiated in the set N'. The results of template lookup for each type in N' is placed in set T_2, which is then instantiated to the set T_3 (The listof function converts a set into a list). We type check the body of the function in an environment extended with T_3. If there are free type variables in the template pattern π, then type checking may result in ?. Otherwise the type of the body must be identical to the return type.

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

$$
(\textsc{MemFun})\quad
\frac{
\begin{array}{c}
T_1; \mathrm{FTV}(\pi) \vdash \tau_1 \Downarrow \tau_1' \quad T_1; \mathrm{FTV}(\pi) \vdash \tau_2 \Downarrow \tau_2' \quad T_1; \mathrm{FTV}(\pi) \vdash e \Downarrow e' \\
N = \{\tau \mid \tau \in e'\} \cup \{\pi, \tau_1', \tau_2'\} \\
N' = \{\tau \in N \mid \tau \text{ is not defined in } T_1\} \quad \forall \tau \in N'.\ \mathit{inst}(T_1, \tau) \neq \bot \\
T_2 = \{\tau\{m\!:\!\kappa\ \tau'\} \mid \tau \in N' \wedge \mathit{inst}(T_1, \tau) = \lfloor \tau\{m\!:\!\kappa\ \tau'\} \rfloor\} \\
T_1; F_1 \vdash \mathsf{listof}(T_2) \Downarrow T_3; F_1 \\
T_3; \mathrm{FTV}(\pi); x : \tau_1' \vdash e' : a \qquad (\mathrm{FTV}(\pi) \neq \emptyset \wedge a = ?) \vee a = \lfloor \tau_2' \rfloor \\
F_2 = \{\pi \texttt{ has } f(x\!:\!\tau_1') \to \tau_2'\{e'\}\}
\end{array}
}{
T_1; F_1 \vdash \pi \texttt{ has } f(x\!:\!\tau_1) \to \tau_2\{e\} \Downarrow T_3; F_1 \cup F_2
}
$$

4.2 Program instantiation

The definition of the program instantiation judgment is in Fig. 6 and the following describes each rule.

(C-Nil): Program instantiation is finished when there are definitions for all of the functions used in the program. The symbol [] is used for the empty list.

(C-InstFun): Once the entire program has been processed we instantiate member functions that are used but not yet defined. 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 apply the (MemFun) rule to the member function and then process the rest of the program.

Fig. 6. Program instantiation. $T; F \vdash p \Downarrow T; F$

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

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

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

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

4.3 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 6 (Type soundness for evaluation). The following lemma captures the invariants that are maintained during program instantiation to achieve this goal.

Lemma 7. *(Instantiation produces a well-typed program) If $T; F \vdash p \Downarrow T'; F'$, 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 3 (Properties of type evaluation) and and Lemma 4 (Environment weakening for well-typed expressions). The cases for member function definitions and member function instantiation also use Proposition 3 and Lemma 4. In addition they use Lemma 1 (Member lookup produces closed types).

The proof of the type-safety theorem is a straightforward use of Lemma 7 (Instantiation produces a well-typed program) and Lemma 6 (Type safety of expression evaluation).

Theorem 1. *(Type Safety) If*

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

then for all n, $F; T \vdash \texttt{Main}\langle\rangle.\texttt{main(n)} \hookrightarrow_n ans$ *and either* $ans = \mathsf{Abort}$ *or there exists* v *such that* $ans = v$ *and* $T; \emptyset; \emptyset \vdash v : \texttt{int}$.

Proof. From $\texttt{Main}\langle\rangle\{\texttt{main} : \texttt{int} \rightarrow \texttt{int}\} \in T$ we have $T; \emptyset; \emptyset \vdash \texttt{Main}\langle\rangle.\texttt{main(n)} : \texttt{int}$. By Lemma 7 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 $\texttt{Main}\langle\rangle$ `has` $\texttt{main(x:int)} \rightarrow \texttt{int}\{e\} \in F$ we know that the function used in $\texttt{Main}\langle\rangle.\texttt{main(n)}$ is defined, so we apply Lemma 6 to obtain the conclusion.

5 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.

6 Related work

Recently, Stroustrup and Dos Reis proposed a formal account of the type system for C++ [13, 14]. 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. [16] 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 [20].

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 [15]. These languages provide first-class support for program generation by including a hygienic quasi-quotation mechanism. The advanced type system used in MetaOCaml guarantees that the generated code is typable. There are no such guarantees in C++. The formal semantics defined in this paper will facilitate comparing C++ with languages such as MetaOCaml and will help in finding ways to improve C++. The C++ Standards Committee has begun to investigate improved support for metaprogramming [18].

7 Conclusion

This paper presents a formal account of C++ templates. We identify a small subset, named C++.T, that includes templates, specialization, and member func-

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

tions. 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 encountered 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. However, evaluation of open terms is not allowed. We plan to investigate how this affects the expressivity of C++ templates as a mechanism for writing program generators.

Acknowledgments

We thank the anonymous reviewers for their careful reading of this paper. We thank Emir Pasalic and Ronald Garcia for reading drafts and suggesting improvements and we thank Jaakko Järvi, Doug Gregor, Todd Veldhuizen, and Jeremiah Willcock for helpful discussions. We especially thank John Spicer and Daveed Vandevoorde of Edison Design Group for help with interpreting the C++ Standard.

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] Erik Ernst, Klaus Ostermann, and William R. Cook. A virtual class calculus. In *POPL'06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 270–282, New York, NY, USA, 2006. ACM Press.

[7] 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.

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

[9] Xavier Leroy. The Objective Caml system: Documentation and user's manual, 2000. With Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon.

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

[11] Tobias Nipkow. Structured proofs in Isar/HOL. In *Types for Proofs and Programs (TYPES 2002)*, number 2646 in LNCS, 2002.

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

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

[14] Gabriel Dos Reis and Bjarne Stroustrup. Specifying c++ concepts. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 295–308, New York, NY, USA, 2006. ACM Press.

[15] 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.

[16] 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.

[17] Jeremy Siek and Walid Taha. C++.T formalization in Isar. Technical Report TR05-458, Rice University, Houston, TX, December 2005.

[18] 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.

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

[20] Charles Wallace. *Specification and validation methods*, chapter The semantics of the C++ programming language, pages 131–164. Oxford University Press, Inc., New York, NY, USA, 1995.

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