Support for the Evolution of C++ Generic Functions

Peter Pirkelbauer, Damian Dechev, and Bjarne Stroustrup

Department of Computer Science and Engineering
Texas A&M University
College Station, TX 77843-3112
peter.pirkelbauer@tamu.edu ddechev@sandia.gov bs@cse.tamu.edu

Abstract. The choice of requirements for an argument of a generic type or algorithm is a central design issue in generic programming. In the context of C++, a specification of requirements for a template argument or a set of template arguments is called a *concept*.
In this paper, we present a novel tool, TACE[1], designed to help programmers understand the requirements that their code de facto imposes on arguments and help simplify and generalize those through comparisons with libraries of well-defined and precisely-specified concepts. TACE automatically extracts requirements from the body of template functions. These requirements are expressed using the notation and semantics developed by the ISO C++ standards committee. TACE converts implied requirements into concept definitions and compares them against concepts from a repository. Components of a well-defined library exhibit commonalities that allow us to detect problems by comparing requirement from many components: Design and implementation problems manifest themselves as minor variations in requirements. TACE points to source code that cannot be constrained by concepts and to code where small modifications would allow the use of less constraining concepts. For people who use a version of C++ with concept support, TACE can serve as a core engine for automated source code rejuvenation.

1 Introduction

A fundamental idea of generic programming is the application of mathematical principles to the specification of software abstractions [1]. ISO C++ [2][3] supports generic programming through the use of templates. Unfortunately, it does not directly support the specification of requirements for arguments to generic types and functions [4]. However, research into language-level support for specifying such requirements, known as *concepts*, for C++ has progressed to the point their impact on software can be examined [5][6][7]. Our work is aimed at helping programmers cope with the current lack of direct support for concepts and to ease the future transition to language-supported concepts.

[1] template analysis and concept extraction

Templates are a compile-time mechanism to parameterize functions and classes over types. When the concrete template argument type becomes known to the compiler, it replaces the corresponding type parameter (template instantiation), and type checks the instantiated template body. This compilation model is flexible, type safe, and can lead to high performance code [8]. For over a decade, C++ templates have helped deliver programs that are expressive, maintainable, efficient, and organized into highly reusable components [9]. Many libraries, such as the C++ Standard Template Library (STL) [10], the BOOST graph library [11], STAPL [12], for which adaptability and performance are paramount rest on the template mechanism.

C++ currently does not allow the requirements for the successful instantiation of a template to be explicitly stated. Instead, such requirements must be found in documentation or inferred from the template body. An attempt to instantiate a template with types that do not meet its requirements fails with error messages that can be very hard to understand [5]. Also, C++ provides only weak support for overloaded templates. A number of programming techniques [13][14][15][16] offer partial solutions to these problems, but they tend to raise the level of complexity of template implementations and can make programs harder to understand.

Concepts [5][6][7] provide systematic remedies and deliver better support for the design and development of generic programs. Concepts improve the expressiveness, make error messages more precise, and provide better control of the compile-time resolution of templates. Importantly, the use of concepts does not incur runtime overhead when compared to templates not using concepts. Unfortunately, concerns about usability, scalability, and the time needed to stabilize a design prevented concepts from being included in the next revision of C++ [17].

In this paper, we present a novel tool for template analysis and concept extraction, TACE, that addresses some of these concerns. TACE extracts concept requirements from real world C++ code and helps apply concepts to unconstrained templated code. The paper offers the following contributions:

- A strategy for evolving generic code towards greater generality, greater uniformity, and more precise specification.
- Type level evaluation of uninstantiated template functions and automatic extraction of sets of requirements on template arguments.
- Concept specifications that depend on other generic functions.

Experience with large amounts of generic C++ code and the development of C++ generic libraries, such as the generic components of the C++0x standard library [18] shows that the source code of a class template or a function template is not an adequate specification of its requirements. Such a definition is sufficient for type safe code generation, but even expert programmers find it hard to provide implementations that do not accidentally limit the applicability of a template (compared to its informal documentation). It is also hard to precisely specify template argument requirements and to reason about those.

Consequently, there is wide agreement in the C++ community that a formal statement of template argument requirements in addition to the template body

is required. Using classical type deduction techniques modified to cope with C++, TACE generates such requirements directly from the code to enable the programmer to see the implications of implementation decisions. Furthermore, the set of concepts generated from an implementation is rarely the most general or the simplest. To help improve the design TACE compare the generated (implied) concepts to a library concepts (assumed to be well designed).

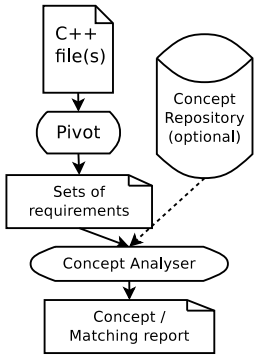

Fig. 1. The TACE tool chain

Fig. 1 shows TACE's tool chain. TACE utilizes the Pivot source-to-source transformation infrastructure [19] to collect and analyze information about C++ template functions. The Pivot's internal program representation (IPR) preserves high-level information present in the source code - it represents uninstantiated templates and is ready for concepts. TACE analyzes expressions, statements, and declarations in the body of template functions. Since function instantiations do not have to be present, TACE can operate on modularily defined template libraries. It evaluates the template body on the type level and extracts the requirements on template arguments. TACE alerts programmers on code, where a requirement cannot be modeled with concepts. It merges the requirements with requirements extracted from functions that the template body potentially invokes. The resulting sets of requirements can be written out as concept definitions.

Our experiments demonstrate that TACE can extract requirements from individual functions. However, our goal is to find higher-level concepts that prove useful at the level of the design of software libraries. TACE achieves this by matching the extracted sets of requirements against concepts stored in a concept repository (e.g., containing standard concepts). In addition to reporting matches, the tool also reports close misses. In some cases, this allows programmers to reformulate their code to facilitate types that model a weaker concept. Our test results for STL indicate that our tool is effective when used in conjunction with a concept repository that contains predefined concepts.

The rest of the paper is organized as follows: §2 provides an overview of concepts as proposed for C++0x, §3 presents the extraction of requirements from function bodies, §4 describes the requirement analysis and how to form concepts for individual functions; §5 describes matching the extracted requirements against predefined concepts from a repository, §7 puts this work in context to related work, and §8 provides a conclusion and outlook to subsequent work.

2 Concepts for C++

Concepts as designed for C++ [6][5][7] provide a mechanism to express constraints on template arguments as sets of syntactic and semantic requirements.

Syntactic requirements describe requirements such as associated functions, types, and templates that are necessary for the template instantiation to succeed. Consider the following template, which determines the distance between two iterators:

```
template<typename Iterator>
size_t distance(Iterator first, Iterator last) {
  size_t n = 0;
  while (first != last) { ++first; ++n; }
  return n;
}
```

The function `distance` requires types that substitute for the type parameter `Iterator` have a copy constructor (to copy the arguments), an inequality (`!=`) operator, and an increment (`++`) operator. A requirement's argument type can be derived from the source code. Requirements can be stated using a C++ function signature like notation.

```
concept DistanceIterator<typename T> {
  T::T(const T&); // copy constructor
  bool operator!=(T, T);
  void operator++(T);
}
```

In order not to over-constrain templates, function signatures of types that model the concept need not match the concept signature exactly. C++'s pseudo signatures allow automatic conversions of argument types. This means that an implementation of `operator!=` can accept types that are constructable from `T`.

The return type of a function has to be named but can remain unbound. In the following example, the concrete result type of `operator*` is irrelevant, as long as there exists an `operator==` that handles the result type together with an `int` argument.

```
template <ZeroInit T>
bool is_zero(T t) {
  return (*t == 0);
}
```

Concepts introduce *associated types* to model such types. The following concept definition introduces an associated type `ResDeref` to specify the return type of `operator*`. Associated types can be constrained by nested requirements (e.g., the `requires` clause).

```
concept ZeroInit<typename T> {
  typename ResDeref;

  // nested requirements
  requires Constructable<T, int>; // providing constructor ResDeref(int)
  requires EqualityComparable<ResDeref>; // providing operator==

  ResDeref operator*(T); // provides deref operator*
}
```

The compiler will use the concept definitions to type check expressions, declarations, and statements of the template body without instantiating it. Moreover, any type (or combination of types) that fulfills the concept requirements can be used to instantiate the template bodies.

Semantic requirements describe behavioral properties, such as the equivalence of operations or runtime complexity. Types that satisfy the semantic requirements are guaranteed to work properly with a generic algorithm. Axioms model some behavioral properties. In the following example, the axiom `indirect_deref` specifies that the operations of the left and right side of `<=>` are equivalent. Compilers are free to use axioms for code optimizations.

```
concept Pointer<typename T> {
  typename data;
  data operator*(T);
  T operator+(T, size_t);
  data operator[](T, size_t);

  axiom indirect_deref(T t, size_t n) {
    t[n] <=> *(t+n);
  }
}
```

Concepts can extend one or more existing concepts and add new requirements. Any requirement of the "base" concept remains valid for its *concept refinements*.

```
concept BidirectionalIterator<typename T> {
  requires ForwardIterator<T>;
  ...
}
```

Concept refinements are useful for the implementation of a family of generic functions. A base implementation constrains its template arguments with a general concept, while specialized versions exploit the stronger requirements of concept refinements to provide more powerful or more efficient implementations. Consider, the STL algorithm `advance(Iter, Size)` for which three different implementations exist. Its basic implementation is defined for input-iterators and

has runtime complexity $O(Size)$. The version for bidirectional-iterators can handle negative distances, and the implementation for random access improves the runtime complexity to $O(1)$. The compiler selects the implementation according to the concept a specific type models [20].

Concepts can be used to constrain template arguments of stand-alone functions. In such a scenario, the extracted concept requirements reflect the function implementation directly. In the context of template libraries, clustering similar sets of requirements yields reusable concepts, where each concept constrains a family of types that posses similar qualities. Clustering requirements results in fewer and easier to comprehend concepts and makes concepts more reusable. An example of concepts, refinements, and their application is STL's iterator hierarchy, which groups iterators by their access capabilities.

3 Requirement Extraction

TACE extracts individual concept requirements from the body of template functions by evaluating declarations, statements, and expressions on the type level. Similar to the usage pattern style of concept specification [8], we derive the requirements by reading C++'s evaluation rules [21] backwards. The type level evaluation yields a set of requirements ζ that reflect functional requirements and associated typenames.

3.1 Evaluation of Expressions

A functional requirement $op(arg_1, \ \ldots, \ arg_n) \rightarrow res$ is similar to a C++ signature. It consists of a list of argument types (arg) and has a result type (res). Since the concrete type of template dependent expressions is not known, the evaluator classifies the type of expressions into three groups:
Concrete types: this group comprises all types that are legal in non template context. It includes built-in types, user defined types, and templates that have been instantiated with concrete types. We denote types of this class with C.
Named template dependent types: this group comprises named but not yet known types (i.e., class type template parameters, dependent types, associated types, and instantiations that are parametrized on unknown types), and their derivatives. Derivatives are constructed by applying pointers, references, `const` and `volatile` qualifiers on a type. Thus, a template argument `T`, `T*`, `T**`, `const T`, `T&`, `typename traits<T>::value_type` are examples for types grouped into this category. We denote types of this class with T.
Requirement results: This group comprises fresh type variables. They occur in the context of evaluating expressions where one or more subexpressions have a non concrete type. The symbol R denotes types of this class. The types R are unique for each operation, identified by name and argument types. Only the fact that multiple occurrences of the same function must have the same return type, enables the accumulation of constraints on a requirement result. (e.g., the STL algorithm `search` contains two calls to `find`).

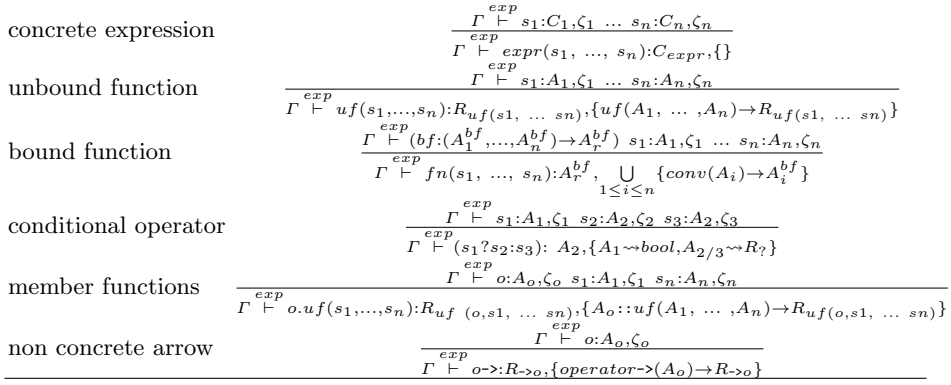

Fig. 2. Evaluation rules of expressions

In the ensuing description, we use N for non concrete types ($T \cup R$) and A for any type ($N \cup C$). For each expression, the evaluator yields a tupel consisting of the return type and the extracted requirements. For example, $expr : C, \zeta$ denotes an expression that has a concrete type and where ζ denotes the requirements extracted for *expr* and its subexpressions. We use $X \rightsquigarrow Y$ to denote type X be convertible to type Y. Fig. 2 and Fig. 3 shows TACE's evaluation rules of expressions in a template body. Note, that the rules in Fig. 2 and Fig. 3 only show new requirements, but omit the union of requirements (ζ) extracted from subexpressions.

A *concrete expression* (*expr*) is an expression that does not depend on any template argument (e.g., literals, or expressions where all subexpressions (s) have concrete type). The result has a concrete type. ζ contains the union of the requirements of its subexpressions.

Calls to *unbound functions* (uf) (and unbound overloadable operators, constructors, and destructor) have at least one argument that depends on an unknown type N. Since uf is unknown, its result type is denoted with a fresh type variable $R_{uf\ (s1,\ \dots\ sn)}$.

Calls to *bound functions* (bf) have the result type that they specify. bf's specification of parameter and return types can add conversion requirements to ζ (i.e., when the type of a subexpression differs from the specified parameter type and when at least one of these types is not concrete.)

The *conditional operator* (`?:`) cannot be overloaded. The ISO standard definition requires the first subexpression be convertible to `bool`. The types of the second and third subexpression either have to be the same, or exactly one conversion that results in the two arguments having the same type has to exist.

Other not overloadable operators (i.e., `typeid` and `sizeof`) do not add any constraints and are evaluated according to regular C++ rules.

C++ concepts do not support modeling of member variables. Thus the application of a member selection (i.e, the `dot` or `arrow operator` can only refer to a member function name. The type evaluator rejects any dot expression that

occurs not in the context of evaluating the receiver of a call expression. Any member selection operation is transformed into an unbound member function call.

For *non concrete objects*, the evaluator treats the `arrow` as a unary operator that yields an object of unknown result type. The object becomes the receiver of a subsequent call to an unbound member function.

static cast $\dfrac{\Gamma \overset{exp}{\vdash} A_{tgt},\zeta_{tgt}\ o{:}A_o,\zeta_o}{\Gamma \overset{exp}{\vdash} A_{tgt},\{A_o \rightsquigarrow A_{tgt}\}}$

dynamic cast $\dfrac{\Gamma \overset{exp}{\vdash} A_{tgt},\zeta_{tgt}\ o{:}A_o,\zeta_o}{\Gamma \overset{exp}{\vdash} A_{tgt},\{PolymorphicClass{<}A_o{>}\}}$

other casts $\dfrac{\Gamma \overset{exp}{\vdash} A_{tgt},\zeta_{tgt}\ o{:}A_o,\zeta_o}{\Gamma \overset{exp}{\vdash} A_{tgt},\{\}}$

Fig. 3. Evaluation rules of cast expressions

`Cast` expressions are evaluated according to the rules specified in Fig. 3. The target type of a cast expression is also evaluated and can add dependent name requirements to ζ. A `static_cast` requires the source type be convertible to the target type. A `dynamic_cast` requires the source type to be a polymorphic class (`PolymorphicClass` is part of C++ with concepts).

The evaluation of operations on pointers follows the regular C++ rules, thus the result of dereferencing $T*$ yields $T\&$, the arrow operator yields a member function selection of T, taking the address of $T*$ yields T**, and any arithmetic expression on $T*$ has type $T*$. *Variables* in expressions are typed as lvalues of their declared type.

3.2 Evaluation of Declarations and Statements

statement context $\dfrac{\Gamma \overset{stmt}{\vdash} \tau{\in}N,s{:}A,\zeta}{\epsilon,\zeta{+}A{\rightsquigarrow}\tau}$

default ctor $\dfrac{\Gamma \overset{decl}{\vdash} o{:}(\Gamma,\tau{\in}N_o)}{\Gamma \overset{decl}{\vdash} o{:}\tau,\{\tau{::}ctor()\}}$

single argument ctor $\dfrac{\Gamma \overset{decl}{\vdash} o{:}(\Gamma,\tau{\in}N_o),s_1{:}A_1,\zeta_1}{\Gamma \overset{decl}{\vdash} o{:}\tau,\zeta_1{+}\tau{::}ctor(const\tau\&){+}A_0{\rightsquigarrow}\tau}$

constructor $\dfrac{\Gamma \overset{decl}{\vdash} o{:}(\Gamma,\tau{\in}N_o),s_1{:}A_1,\zeta_1,\ \ldots,\ s_n{:}A_n,\zeta_n}{\Gamma \overset{decl}{\vdash} o{:}\tau,\ \bigcup\limits_{1\le i\le n} \zeta_i{+}\tau{::}ctor(A_1,\ \ldots,\ A_n)}$

parameter $\dfrac{\Gamma \overset{decl}{\vdash} p{:}(\Gamma,\tau{\in}N_o)}{\Gamma \overset{decl}{\vdash} p{:}\tau,\{\tau{::}ctor(A_1,\ \ldots,\ A_n)\}}$

Fig. 4. Evaluation rules of declarations

We derive the following constraints from the evaluation of statements and declarations (of variables and parameters):
Statements: The condition expressions of `if`, `while`, `for` require the expression to be convertible to `bool`. The `return` statement requires convertibility of the expression to the function return type. The expression of the `switch` statement is either convertible to `signed` or `unsigned` integral types. We introduce an artificial type `Integer` that subsumes both types. The type will be resolved later, if more information becomes available.
Object declarations: Variable declarations of object type require the presence of a constructor. Constructions with a single argument (i.e., $T\ t\ =\ arg$) are modeled to require $A_{arg} \rightsquigarrow T$ and a copy constructor on T.
References: Bindings to lvalue (mutable) references (i.e., declarations, function calls, and `return` statements) imposer stricter requirements. Instead of convertibility, they require the result type of an expression be an exact type (instead of a convertible type).

3.3 Evaluation of Class Instantiations

The current implementation focuses on extracting requirements from functions, and thus treats any instantiation of classes that have data members and where the template parameter is used as template argument as concrete type (e.g. `pair`, `reverse_iterator`); ζ remains unchanged. To allow the analysis of real world C++ template functions, TACE analyses classes that contain static members (types, functions, and data). Particularly, trait classes can add dependent type requirements to ζ. For example, the instantiation of `iterator_traits<T>::value_type` leads to the type constraint `T::value_type`.

Static (templated) member functions of templated classes (e.g.: the various variants of `sort`) are treated as if they were regular template functions. The template parameters of the surrounding class extend the template parameters of the member function. For example:

```
template <class T>
struct S { template <class U> static T bar(U u); };
```

is treated as:

```
template <class T, class U> T bar(U u);
```

3.4 Examples

We use GCC's implementation (4.1.3) of the STL function `search` to illustrate our approach. The function and concept declarations start with:

```
template<typename FwdIter1, typename FwdIter2>
FwdIter1
search(FwdIter1 first1, FwdIter1 last1, FwdIter2 first2, FwdIter2 last2) {

concept Search <typename FwdIter1, typename FwdIter2> {
  FwdIter1::FwdIter1(const FwdIter1&); // argument construction
  FwdIter2::FwdIter2(const FwdIter2&); // argument construction
```

We begin by extracting the requirements from the first statement:

```
// STL search
if (first1 == last1 || first2 == last2)
  return first1;
```

The condition of the `if` statement consists of an `operator||` call that has two arguments. The left and right arguments are produced by calls to `operator==` that in turn take two arguments of `FwdIter1` and `FwdIter2` respectively. `FwdIter1` is an unknown type, thus the specific `operator==` and its result type are unknown; the requirement $operator{==}(FwdIter1\&, FwdIter1\&) \to r_1$ is added to ζ_{search}. The $operator{==}$ on the right hand side of $operator\|\|$ is treated likewise. From the evaluation of the first two expressions, we determine the input arguments for `operator||` to be r_1, r_2, the result type is r_3. Since r_3 is evaluated as condition, it must be convertible to `bool`. The return statement does not produce any new requirement, because the copy constructor of `FwdIter1` is already part of ζ_{search}.

```
// requirements of search
typename r1;
r1 operator==(FwdIter1&, FwdIter1&);
typename r2;
r2 operator==(FwdIter2&, FwdIter2&);
typename r3;
r3 operator||(r1, r2);
operator bool(r3);
```

We proceed with the source code of `search`:

```
// STL search (cont'd)
FwdIter2 tmp(first2);
++tmp;
if (tmp == last2)
  return find(first1, last1, *first2);
```

After adding $operator{++}(FwdIter2\&) \to r_4$, the evaluator encounters the `if` condition. The requirement $operator{==}(FwdIter2\&, FwdIter2\&) \to r_2$ is already in ζ_{search}, but the context of evaluating a condition expression leads to the new requirement $r_2 \rightsquigarrow bool$. Evaluating the argument list yields the requirement $operator * (FwdIter2\&) \to r_7$. The call to `find` results in the requirement $find(FwdIter1\&, FwdIter1\&, r_7) \to r_8$, and the context of the `return` statement adds $r_8 \rightsquigarrow FwdIter1$.

```
// requirements of search (cont'd)
r5 operator++(FwdIter2&);
operator bool(r2);
typename r7;
r7 operator*(FwdIter2&);
typename r8;
r8 find(FwdIter1&, FwdIter1&, r7);
operator FwdIter1(r8);
```

We continue with the source code of `search`:

```
// STL search (cont'd)
FwdIter2 p1, p;
p1 = first2;
++p1;
FwdIter1 current = first1;
```

The declarations of `p1` and `p` require the presence of a default constructor on `FwdIter2` and assignment operator.

```
// requirements of search (cont'd)
FwdIter2::FwdIter2(); // default constructor
void operator=(FwdIter2&, FwdIter2&);
```

We skip the remaining code and requirements.

4 From Requirements to Concepts

This section discusses the analysis and manipulation of the extracted requirements with the goal to print out *function specific concepts*.

4.1 Function Calls

Consider the requirements that were extracted from `search` (§3.4). It contains two calls to a function `find`, an unbound non-member call that potentially (or likely) resolves to STL's templated function. We have a choice, how we can handle such functions.

• A simplistic approach (§3.4) could print the concept `Search` and represent the calls to find as functional requirement.

```
typename r8;
r8 find(FwdIter1&, FwdIter1&, r7);
operator FwdIter1(r8);
void operator=(FwdIter1&, r8);
```

• Another approach would replace the requirements related to `find` with a simple refinement clause, and eliminate the requirements on `r8` (the conversion requirement becomes obsolete, and `operator=(FwdIter1, FwdIter1)` already exists in `search`).

```
requires Find<FwdIter1, r7>;
```

Both approaches lead to (deeply) nested requirements. `Search` does not expose the requirements on `FwdIter1` (or on the combination of `FwdIter1` with `r7`) that stem from `Find`. Thus, we would expect requirement errors that stem from deeply nested templated function calls to remain hard to understand for programmers.

• TACE's approach is to merge the requirements of the callee into the caller, if a callee exists. A callee exists, when there is a template function with the same name defined in the same namespace, and when that function's parameter are at least as general as the arguments of the call expression. (e.g., `search` calls `find`).

4.2 Result Type Reduction

In the extracted set of requirements, any result of an operation is represented as an unnamed type requirement (i.e., associated types such as `r1` and `r2` in §3.4). However, the evaluation context contributed more information about these types in the form of conversion requirements. TACE invokes a function *reduce* that propagates the target types of conversions.

$$reduce(\zeta) \rightarrow \zeta'$$

Should a requirement result have more than one conversion targets (for example, an unbound function was evaluated in the context of `bool` and `int`), we apply the following subsumption rule: assuming n conversion requirements with the same input type (R) but distinct target types A_i.

$$R' = \begin{cases} A_j & \text{if } \exists_j \forall_i \text{ such that } A_j \rightsquigarrow A_i \\ R & \text{otherwise} \end{cases}$$

Note, that the $A_j \rightsquigarrow A_i$ must be part of ζ, or defined for C++ built in types. If such an A_j exists, all operations that depend on R are updated, and become dependent on A_j. Any conversion requirement on R is dropped from ζ. When R is not evaluated by another function it gets the result type `void`. If R is evaluated by another expression, but no conversion requirement exists, the result type R' remains unnamed (i.e. becomes an associated type).

After the return type has been determined, the new type R' is propagated to all operations that use it as argument type. By doing so, the set of requirements can be further reduced (e.g., if all argument types of an operation are in C, the requirement can be eliminated, or in case the operation does not exist, an error reported) and more requirement result types become named (if an argument type becomes T, another operation on T might already exist). Reduction is a repetitive process that stops when a fixed point is reached.
For example, *reduce* reduces the set of requirements that we got from merging `search` and `find`:

```
concept search <typename FwdIter1, typename FwdIter2> {
  FwdIter1::FwdIter1(const FwdIter1&);
  FwdIter2::FwdIter2(const FwdIter2&);
  bool operator==(FwdIter1&, FwdIter1&);
  bool operator==(FwdIter2&, FwdIter2&);
  typename r4;
  r4 operator++(FwdIter2&);
  typename r5;
  r5 operator*(FwdIter2&);
  FwdIter2::FwdIter2();
  void operator=(FwdIter2&, FwdIter2&);
  bool operator!=(FwdIter1&, FwdIter1&);
  void operator=(FwdIter1&, FwdIter1&);
  typename r11;
  r11 operator++(FwdIter1&);
```

```
  bool operator==(r11, FwdIter1&);
  typename r13;
  r13 operator*(FwdIter1&);
  bool operator==(r13, r5);
  bool operator==(r4, FwdIter2&);

  typename r529;          // from find
  r529 operator==(r13, const r5&);
  typename r530;
  r530 operator!(r529);
  bool operator&&(bool, r530);
}
```

Due to the presence of the conversion `operator FwdIter1(r8)`, `FwdIter1&` substitutes for `r8` in `void operator=(FwdIter1&, r8)`. Similar `r1` and `r2` become convertible to `bool`, thus the requirement `r3 operator||(r1, r2);` is dropped.

The result of `reduce` may constrain the type system more than the original set of requirements. Thus, `reduce` has to occur after merging all requirements from potential callees, when all conversion requirements on types are available.

5 Recovery From Repository

Template libraries utilize concepts to constrain the template arguments of a group of functions that operate on types with similar capabilities. This requires clustering similar sets of requirements into concepts. We tackle this by using a concept repository, which contains a number of predefined concept definitions (e.g., core concepts or concepts that users define for specific libraries). The use of a concept repository offers users the following benefits:

- reduces the number of concepts
- improves the structure of concepts
- exposes the refinement relationships of concepts, which allows for more exact analysis
- replaces requirement results with named types (concrete, template dependent, or associated typenames)

The repository we use to drive the examples in this sections contains the following concepts: `IntegralType<T>`, `RegularType<T>`, `ForwardIterator<T>`, `BidirectionalIterator<T>`, `RandomaccessIterator<T>`, and `EqualityComparable<T>`.

5.1 Concept Kernel

In order to match the extracted requirements of each template argument against concepts in the repository that depend on fewer template arguments, we partition the unreduced set into smaller sets called *kernels*. We define a concept kernel over a set of template arguments $\widehat{T}$ to be a subset of the original set of requirements ζ.

$$kernel(\zeta_{function}, \widehat{T}) \rightarrow \zeta_{kernel}$$

ζ_{kernel} is a projection that captures all operations on types that directly or indirectly originate from the template arguments in $\widehat{T}$.

$$\zeta_{kernel} \Leftrightarrow \{op | op \in \zeta_{src}, \phi_{\widehat{T}}(op)\}$$

For the sake of brevity, we also say that a type is in ζ_{kernel}, if the type refers to a result R of an operation in ζ_{kernel}.

$$\phi_{\widehat{T}}(op) = \begin{cases} 1 & \text{true for a } op(arg_1, \ldots, arg_n) \rightarrow res \\ & \text{if } \forall_i \; arg_i \in \widehat{T} \cup \zeta_{kernel} \cup C \\ 0 & \text{otherwise} \end{cases}$$

$\phi_{\widehat{T}}(op)$ is true, if all arguments of op either are in $\widehat{T}$, are result types from operations in ζ_{kernel}, or are concrete.

As an example, we give the concept kernel for the first template argument of `search`.

```
FwdIter1::FwdIter1(const FwdIter1&);
typename r1652;
r1652 operator==(FwdIter1&, FwdIter1&);
typename r1654;
r1654 operator!=(FwdIter1&, FwdIter1&);
operator bool (r1654);
operator bool (r1652);
void operator=(FwdIter1&, FwdIter1&);
typename r1658;
r1658 operator++(FwdIter1&);
typename r1659;
r1659 operator==(r1658, FwdIter1&);
operator bool (r1659);
typename r1661;
r1661 operator*(FwdIter1&);
```

5.2 Concept Matching

For each function, TACE compares the kernels against the concepts in the repository. The mappings from a kernel's template parameters to a concept's template parameters are generated from the arguments of the first operation in a concept kernel and the parameter declarations of operations with the same name in the concept.

For any requirement in the kernel, a concept has to contain a single best matching requirement (multiple best matching signatures indicate an ambiguity). Matching takes into account the usual C++ binding rules and all conversion requirements defined in the concept. TACE checks the consistency of a concept requirement's result type with all conversion requirements in the kernel.

If checking succeeds, TACE updates the concept's result type in the kernel's requirements.

For each kernel and concept pair, TACE partitions the requirements into satisfiable, unsatisfiable, associated, and available functions. An empty set of unsatisfiable requirements indicates a match. TACE can report small sets of unsatisfiable requirements (i.e., near misses), thereby allowing users to modify the function implementation (or the concept in the repository) to make a concept-function pair work together. The group of associated requirements contains unmatched requirements on associated types. For example, any iterator requires the value type to be regular. Besides regularity, some functions such as `lower_bound` require less than comparability of container elements. Associated requirements are subsequently matched. The group of available functions contains requirements, where generic implementations exist.

This produces a set of candidate concepts. For example, the three iterator categories match the template parameters of `search`. `find` has two template arguments. Any iterator concept matches the first argument. Every concept in the repository matches the second argument.

TACE reduces the set of candidates by eliminating all refinements, for which the refinee is also a candidate. In our case, the concepts for `bidirectional-` and `randomaccess- iterators` are eliminated from the candidate set.

To eliminate more false positive matches, TACE also analyzes the candidate set in context of available functions. Any valid candidate, has to satisfy the requirements of a callee's candidate. Consider `search`'s (§3.4) call of `find`. Any concept that does not match `find`'s first template parameter cannot match `search`'s first parameter.

Conversely, for functions that continue having multiple candidates, TACE eliminates those candidates that, if chosen, would invalidate callers. Consider the function `advance` and its two implementations `__advance` for input iterator and bidirectional iterator. Besides the iterator concepts the template argument of `__advance` has a false positive match in number concepts (e.g., IntegralType). `advance` employs the tag dispatching idiom, which requires an iterator. Thus, TACE eliminates the non-iterator concepts from the candidates. This heuristic elimination of candidates performs well for the standard template library.

If our tool deals with functions that use only independent template parameters, the analysis is done and reported. For functions with more than one template argument, TACE generates the concept requirements using a Cartesian join of the results of the individual kernels. The following code snippet shows the result for `search`.

```
concept Search<typename FwdIter1, typename FwdIter2> {
  requires ForwardIterator<FwdIter1>;
  requires ForwardIterator<FwdIter2>;

  bool operator==( iterator_traits<FwdIter1>::value_type&,
                   iterator_traits<FwdIter2>::value_type&);
}
```

Note, that matching currently does not generate extra conversion requirements. Thus, the `operator==` with two different argument types does not match the `operator==` defined in `EqualityComparable<T>`. We plan to address this issue in the final version of the paper.

5.3 Families of Functions

A generic function can consist of a family of different implementations, where each implementation exploits concept refinements (e.g., `advance` in §2).

A template that calls a generic function needs to incorporate the minimal requirements of the generic function in its concept. To do so, it is necessary to determine the most general implementation. Finding the base implementation is non trivial with real code. Consider STL's `advance` family. TACE extracts the following requirements:

```
// for Input−Iterators
concept AdvInputIter <typename Iter, typename Dist> {
  Dist::Dist(const Dist&);
  void operator++(Iter&);
  bool operator−−(Dist&, int);
}

// for Bidirectional−Iterators
concept AdvBidirectIter <typename Iter, typename Dist> {
  Dist::Dist(const Dist&);
  void operator++(Iter&);
  void operator−−(Iter&);
  bool operator++(Dist&, int);
  bool operator−−(Dist&, int);
}

// for Randomaccess−Iterators
concept AdvRandomaccessIter <typename Iter, typename Dist> {
  Dist::Dist(const Dist&);
  void operator+=(Iter&, Dist&);
}
```

The sets of extracted requirements for the implementations based on input- and bidirectional-iterator are in a subset/superset relation, the set of requirements for the random access iterator based implementation is disjoint with the former sets.

If calls to generic functions where a refinement relationship cannot be inferred occur under scenario §4, TACE requires the user mark the least specific function implementation. A concept repository helps infer the correct refinement relationship.

However, detecting the least specific implementation can still be problematic for generic functions with a too general definition. Consider the implementations of `advance` for input- and random access iterators.

```
// template <class InputIterator, Class Distance>
```

```
// void advance(InputIterator& iter, Distance dist);
void operator++(InputIterator&); // kernel(InputIterator)
Distance operator--(Distance&,int); // kernel(Distance)
operator bool(Distance&); // kernel(Distance)

// template <class RandomaccessIter, Class Distance>
// void advance(RandomaccessIter& iter, Distance dist);
void operator+=(InputIterator&, Distance&); // multi-parametric
```

The kernel for `RandomaccessIter` is empty, thus any concept matches. After eliminating false positive candidates, any iterator concept matches; the requirement `operator+=` becomes an operation that is defined over two independent template arguments. Recent concept based STL implementations make the type dependence between the two parameters explicit and TACE correctly identifies the random access iterator concept.

6 Results

We validated the approach by matching the functions defined in GCC's header file `algorithm`. The file contains more than 9000 non-comment (and non empty) lines of code and defines 115 algorithms plus about 100 helper functions. The `algorithm` header exercises some of the most advanced language features and design techniques used for generic programming in C++.

The success rate of the concept recovery depends on the concepts in the repository. A repository containing syntactically similar concepts will lead to ambiguous results. We ran the tests against the repository introduced in §5 plus concepts that are defined over multiple template arguments (`UnaryFunction`, `UnaryPredicate`, `BinaryFunction` and `BinaryPredicate`). The predicates refine the functions and require the return type be convertible to `bool`.

TACE found a number of functions, where the annotation in code overly constrain the template arguments, such as `__unguarded_linear_insert` (STL's specifications are meaningful though, as the identified functions are helpers of algorithms requiring random access.) A limitation of TACE's current implementation is that matching does not (and a static analysis tool cannot always) capture the semantic level of concepts. Consider `find_end` for bidirectional iterators, which takes two different iterator types. The second iterator type requires only forward access and the existence of `advance(it, n)`. `n`'s possible negativity is what requires bidirectional access. Over the entire test set, TACE currently recognizes about 75% of iterator concepts correctly and unambiguously. For about 15% TACE produces a false positive match (e.g., Number) alongside the correct iterator concept. TACE recognizes all binary functions, but due to the reason stated at the end of §5.2 does not generate the conversion requirement on result types. The predicates (including template arguments marked as `Compare` and `StrictWeakOrding`) are identified correctly.

7 Related Work

Sutton and Maletic [22] describe an approach to match requirements against a set of predefined concepts based on formal concept analysis. Their analysis tool finds combinations of multiple concepts, if needed, to cover the concept requirements of a template functions. In order to avoid complications from "the reality of C++ programming" [22], the authors validate their approach with a number of self implemented STL algorithms, for which they obtain results that closely match the C++ standard specification. The authors discuss how small changes in the implementation can lead to small variations in their identified concepts. Their work does not provide a formal description of constraint generation and does not mention function families nor functions that utilize other generic functions.

Dos Reis and Stroustrup [8] present an alternative idea for concept specification and checking. Their approach states concepts in terms of usage patterns, a form of requirement specification that mirrors the declarations and expressions in the template body that involve template arguments. If type checking of a concrete type against the usage pattern succeeds, then template instantiation will succeed too. In essence, TACE reverses this process and derives the requirements from C++ source code and converts them into signature based concepts.

The aim of type inference for dynamically typed languages, the derivation of type annotations from dynamically typed code, is somewhat similar to concept recovery. For example, Agesen et al [23]'s dynamic type inference on SELF generates local constraints on objects from method bodies. By analyzing edges along trace graphs their analysis derives global constraints from local constraints. This kind of analysis differs from our work in various ways. Agesen et al start at a specific entry point of a complete program (i.e., function `main` in a C++ program). This provides concrete information on object instantiations from prototypes. Concept recovery neither depends on a single entry point, nor does the analyzed program have to be complete (instantiations are not required). Moreover, concept recovery is not concerned with finding concrete type annotations, but with finding higher level abstractions that describe multiple types (concepts). On the semantic level, C++'s type system differs from dynamically typed languages. C++ allows automatic type conversions, overloading, and function results are typed.

8 Conclusion and Future Work

In this paper, we have presented our tool, TACE, that extracts sets of requirements from real C++ code. TACE analyzes these requirements and generates concept definitions for functions. Alternatively, our tool clusters requirement sets into concepts by matching against predefined concepts in a repository.

At the moment TACE can handle a large number of template functions in STL. The Pivot provides an extensible compiler based framework that enables enhancements of our analysis. In particular, obtaining analysis results for template classes is necessary for the evaluation of compile time values (e.g., `__is_scalar`, tag hierarchies) that are often used to select a specific implementation from a generic function family.

TACE is part of a larger project to raise the level of abstraction of existing C++ programs through the use of high-level program analysis and transformations.

References

1. Stepanov, A., McJones, P.: Elements of Programming. Addison-Wesley Professional (June 2009)
2. ISO/IEC 14882 International Standard: Programming languages: C++. American National Standards Institute (September 1998)
3. Stroustrup, B.: The C++ Programming Language. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2000)
4. Stroustrup, B.: The design and evolution of C++. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA (1994)
5. Gregor, D., Järvi, J., Siek, J., Stroustrup, B., Dos Reis, G., Lumsdaine, A.: Concepts: linguistic support for generic programming in C++. In: OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications, New York, NY, USA, ACM Press (2006) 291–310
6. Becker, P.: Working draft, standard for programming language C++. Technical Report N2914 (June 2009)
7. Gregor, D., Stroustrup, B., Siek, J., Widman, J.: Proposed wording for concepts (revision 4). Technical Report N2501, JTC1/SC22/WG21 C++ Standards Committee (February 2008)
8. Dos Reis, G., Stroustrup, B.: Specifying C++ concepts. In: POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, New York, NY, USA, ACM Press (2006) 295–308
9. Stroustrup, B.: Abstraction and the C++ machine model. In: ICESS'04: 1st International Conference on embedded Software and Systems. Volume 3605 of Lecture Notes in Computer Science., Springer (2004) 1–13
10. Austern, M.H.: Generic programming and the STL: using and extending the C++ Standard Template Library. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1998)
11. Siek, J.G., Lee, L.Q., Lumsdaine, A.: The Boost Graph Library: user guide and reference manual. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2002)
12. An, P., Jula, A., Rus, S., Saunders, S., Smith, T., Tanase, G., Thomas, N., Amato, N., Rauchwerger, L.: Stapl: A standard template adaptive parallel C++ library. In: LCPC '01, Cumberland Falls, Kentucky (Aug 2001) 193–208
13. Abrahams, D., Gurtovoy, A.: C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond (C++ in Depth Series). Addison-Wesley Professional (2004)
14. Alexandrescu, A.: Modern C++ design: generic programming and design patterns applied. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2001)
15. Järvi, J., Willcock, J., Hinnant, H., Lumsdaine, A.: Function overloading based on arbitrary properties of types. C/C++ Users Journal (21(6)) (June 2003) 25–32
16. Siek, J., Lumsdaine, A.: Concept checking: Binding parametric polymorphism in C++. In: First Workshop on C++ Template Programming, Erfurt, Germany. (October 10 2000)

17. Stroustrup, B.: Expounds on concepts and the future of C++. Interview with Danny Kalev (August 2009) www.devx.com/cplus/Article/42448/0/page/1, retrieved on October 1st, 2009.
18. Becker, P.: The C++ Standard Library Extensions: A Tutorial and Reference. 1st edn. Addison-Wesley Professional, Boston, MA, USA (2006)
19. Dos Reis, G., Stroustrup, B.: A principled, complete, and efficient representation of C++. In Suzuki, M., Hong, H., Anai, H., Yap, C., Sato, Y., Yoshida, H., eds.: The Joint Conference of ASCM 2009 and MACIS 2009. Volume 22 of MI Lecture Note Series., Fukuoka, Japan, COE (December 2009) 151–166
20. Järvi, J., Gregor, D., Willcock, J., Lumsdaine, A., Siek, J.: Algorithm specialization in generic programming: challenges of constrained generics in C++. In: PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation, New York, NY, USA, ACM (2006) 272–282
21. Dos Reis, G., Stroustrup, B.: A C++ formalism. Technical Report N1885, JTC1/SC22/WG21 C++ Standards Committee (2005)
22. Sutton, A., Maletic, J.I.: Automatically identifying C++0x concepts in function templates. In: ICSM '08: 24th IEEE International Conference on Software Maintenance, 2008, Beijing, China, IEEE (2008) 57–66
23. Agesen, O., Palsberg, J., Schwartzbach, M.: Type inference of SELF. In: ECOOP '93: Proceedings of the 7th European Conference on Object-Oriented Programming, London, UK, Springer (1993) 247–267