Support for the Evolution of C++ Generic Functions

Peter Pirkelbauer[1], Damian Dechev[2], and Bjarne Stroustrup[3]

[1] Lawrence Livermore National Laboratory
Center for Applied Scientific Computing
Livermore, CA 94550-9698
peter.pirkelbauer@llnl.gov

[2] University of Central Florida
Department of Electrical Engineering and Computer Science
Orlando, FL 32816-2362
dechev@eecs.ucf.edu

[3] Texas A&M University
Department of Computer Science and Engineering
College Station, TX 77843-3112
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 (template analysis and concept extraction), 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 function templates. 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 requirements 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 [22]. ISO C++ [15][24] 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 [23]. However, research into language-level support for specifying such requirements, known as *concepts*, for C++ has progressed to the point where their impact on software can be examined [13][7][14]. Our work is aimed at helping programmers cope with the current lack of direct support for concepts and ease the future transition to language-supported concepts.

Templates are a compile-time mechanism to parameterize functions and classes over types and values. 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 [10]. For over a decade, C++ templates have helped deliver programs that are expressive, maintainable, efficient, and organized into highly reusable components [25]. Many libraries, such as the C++ Standard Template Library (STL) [5], the BOOST graph library [21], and the parallel computation system, STAPL [4], 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. These requirements must be found in documentation or inferred from the template body. For attempts to instantiate a template with types that do not meet its requirements, current compilers often fail with hard to understand error messages [13]. Also, C++ provides only weak support for overloaded templates. While numerous programming techniques [1][3][18][20] offer partial solutions, they tend to raise the complexity.

Concepts [13][7][14] were developed to provide systematic remedies and deliver better support for the design and development of generic programs. As defined for C++0x, concepts improve 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. Despite many years design efforts, implementation work, and experimental use, concerns about usability, scalability, and the time needed to stabilize a design prevented concepts from being included as a language mechanism in the next revision of C++ [27][26]. However, we are left with a notation and a set of concepts developed for the STL and other libraries that can be used to describe and benchmark our use of design-level concepts.

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 industrial-strength 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 function templates and automatic extraction of sets of requirements on template arguments.
- Concept analysis that takes called functions into account.

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 li-

brary [6], shows that the source code a 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 traditional type deduction techniques [8] modified to cope with C++, TACE generates such requirements directly from code. This helps programmers see the implications of implementation choices. Furthermore, the set of concepts generated from an implementation is rarely the most reusable or the simplest. To help validate a library implementation TACE compares the generated (implied) concepts to pre-defined library concepts.

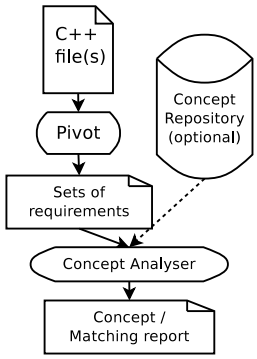

Fig. 1. The TACE tool chain

Fig. 1 shows TACE's tool chain. TACE utilizes the Pivot source-to-source transformation infrastructure [11] to collect and analyze information about C++ function templates. 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 function templates and extracts the requirements on template arguments. 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.

However, our goal is to find higher-level concepts that prove useful at the level of the design of software libraries. In particular, we do not just want to find the requirements of a particular implementation of an algorithm or the absolute minimal set of requirements. We want to discover candidates for concepts that are widely usable in interface specifications for algorithms. To recognize such

concepts we need the "advice" of an experienced human. 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. TACE does not try to discover semantic properties of a concept ("axioms" [12]). In general, doing so is beyond the scope of static analysis. 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 C++ with concepts, §3 presents the extraction of requirements from bodies of template functions, §4 describes requirement integration and reduction; §5 discusses matching the extracted requirements against predefined concepts from a repository, §7 discusses related work, and §8 presents a conclusion and an outlook on subsequent work.

2 Concepts for C++

Concepts as designed for C++0x [13][7][14] 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 allow 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), a destructor (to destruct the argument at the end of the scope), 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++ signature like notation.

```
concept DistanceRequirements<typename T> {
  T::T(const T&); // copy constructor
  T::~T(); // destructor
  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. Signatures in concepts 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 functional requirement has to be named but can remain unbound. The following example shows a function with two parameters of type `T`, where `T` is a template argument constrained by the concept `TrivialIterator`.

The function tests whether the return values of the `operator*` are equal. The type of the return values is irrelevant as long as there exists an `operator==` that can compare the two. The result type of the equality comparison must be convertible to `bool`.

```
template <TrivialIterator T>
bool same_elements(T lhs, T rhs) {
  return (*lhs == *rhs);
}
```

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

```
concept TrivialIterator<typename T> {
  typename ValueType;
  // nested requirements
  requires EqualityComparable<ValueType>; // operator== of ValueType
  ValueType operator*(T); // deref operator*
  ...
}
```

The compiler will use the concept definitions to type check expressions, declarations, and statements of the template body without instantiating it. Any type (or combination of types) that defines the required operations and types is a model of the concept. These types can be used for template instantiation.

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. Axioms can specify the equivalence of operations. The two operations are separated by an operator `<=>`. In the following example, the axiom `indirect_deref` specifies that the operations of the left and right side of `<=>` produce the same result. This is the case for pointers or random access iterators. 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*. Consider a trivial iterator abstraction, which essentially defines an operation to access its element (`operator*`). The concept `ForwardIterator` adds operations to traverse a sequential data structure in one way.

```
concept ForwardIterator<typename T> {
  requires TrivialIterator<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 [16].

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.

Concept requirements are bound to concrete operations and types by the means of *concept maps*. Concept maps can be automatically generated. Should a type's operations not exactly match the requirement definition (e.g., when a function is named differently), concept maps allow for an easy adaptation [17].

3 Requirement Extraction

TACE extracts individual concept requirements from the body of function templates by inferring properties of types from declarations, statements, and expressions. Similar to the usage pattern style of concept specification [10], we derive the requirements by reading C++'s evaluation rules [9] backwards. We say backwards, because type checking usually tests whether expressions, statements, and declarations together with type (concept) constraints result in a well-typed program. In this work, we start with an empty set of constraints and derive the type (concept) constraints that make type-checking of expressions, statements, and declarations succeed. The derived constraints (ζ) 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`).

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 tuple 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 is convertible to type Y. Table 1 shows TACE's evaluation rules of expressions in a template body.

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. The subexpressions have concrete type, but their subexpressions can be template dependent (e.g., `sizeof(T)`). Thus, ζ is the union of subexpression requirements.

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 $R_{uf\ (s_1,\ \ldots\ s_n)}$.

Bound functions are functions, where the type of the function can be resolved at the compile time of the template body. Examples of bound functions include calls to member functions, where the type of the receiver object is known, and calls, where argument dependent lookup [15] is suppressed. Calls to bound functions (*bf*) have the result type of the callee. *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`. TACE's evaluation rules require the second and the third subexpression to be the same type. Here TACE is currently stricter than the ISO C++ evaluation which allows for a conversion of one of the result types.

concrete expression	$\dfrac{\Gamma \overset{exp}{\vdash} s_1{:}C_1,\zeta_1 \ \ldots \ s_n{:}C_n,\zeta_n}{\Gamma \overset{exp}{\vdash} expr(s_1,\ \ldots,\ s_n){:}C_{expr},\ \bigcup\limits_{1\le i\le n}\zeta_i}$
unbound function	$\dfrac{\Gamma \overset{exp}{\vdash} s_1{:}A_1,\zeta_1 \ \ldots \ s_n{:}A_n,\zeta_n}{\Gamma \overset{exp}{\vdash} uf(s_1,\ldots,s_n){:}R_{uf(s1,\ \ldots\ sn)},\ \bigcup\limits_{1\le i\le n}\zeta_i \cup \{uf(A_1,\ \ldots\ ,A_n)\to R_{uf(s_1,\ \ldots\ s_n)}\}}$
bound function	$\dfrac{\Gamma \overset{exp}{\vdash} (bf{:}(A_1^{bf},\ldots,A_n^{bf})\to A_r^{bf})\ \ s_1{:}A_1,\zeta_1 \ \ldots \ s_n{:}A_n,\zeta_n}{\Gamma \overset{exp}{\vdash} fn(s_1,\ \ldots,\ s_n){:}A_r^{bf},\ \bigcup\limits_{1\le i\le n}\zeta_i \cup \{A_i \rightsquigarrow A_i^{bf}\}}$
conditional operator	$\dfrac{\Gamma \overset{exp}{\vdash} s_1{:}A_1,\zeta_1\ \ s_2{:}A_2,\zeta_2\ \ s_3{:}A_2,\zeta_3}{\Gamma \overset{exp}{\vdash} (s_1?s_2{:}s_3){:}\ A_2,\ \bigcup\limits_{1\le i\le 3}\zeta_i \cup \{A_1 \rightsquigarrow bool\}}$
member functions	$\dfrac{\Gamma \overset{exp}{\vdash} o{:}A_o,\zeta_o\ \ s_1{:}A_1,\zeta_1\ \ s_n{:}A_n,\zeta_n}{\Gamma \overset{exp}{\vdash} o.uf(s_1,\ldots,s_n){:}R_{uf\ (o,s_1,\ \ldots\ s_n)},\zeta_o \cup \bigcup\limits_{1\le i\le n}\zeta_i \cup \{uf(A_o,A_1,\ \ldots\ ,A_n)\to R_{uf(o,s_1,\ \ldots\ s_n)}\}}$
non concrete arrow	$\dfrac{\Gamma \overset{exp}{\vdash} o{:}A_o,\zeta_o}{\Gamma \overset{exp}{\vdash} o\texttt{->}{:}R_{\texttt{->}o},\{operator\texttt{->}(A_o)\to R_{\texttt{->}o}\}}$
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	$\dfrac{\Gamma \overset{exp}{\vdash} A_{tgt},\zeta_{tgt}\ \ o{:}A_o,\zeta_o}{\Gamma \overset{exp}{\vdash} A_{tgt},\{\}}$

Table 1. Evaluation rules for expressions

The other not overloadable operators (i.e., `typeid` and `sizeof`) have a concrete result type. The set of extracted requirements is the same as for their subexpression or type.

C++ concepts do not support modeling of member variables. A member selection (i.e, the `dot` or `arrow operator` can only refer to a member function name. The evaluator rejects any dot expression that occurs not in the context of evaluating the receiver of a call expression.

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.

`Cast` expressions are evaluated according to the rules specified in Table 1. 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	$\dfrac{\Gamma \overset{decl}{\vdash} o{:}(\Gamma,\tau{\in}N_o),s_1{:}A_1,\zeta_1,\ ...,\ s_n{:}A_n,\zeta_n}{\Gamma \overset{decl}{\vdash} o{:}\tau,\ \bigcup_{1\le i\le n}\zeta_i{+}\tau{::}ctor(A_1,\ ...,\ A_n)}$
parameter	$\dfrac{\Gamma \overset{decl}{\vdash} p{:}(\Gamma,\tau{\in}N_o)}{\Gamma \overset{decl}{\vdash} p{:}\tau,\{\tau{::}ctor(A_1,\ ...,\ A_n)\}}$

Table 2. Evaluation rules for statements and declarations

Table 2 shows the evaluation rules for 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 and destructor. Single argument construction (i.e., $T\ t\ =\ arg$) is 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) impose 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++ function templates, TACE analyzes 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 member function (templates) of class templates (e.g.: the various variants of `sort`) are treated as if they were regular function templates. 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 Example

We use the beginning of GCC's (4.1.3) implementation of the STL algorithm `search` to illustrate our approach. Fig. 2 shows a portion of the implementation and the requirements that get extracted from it.

We begin by extracting the requirements from the argument list. Their types are `FwdIter1` and `FwdIter2`. They are passed by value. According to the evaluation rule for parameters, their type has to support copy construction and destruction.

The condition of the `if` statement is evaluated bottom up. The right hand side of the `operator||` is an equality comparison (`operator==`) of two parameters of type `FwdIter1&`. Since `FwdIter1` is an unknown type, the operation is evaluated according to the unbound function rule. The result type of operator is a fresh type variable ($r1$). The set of requirements consists of the equality comparison `operator==(FwdIter1&, FwdIter1&)` $\rightarrow$ `r`$_1$. Similarily, the

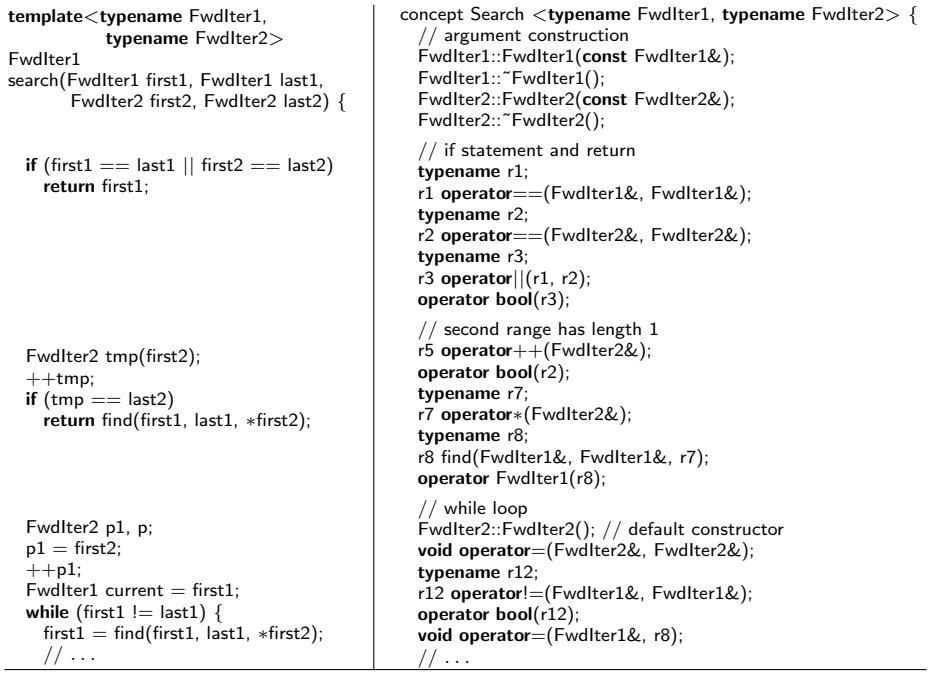

Fig. 2. Requirement extraction

evaluation of the comparison of `first2` and `last2` yields $\mathtt{r}_2$ and the requirement `operator==(FwdIter2&, FwdIter2&)` $\rightarrow \mathtt{r}_2$ The evaluation proceeds with the `operator||`. Both arguments have an undetermined type ($\mathtt{r}_1$ and $\mathtt{r}_2$). Thus, the operation is evaluated according to the unbound function rule. The result type is $\mathtt{r}_3$. `operator==`$(\mathtt{r}_1, \mathtt{r}_2) \rightarrow \mathtt{r}_3$ and the requirements extracted for the subexpressions form the set of requirements. $\mathtt{r}_3$ is evaluated by an if statement. According to the rule statement context, $\mathtt{r}_3$ has to be convertible to `bool`. The return statement does not produce any new requirement, because the copy constructor of `FwdIter1` is already part of $\zeta_{\mathtt{search}}$.

The next source line declares a local variable `tmp`. Its initial value is constructed from a single argument of the same type. Thus, this line requires a copy constructor on `FwdIter2` (evaluation rule for constructors). The next line moves the iterator `tmp` forward by one. The source line is evaluated according to the unbound function rule. The return type is $\mathtt{r}_5$, the extracted requirement is `operator++(FwdIter2&)` $\rightarrow \mathtt{r}_5$. The expression of the following `if` statement compares two expressions of type `FwdIter2&`. Unification with the already extracted requirements in $\zeta_{\mathtt{search}}$ yields the result type $\mathtt{r}_2$. Evaluating $\mathtt{r}_2$ according to the statement context rule yields an additional conversion requirement $\mathtt{r}_2 \rightsquigarrow \mathtt{bool}$. The next line of code returns the result of a function call. First, the argument list of the call is processed. The first two arguments are references to parameters of type `FwdIter1`; the third argument dereferences a parameter of type `FwdIter2`.

According to the unbound function rule, this expression yields to a new result type $\texttt{r}_7$ and the requirement $\texttt{operator*(FwdIter2\&)} \rightarrow \texttt{r}_7$. Then TACE applies the unbound function rule to the function call itself. This yields the result type $\texttt{r}_8$ and the requirement $\texttt{find(FwdIter1\&, FwdIter1\&, r}_7\texttt{)} \rightarrow \texttt{r}_8$. From the statement context, we infer $\texttt{r}_8 \rightsquigarrow \texttt{FwdIter1}$.

The declarations of `p1` and `p` require `FwdIter2` support default construction (`FwdIter2::FwdIter2()`). We skip the remaining code and requirements.

4 From Requirements to Concepts

The requirement extraction generates functional requirements for all calls. This includes calls to algorithms that potentially resolve to other generic functions. For example, the requirements that were extracted from `search` (§3.4) contain two calls to a function `find`. We choose to merge the requirements of the callee into the caller's set of requirements, if there is a function template with the same name and where that function's parameters are at least as general as the arguments of the call expression.

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

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

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

```
// search's requirements
FwdIter1::FwdIter1(const FwdIter1&);
FwdIter1::~FwdIter1();
FwdIter2::FwdIter2(const FwdIter2&);
FwdIter2::~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);
```

Fig. 3. Requirements after reduction

```
// requirements on FwdIter1
FwdIter1::FwdIter1(const FwdIter1&);
FwdIter1::~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&);
```

Fig. 4. Kernel of `FwdIter1`

repetitive process that stops when a fixed point is reached. We show two examples for the requirement set in Fig. 2. The conversion `operator FwdIter1(r8)` allows `FwdIter1&` substitute for `r8` in `void operator=(FwdIter1&, r8)`. Similar `r1` and `r2` are convertible to `bool`, thus `r3 operator||(r1, r2)` can be dropped. The number of reductions depends on how much context information is available in code. Fig. 3 shows the result after merging and reducing the requirements of `search` and `find`.

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. These concepts provide a design vocabulary for library domain and help provide a degree of pluggability among algorithms and types. Without those, even the slightest change to an implementation will cause a recompilation of every user; an implementation is not a good general specification of an algorithm. To find reusable components within the extracted requirements, we use 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 and improves their structure
– exposes the refinement relationships of concepts
– 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>`, `TrivialIterator<T>`, `ForwardIterator<T>`, `BidirectionalIterator<T>`, `RandomaccessIterator<T>`, and `EqualityComparable<T>`. `IntegralType` specifies operations that are defined on type `int`. `RegularType` specifies operations that are valid for all built-in types (i.e., default construction, copy construction, destruction, assignment, equality comparison, and address of). `TrivialIterator` specifies the dereference operation and associated iterator types. The other iterators have the operations defined in the STL. In addition, the repository contains concepts defined over multiple template arguments (`UnaryFunction`, `UnaryPredicate`, `BinaryFunction` and `BinaryPredicate`). The predicates refine the functions and require the return type be convertible to `bool`.

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 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. Fig. 4 shows the concept kernel for the first template argument of `search`.

5.2 Concept Matching

For each function, TACE type checks 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). TACE checks the consistency of a concept requirement's result type with all conversion requirements in the kernel.

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.

For functions with more than one template argument, TACE generates the concept requirements using a Cartesian join from results of individual kernels.

The final step in reducing the candidate concepts is the elimination of superfluous refinements. A refinement is superfluous if one of its base concepts is also a candidate, and if the refinement cannot provide better typing. In the case of `search`, this results in `ForwardIterator<FwdIter1>`, `ForwardIterator<FwdIter2>`, and the following unmatched requirement:

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

Matching currently does not generate any new conversion requirements. Conversions could be generated for simple cases, but in general several alternative resolutions are possible and the tool has no way of choosing between them. For example, to match `EqualityComparable<T>`, the operands of `operator==` must be converted to a common type, but knowing only that each operand is Regular, we cannot know which to convert.

With *better typing* we mean that a concept's constraints can be used to type-check more requirements. Consider STL's algorithm `find_if`, that iterates over a range of iterators until it finds an element, where a predicate returns `true`.

```
while (first != last && !pred(*first) // ...
```

For `pred(*first)`, both UnaryFunction and UnaryPredicate are candidates. While the latter is able to type-check the `operator!` based on `bool`, the former needs additional requirements for the negation and logical-and operators.

5.3 Algorithm Families

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 requirements of the most general algorithm. Finding the base algorithm 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 such calls occur under scenario §4, TACE requires the user mark the least specific function implementation. A concept repository helps infer the correct refinement relationship.

6 Validation

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.

TACE found a number of functions, where the annotations 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.) Static analysis tools, such as TACE, are limited to recovering syntactic requirements (such as, operations used and typenames referred to), but cannot deduce semantic details from code. For example, a forward iterator differs only semantically from an input iterator (a forward

iterator can be used in multi-pass algorithms). Also, 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 70% of iterator concepts correctly and unambiguously. For about 10% TACE produces a false positive match (e.g., `IntegralType`) alongside the correct iterator concept. For the input set, TACE classifies all functions and predicates (including template arguments marked as `Compare` and `StrictWeakOrding`) correctly, but due to the reason stated at the end of §5.2 does not generate the conversion requirement on result types.

7 Related Work

Sutton and Maletic [28] 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 function templates. In order to avoid complications from "the reality of C++ programming" [28], the authors validate their approach with a number of reimplemented STL algorithms, for which they obtain results that closely match the C++ standard specification. The authors discuss how small changes in the implementation (e.g., the use of variables to store intermediate results provides more type information) can cause small variations in the identified concepts.

Dos Reis and Stroustrup [10] 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 [2]'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 concerned with finding higher level abstractions that describe multiple types (concepts).

Matching the requirements against the definitions in the concept repository is similar to the Hindley-Milner-Damas (HMD) type inference algorithm for functional languages [8]. HMD and its variations derive a type scheme for untyped entities of a function from type annotations and the utilization of these entities

in the context of defined functions. Type classes [29] help overcome problems of the type inference algorithm that stem from operator overloading. Peterson and Jones [19] present an extension to the HMD algorithm, which utilizes type classes to derive an unambiguous type scheme for polymorphic functions.

Our type checking mechanism that tests the requirement sets against concepts in a repository differs from type checking with type classes in several ways:

- C++ template code follows regular C++ programming style, where variables have to be declared before they can be used. The type of a variable can be template argument dependent. C++ concepts allow overloaded function requirements (e.g., random access iterator's subtraction and difference requirement). The types of variable declarations provide the type information that we use for overload resolution.
- C++'s type system allows type coercions. Based on C++ binding rules and conversion (and constructor) requirements that are defined in the concept, TACE generates all possible type combinations that a specific function requirement can handle. For example, if a signature is defined over `const T&` another signature for `T&` is added to the concept.
  Consequently, checking whether a requirement kernel is expressible by a concept in the repository relies on signature unification. The result type of a function is inferred from the requirement specification in the repository.
- in C++, type checking of expressions is context dependent. For example the disambiguation of overloaded functions relies on the expression context. In a call context, this is the argument list of the call. When a function is assigned to a function pointer, the context is provided by the type of the function pointer (i.e., the left hand side of an assignment determines the type that is expected on the right hand side). When code suppresses argument dependent lookup, the rules become more subtle. In this circumstance, the overload set only contains functions that were available at template definition time.
- Haskell type checking utilizes context reduction. For example, an equality implementation on lists may require that the list elements be comparable. Context reduction, extracts the requirements on the element type from the container context. Extending TACE with context reduction would be useful for deriving requirements from templated data-structures (see §3.3).

8 Conclusion and Future Work

TACE is a 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. 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.

Our results show that a static tool, limited to analyzing syntactic requirements, will not always be able to infer concepts correctly. However, for a large number of industrial-strength function templates, TACE can recover and cluster

constraints correctly. The Pivot system provides an extensible compiler based framework that enables enhancements of our analysis.

TACE's current implementation searches the repository for suitable concepts. A formal analysis based approach [28] could be applicable to more library domains. Also, a more precise analysis of class templates (data structures, trait classes, tag hierarchies, etc.) will improve the analysis precision.

9 Acknowledgments

The authors would like to thank Gabriel Dos Reis, Gary Leavens, Jaakko Järvi, Yuriy Solodkyy, and the anonymous referees for their helpful comments and suggestions.

References

1. Abrahams, D., Gurtovoy, A.: C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond (C++ in Depth Series). Addison-Wesley Professional (2004)
2. Agesen, O., Palsberg, J., Schwartzbach, M.: Type inference of SELF. In: ECOOP '93: Proceedings of the 7th European Conference on Object-Oriented Programming. pp. 247–267. Springer, London, UK (1993)
3. Alexandrescu, A.: Modern C++ design: generic programming and design patterns applied. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2001)
4. 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: Proceedings of the 14th International Conference on Languages and Compilers for Parallel Computing. pp. 193–208. vol. 2624 of LNCS, Springer, Cumberland Falls, KY (Aug 2001)
5. 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)
6. Becker, P.: The C++ Standard Library Extensions: A Tutorial and Reference. Addison-Wesley Professional, Boston, MA, USA, 1st edn. (2006)
7. Becker, P.: Working draft, standard for programming language C++. Tech. Rep. N2914 (June 2009)
8. Damas, L., Milner, R.: Principal type-schemes for functional programs. In: POPL '82: Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 207–212. ACM, New York, NY, USA (1982)
9. Dos Reis, G., Stroustrup, B.: A C++ formalism. Tech. Rep. N1885, JTC1/SC22/WG21 C++ Standards Committee (2005)
10. 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. pp. 295–308. ACM Press, New York, NY, USA (2006)
11. 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. MI Lecture Note Series, vol. 22, pp. 151–166. COE, Fukuoka, Japan (December 2009)

12. Dos Reis, G., Stroustrup, B., Meredith, A.: Axioms: Semantics aspects of C++ concepts. Tech. Rep. N2887, JTC1/SC22/WG21 C++ Standards Committee (June 2009)
13. 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. pp. 291–310. ACM Press, New York, NY, USA (2006)
14. Gregor, D., Stroustrup, B., Siek, J., Widman, J.: Proposed wording for concepts (revision 4). Tech. Rep. N2501, JTC1/SC22/WG21 C++ Standards Committee (February 2008)
15. ISO/IEC 14882 International Standard: Programming languages: C++. American National Standards Institute (September 1998)
16. 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. pp. 272–282. ACM, New York, NY, USA (2006)
17. Järvi, J., Marcus, M.A., Smith, J.N.: Library composition and adaptation using C++ concepts. In: GPCE '07: Proceedings of the 6th international conference on Generative programming and component engineering. pp. 73–82. ACM Press, New York, NY, USA (2007)
18. Järvi, J., Willcock, J., Hinnant, H., Lumsdaine, A.: Function overloading based on arbitrary properties of types. C/C++ Users Journal (21(6)), 25–32 (June 2003)
19. Peterson, J., Jones, M.: Implementing type classes. In: PLDI '93: Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation. pp. 227–236. ACM, New York, NY, USA (1993)
20. Siek, J., Lumsdaine, A.: Concept checking: Binding parametric polymorphism in C++. In: First Workshop on C++ Template Programming, Erfurt, Germany (October 10 2000)
21. 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)
22. Stepanov, A., McJones, P.: Elements of Programming. Addison-Wesley Professional (June 2009)
23. Stroustrup, B.: The design and evolution of C++. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA (1994)
24. Stroustrup, B.: The C++ Programming Language. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2000)
25. Stroustrup, B.: Abstraction and the C++ machine model. In: ICESS'04: 1st International Conference on embedded Software and Systems. Lecture Notes in Computer Science, vol. 3605, pp. 1–13. Springer (2004)
26. Stroustrup, B.: The C++0x "remove concept" decision. Dr.Dobb's Journal 92(August) (2009), republished with permission in Overload Journal July 2009
27. Stroustrup, B.: Expounds on concepts and the future of C++. Interview with Danny Kalev (August 2009)
28. 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. pp. 57–66. IEEE (2008)
29. Wadler, P., Blott, S.: How to make ad-hoc polymorphism less ad hoc. In: POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 60–76. ACM, New York, NY, USA (1989)