ConceptClang: An Implementation of C++ Concepts in Clang

Larisse Voufo, Marcin Zalewski, and Andrew Lumsdaine

Open Systems Lab, Indiana University, USA

{lvoufo,zalewski,lums}@osl.iu.edu

Abstract

Concepts are a proposed C++ extension for constraints-based polymorphism. In this paper, we present our experience implementing an infrastructure for exploring concept designs based on Clang—an LLVM frontend for the C family of languages. We discuss how the primary proposed features of concepts (such as concept-based lookup, overloading and constrained templates) are implemented in Clang, and how our implementation can be extended to support the different approaches suggested within the C++ community. Some illustrations are presented and include a subset of the Boost Graph Library.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Compilers

General Terms Languages, Design, Experimentation, Standardization

Keywords Concepts, C++, Clang, Generic Programming

1. Introduction

The Standard Template Library (STL) [2, 24, 28] introduced generic programming to C++. Generic algorithms of STL are expressed using templates, which make the algorithms independent of a particular choice of types. STL uses concepts to group the common requirements that generic algorithms and data structures impose on their (template) parameters. Since the introduction of STL, the generic paradigm became the inspiration to many modern C++ libraries such as, for example, the Boost Graph Library [26], the Computational Geometry Algorithms Library [5], or the Matrix Template Library [27]. As a rule of thumb, C++ code that utilizes templates uses concepts implicitly[1], even if they are not a part of documentation.

Concepts, however, do not have a direct representation in C++, and they cannot be leveraged for type checking of generic code. In addition, properties of concepts such as associated types or concept-based overloading must be encoded using metaprogramming techniques [11, 19]. Because of these limitations, concepts have been one of the most anticipated features to be introduced in the upcoming update to C++ [20]. In the late stage of the standardization process, however, concepts have been removed [16, 21, 31, 33]. An important factor in the decision to remove concepts was the lack of a transparent and evolving implementation of concepts, and, consequently, the lack of experience with concepts-enabled code.

[1] When not used for metaprogramming.

WGP'11, September 18, 2011, Tokyo, Japan.
Copyright © 2011 ACM 978-1-4503-0861-8/11/09. . . $10.00

In this paper, we discuss our experience with implementing concepts in the Clang compiler [6], a modern LLVM [23] frontend for the C family of languages. Thanks to the licensing scheme and the modern coding style of Clang, our implementation is transparent, manageable, and accessible to the C++ community at large—that is in contrast to other open-source C++ compilers, which are much harder to access for outsiders than Clang due to lower level of abstraction and modularity.

We first introduce the background and related work in Sect. 2. Then, in Sect. 3, we briefly recap generic programming with concepts and its C++ incarnation. In Sects. 4 and 5 we discuss the implementation issues one has to resolve when extending an existing C++ compiler with concepts and how we have handled these issues in our implementation. Finally, in Sect. 6, we present a case study on a small library based on concepts that we used to test our implementation.

Our implementation, ConceptClang, and case study library are available for download [34].

2. Related Work

Concepts, as known in C++, originate from the algebraic specification language Tecton [22]. In Tecton, concepts are used to describe algebraic structures by specifying the common signatures and properties that these structures must share, making specifications independent of unimportant details. The work on Tecton resulted in efforts to afford the same kind of genericity to software, first in Scheme, then in Ada, and ultimately in C++ [30].

In C++, genericity is achieved through the use of templates. Already in 1994, Stroustrup [29], in his "Design and Evolution of C++" book, discussed the need for checking of template arguments in C++ and outlined possible ways of doing so. The ideas were not mature at that time, and templates are only minimally checked in C++. As generic programming has gained ground, the Boost Concept Checking library [25] has been proposed as a provisional solution that improves error checking using standard C++ features. This kind of provisional checking is used by many Boost libraries [8], and has even been incorporated into the STL provided by GCC.

After over a decade of experience with generic programming, concepts have been proposed as a first-class feature for the C++ language [9, 18][2]. The proposals went through many refinements and adjustments in the C++ standardization committee, and in 2007 concepts were voted in as a feature of C++0x [13]. Gregor provided an experimental compiler, ConceptGCC [14], which was able to handle a subset of proposed syntax and semantics. In the final stages of the formalization process, however, doubts have arisen about the impact of concepts on an average C++ programmer, and there

[2] See `http://www.generic-programming.org/` for a comprehensive list of publications.

was not enough experience with using concepts to resolve these doubts. Consequently, the committee has reluctantly decided to remove concepts from the upcoming standard [16, 21, 31, 33], with an intention of revisiting concepts for the next round of standardization.

Our work picks up where the discussion on concepts was left off. Our goal is to provide an extensible and easily accessible implementation of concepts, with an initial design based on the latest C++ standard draft that included concepts [3]. Our implementation does not endorse any particular design choices made in the standard draft; instead, we strive to provide an infrastructure that can serve as a base for experimentation with different designs. The common infrastructure of an implementation for concepts come from the basic elements of generic programming. In our discussion in Sect. 4, we follow the previous work that identified these basic elements in a cross-languages comparisons of generic programming features [4, 11].

Our implementation, ConceptClang, is based on Clang [6], the frontend for the C family of languages in the LLVM compiler infrastructure [23]. We have chosen Clang because it is the only compiler that satisfies all of our requirements. It is released under a license that allows extension and experimentation. Clang is developed using modern C++ technology, making the code highly structured and, given the scale of the project, easily understandable. Clang is modular, and one of the main principles driving its development is a library-based approach that provides clear interfaces between the parts of the compiler. Last but not least, Clang is already a production grade compiler that supports all of C++ and the support for future C++0x is being actively developed[3], which will ensure that our implementation will be a feasible base for serious experiments.

Our implementation is not the first attempt at implementing concepts. ConceptGCC [14] is an experimental compiler based on GCC [12]. ConceptGCC has implemented a large part of the concepts proposal, and it incorporated a largely "conceptualized" version of STL. ConceptGCC, however, does not fulfill the goals we set for our implementation. It is based on GCC, which has much steeper learning curve than Clang, is much harder to modify, and the resulting infrastructure is not transparent. Furthermore, the development of ConceptGCC has been effectively halted in 2008, making it fall behind concept proposals and hard to integrate with newer GCC versions. The STL implementation bundled with ConceptGCC, however, will serve as a base of our future STL implementation for ConceptClang. Concepts have also been partially implemented as "syntactic sugar" [7] using C++ Transformers [1]. Such implementation is suitable for transitional period, but it cannot completely cover all concept-related features. Further problems with such implementation are that it is syntax-based, forgoing most of the semantic information, that it is an additional tool in the tool chain, and that it does not expose parts of concepts infrastructure in a transparent, modular, and extensible manner.

In the next section, we briefly recapitulate C++ concepts as defined in the latest standard draft, and discuss the basic elements of generic programming.

3. Concepts

While concept proposals for C++ differ in details, they share a common set of underlying ideas that are constant between the proposals, especially from the perspective of a compiler implementer. Essentially, one can abstract the elements of generic programming with concepts [4, 11] into five main components:

- concepts,
- refinement,
- modeling,
- constraints, and
- generic algorithms.

```
1  concept Hashset<typename X> : HasEmpty<X> {
2    typename element;
3    requires (Hashable<element>);
4    int size(X);
5  }
6  // modeling
7  template<typename K>
8  requires (Hashable<K>)
9  concept_map Hashset<intmap<list<K>>> {
10   typedef K element;
11   int size(intmap<list<K>> m) { ... }
12 }
13 // algorithm
14 template<typename T>
15 requires (Hashset<T>)
16 bool almostFull(T h) { ... }
17
18 // instantiation
19 intmap<list<int>> h;
20 bool test = almostFull(h);
```

Figure 1. Generic programming with concepts in C++.

The code in Fig. 1 illustrates these elements. First, a *Hashset concept* is defined. The concept *refines* another concept, *HasEmpty*, which is not shown in the code listing. The *Hashset* concept has one parameter, *X*, that is part of the concept's signature and that is later used to identify modelings. The concept defines *associated entities*: an associated type *element*, an associated requirement (Line 3, the definition of the *Hashable* concept not shown), and an associated function *size*. The complete meaning of the *Hashset* concept is defined by its associated entities and the associated entities of the *HasEmpty* and *Hashable* concepts.

Given a concept, one has to establish the relationship between the concept and the types that *model* the concept by providing *modelings*. In Fig. 1, modeling is established by a *concept_map* template. The template states that for any type *K* that is *Hashable* the *intmap<list<K>>* type models the *Hashset* concept. The compiler verifies this relationship by checking that all associated entities of the *Hashset* concept have been provided in the concept map.

Generic algorithms are parametrized by types, and the types are *constrained* by concepts (Line 15). In Fig. 1, the generic algorithm *almostFull* requires that its argument meet the *Hashset* concept. The constraint serves as an interface between the algorithm and the environment: whenever the type parameter *T* is used, the compiler must ensure that it is only used in one of the ways that the *Hashset* concept allows. For example, *almostFull* can use the *size* function or the associated *element* type.

When a generic algorithm is used, the compiler must check that it is used properly. The check consists of finding the modelings required by the constraints of the generic algorithm and substituting the implementations of the associated entities of concepts with appropriate implementations from the modelings.

In this section, we introduced the terminology, but we did not explore all complexities of generic programming with concepts. In the next section, we explore the details of generic programming in C++, and we discuss the issues that a compiler implementer must consider when adding support for C++ concepts. Our discussion is similar and at times intersects with the implementation description of ConceptGCC [17].

[3] See `http://clang.llvm.org/cxx_status.html`

```
int x;
std::accumulate(x, x, x);
```

```
/usr/include/c++/4.2.1/bits/stl_numeric.h:88:20: error:
      indirection requires pointer operand ('int' invalid)
      __init = __init + *__first;
                        ^~~~~~~~
test.cpp:5:3: note: in instantiation of function template
      specialization 'std::accumulate<int, int>' requested here
  std::accumulate(x, x, x);
  ^
```

Figure 2. Example of template instantiation error produced by Clang.

4. Implementation Issues

Concepts are an extension to a well established language for which there are many high quality compilers. Thus, a compiler implementer must consider how to fit concepts infrastructure into the existing compiler bits. In addition, concepts are still under discussion, and there are some variations in the different designs. The issues we describe in this section guide our implementation, but they are also the issues that other compiler implementers may encounter.

4.1 Concepts in C++

In Sect. 3, we have given a description of generic programming using general terms. In C++, generic programming with concepts comes down to templates, as can be seen in Fig. 1. Function and class templates abstract from types, using template parameters instead. But when a template is instantiated, the way that the template parameter is used must match the implementation that a type provides. If the use does not match the implementation, errors result at instantiation time. Fig. 2 shows an error generated by Clang on an incorrect use of the STL *accumulate* algorithm. The error is simple when it comes to template instantiation errors as it does not involve nested instantiation, but, even so, it points to the internals of the library. The problem is that the misuse of templates cannot be caught at the interface level. In practice, such errors get overly complex and large.

Concepts, as a language feature, introduce an interface between the template code and the types that the template code can be instantiated with. Concepts provide means to establish an environment for type checking of templates that includes functions, types, and function templates. This environment can be built up from smaller pieces, using mechanisms such as refinement and associated requirements. Once concepts are established, they can be used to constrain templates. A constrained template is fully checked using standard rules of C++ type checking, against the environment provided by template constraints. When modelings are provided, particular types are checked against concepts, either implicitly or explicitly through concept maps (explicit maps can be synthesized by the compiler on demand for "auto"—or implicit—concepts). Once a constrained template has been type checked, it can be safely, up to some exceptions discussed later, instantiated with types that provide all the necessary implementation, or a clear error can be given about the lack of necessary modelings. Such errors are at the interface level, not referring to the depths of the library code as non-constrained template instantiation errors do.

4.2 Concept Definition

Concepts are similar to the existing constructs of class templates and namespaces. Concepts, as class templates, have parameters but, as namespaces, have no state nor run-time representation. Concepts can be refined and can have nested requirements, which is similar to class inheritance but not quite the same. Finally, concepts have members that are unlike any other construct of C++, whether these members are pseudo signatures or valid expressions of Dos Reis and Stroustrup [9].

A compiler implementer must consider how concepts should be represented. One possibility is to adapt the implementation of class templates, but due to the differences between classes and concepts, this implementation path may be difficult. If the compiler is modular enough to pick and choose parts of the implementation, concepts have the following properties: a scope, parameters, refinements, associated requirements, and members.

Thus, a concept should inherit properties of a parametrized scope, with the addition of new mechanisms of refinements, associated requirements, and special members, be it pseudo signatures and associated types, or valid expressions.

Implementation for concepts as parametrized scopes with refinements and associated requirements is pretty independent of particular design choices for concept members. Choosing between pseudo signatures and valid expressions, however, impacts the implementation in a much stronger way. Pseudo signatures are similar to the existing C++ declarations, but they have slightly different syntax in many cases. For example, expressing requirements for class members, including constructors and destructors, must refer to parameters in contrast to the declarations in classes where members implicitly refer to the class being defined[4]:

```
concept Container<typename X> {
  X::X(int n);
  X::~X();
  bool X::empty() const;
}
```

Many other adaptations of syntax are necessary for concepts, but pseudo signatures are similar to the usual C++ signatures. Thus, the parsing code of the compiler, for some declarations and definitions (for default implementation), can be reused. The situation is similar for valid expressions, but they must be parsed like expressions, not declarations.

4.3 Modeling

Modelings have two aspects. First, they define and establish a mapping between one or more types and a concept. Second, they provide implementations stating the ways in which the types match the concept. This said, they must be identified by naming the arguments of the concept. In that sense, each modeling has a unique name, just like template specializations in C++. The unique names are called *concept ids* and are formed with concept names followed by template arguments.

Modelings can be templates, as illustrated by the concept map for the *Hashset* concept in Fig. 1. Gregor and Siek [17] explicitly relate concepts and their models to templates. To that effect, a concept is like the primary template and models of the concept are either partial or explicit specializations of the primary concept template. This analogy works very well, except for the difference in syntax, by which a concept map without template parameters does not necessitate the *template<>* construct of an explicit specialization. As a result, implementations of template specialization should be reused, if possible, for identifiers of concept maps.

The content of a concept map, however, must follow strict rules unlike class template specializations. The compiler has to ensure that only the members of a concept have a satisfying implementation in a concept map, and all such members do. There are two kinds of associated entities that concept maps have to provide:

- values for associated types and class templates, and
- implementations for associated functions and function templates.

[4] An example from [3]. Other examples will not be explicitly attributed.

The values of associated types are declared using the familiar *typedef* syntax, and the goal of the compiler implementer should be to reuse the existing type alias infrastructure. Similar recommendation applies to associated class templates, which are defined using the upcoming C++0x template alias syntax [10]. Associated functions and function templates are looked up in models using closest match rules, putting the "pseudo" in pseudo signatures. So, for example, one function in a model may satisfy more than one pseudo signatures:

```
concept A<typename T> {
  void foo(T);
  void foo(const T);
}

concept_map A<int> {
  void foo(int i) {}  // satisfies both pseudo
                         signatures of A
}
```

Also, as a further complication, implementations may be looked up by the compiler in the scope of a concept map, and saved as a "candidate set" resulting from overload resolution based on a pseudo signature [15]. In general, satisfaction of associated functions should be integrated with the overload resolution implementation of the compiler, where modeling implementations are an overload set, looked up in the scope of the concept map or the surrounding scope.

Finally, the issue that has stirred up much controversy in the C++ standardization committee is that modelings can be stated explicitly, using concept maps, or implicitly where the compiler automatically deduces associated entities. From the implementation perspective, explicit concept maps require extra effort of implementing the additional syntax and providing the code necessary for associating requirements with explicitly provided implementations. Given an implementation of implicit but optionally explicit (or the other way around) modelings, switching exclusively to one option should be a minimal implementation effort.

4.4 Constrained Templates

Handling of constrained templates consists of two quite distinct tasks: type checking of templates and template instantiation.

4.4.1 Separate Type Checking

Constrained templates and their modelings are type checked *separately*, against a common interface defined by concepts. Once separate type checking is complete, with a successful *requirements satisfaction* check—a.k.a. *constraints* check, the use of a template is safe. In the previous section, we described type checking of modelings, and in this section we describe type checking of constrained templates and requirements satisfaction checking.

Type Checking Against Constraints. Constrained templates differ strongly from unconstrained templates in that there are no dependent names. An unconstrained template is checked only up to a point where it does not depend on template parameters and any expression or declaration involving template parameters is only checked for basic syntactic correctness. Constrained templates are checked as usual C++ code, against the environment built up by the constraints.

A constrained template may have more than one constraint, each referring to at least one template parameter and possibly other non-parameter types. The task of the compiler is to unify all these constraints into a coherent environment. There are two main ways of achieving this. The specification of concepts [3] is written based on the implementation experience in ConceptGCC, and it suggests that constraints be used to construct *archetypes* for all template parameters and concept maps. Archetypes are created by traversing constraints and adding associated types and member functions (including constructors, destructors, etc.) to an internal set of hidden types maintained by the compiler. Concept map archetypes are hidden concept maps created by adding archetypes for associated types and associated non-member functions, which are also injected into the scope of the constrained template for unqualified lookup. For functions, only signatures need to be created in archetypes. Then, when archetypes are extracted from constraints, the usual C++ type checker can be applied.

This approach is simple, and allows to easily deal with duplicate associated functions that might arise. In another approach, building an actual environment of archetypes and functions may be avoided, and lookup may be performed directly in the constraints. Such approach may save the effort of constructing full environments when only a small part of constraints is used, but it may be more expensive to traverse constraints repeatedly in larger templates. The results of type checking must be stored for instantiation, so that the compiler can replace the members looked up in the constraints with their implementations in the modelings.

Constrained templates can refer to other constrained templates. The compiler has to be able to use the environment provided by constraints to check the reference to another constrained template. Checking such nested uses of constrained templates boils down to a requirements satisfaction check ensuring that all modelings required by a nested template are implied by the surrounding template.

Requirements Satisfaction. Requirements satisfaction check occurs after template arguments deduction, the process in which the compiler deduces the actual template arguments of a template from the template use. When the type arguments are known, the compiler has to ensure that all modelings required by the template are found in appropriate scopes. If some required modelings are not found, the programmer is presented with a clear error message enumerating the missing modelings.

Modelings are found using the same rules as the rules for choosing template specialization. A non-template model is always better than a template model, and template models are ordered just as template specializations are. If two or more models are equally good, ambiguity arises just like for template specializations. Implicit concepts are generated on demand, when no explicit modelings exist. Model lookup may require backing out of modelings in cases of failed concept map templates, failed default implementations of concept members, and failed concept maps generated for implicit concepts. Backing out of tentative entities has traditionally been a hard task for C++ compilers, so special care has to be taken in the implementation to enable the correct behavior.

4.4.2 Instantiation of Constrained Templates

The instantiation of constrained templates is quite different from that of unconstrained templates, especially when it comes to function calls within templates. In a constrained template, type checking binds all names to the constraints of the template. At instantiation time, as the modelings for the constraints are available, these name bindings need to be substituted with the corresponding bindings from the modelings. Calls to associated functions of concepts, in particular, go through overload resolution over their corresponding candidates set—an operation which may result in undesired ambiguity. This possibility of late failure may cause "old style" errors from within template instantiations to occur, but at least these errors are more localized in this case.

Calls to constrained templates get checked twice, at type checking and at instantiation time. Each time, the call goes through overload resolution, and the second time is another opportunity for late failure.

Calls to unconstrained templates are no longer possible, as they are not safe. This may disable optimizations available from legacy libraries. However, these libraries may be adapted to concept-based settings.

4.5 Refinement and Associated Requirements

Refinement allows building concepts from other concepts. Refinement implies the inclusion of all associated members of a refined concept into the refining concept. Furthermore, refinement is also a convenience feature for defining modelings, where a modeling for a refining concept implies modelings for refined concepts. As explained in the previous section, refinement enables overloading on concepts since satisfaction of one constraint may imply satisfaction of other constraints along the refinement chain.

Associated requirements differ from refinements in three ways. First, nested requirements can be used with associated types of concepts, while refinement may only be used with concept parameters. Second, associated entities of refined concepts are included into the refining concept while the associated entities of associated requirements are not. Finally, when a concept map for a refining concept is defined, the concept maps for refined concepts are generated using the implementation given in the map and using the usual implicit generation rules, if no explicit implementation is given. Associated requirements, on the other hand, have to be satisfied before a concept map for the concept that includes them is defined.

Implementation of refinement requires name lookup in refined concepts, implication of refined concepts in requirement clauses, generation of concept maps for refined concepts when necessary, and compatibility checking between concept maps in a refinement hierarchy. Associated requirements require a much simpler implementation that introduces assumptions about existing modelings in concepts that include associated requirements, and that introduces additional constraints corresponding to associated requirements on concept maps and constrained templates. While both mechanisms serve as concept building devices, their implementation requirements are quite different.

4.6 Concept-Based Overloading

The introduction of concepts has been tied with the a characteristic feature of C++ called *overloading*. A template class or function can have multiple implementations based on differing type patterns, and the best candidate must be chosen at the time of use of the template. Choosing the best candidate is based on comparing two different patterns to determine which one is more specialized than the other, using the language's rules of partial ordering for templates.

With concepts in the picture, the constraints of the template must also be taken into account during the comparison. A rule of thumb is that if a template can be instantiated given the constraints of another, then it is more specialized. And if the converse is also true, then neither is more specialized, resulting in an ambiguity.

According to the concepts wording, the result of comparing the type patterns (without the constraints) takes priority over the result of comparing the constraints, as illustrated in Fig. 3. While this behavior may be favorable in conserving pre-existing optimizations in a given library, it may hinder the safety that one hoped to achieve by adding constrained template implementations in the library. One must thus make a decision on the appropriate partial ordering mechanism to base template specialization on.

Making the choice in the appropriate ordering mechanism is only one of the issues that an implementer should consider. One should also ensure that the compiler does not reject the overload of function templates that are only different by their list of constraints. Finally, when unable to choose a viable candidate, the compiler should return an informative diagnostic of all the possible candidates as well as why they were not chosen.

```
//#1
template<typename T>
requires C<T>
foo (T& a) { ... }

//#2
template<typename T>
foo (const T& a) { ... }

// concept_map C<const int> has been defined.

//Instantiation chooses #2
const int i = 1;
foo(i);
```

Figure 3. Unexpected case of overloading.

5. ConceptClang Infrastructure

Given the issues discussed in the previous section (Sect. 4), we are now ready to get into the implementation details of our ConceptClang infrastructure for concepts. While ConceptClang is still work in progress, the logic that it is based on will provide important insights to other implementers, in both understanding Clang and implementing parts of concepts infrastructure. Noticeably, compared to previous implementations, we treat concepts as first-class entities in the compiler implementation by keeping the implementation of concepts and concept maps separate—though analogous at times—from that of class templates and specializations.

We start our discussion with the definitions of concepts and their models, then continue with template definitions and their uses, and we end with concept-based overloading.

In the remainder of this section, we frequently refer to the implementation of Clang. For brevity, where Clang names are self-explanatory, which is usually the case, we use them as nouns. For example, *TemplateDecl* is the name of a class representing template declarations, and *RestrictedScope* is the name of a flag that indicates a restricted scope.

5.1 Concepts and Concept Maps

The representation of concepts and concept maps strongly influences the implementation and reusability of concept checking. In this section, we introduce our representations of concepts and concept maps. In Sect. 5.1.3, we discuss concept map *archetypes*, which are an internal compiler notion of temporary concept maps used for type checking. Finally, in Sect. 5.1.5, we briefly discuss how we handle the difference between implicit and explicit concepts, and how the infrastructure can be adapted for alternative proposals. Subsequent sections after this will highlight the reusability of our concept checking.

5.1.1 Concept Declaration

We represent concept declarations as template declarations; our *ConceptDecl* publicly inherits from Clang's existing *TemplateDecl*. A *TemplateDecl* object holds a list of type, template, and non-type template parameters, and concept declarations use that list to represent concept parameters. A concept declaration is also a declaration context, inheriting from Clang's *DeclContext*, that contains other declarations:

- associated type and class template declarations represented as *TemplateTypeParmDecl* and *TemplateTemplateParmDecl*, respectively,
- default values for associated types represented as type alias declarations, using Clang's *TypedefDecl*,
- associated functions and function templates using *FunctionDecl* and *FunctionTemplateDecl* that can also store a function body for a default implementation, and

```
void func();                         // #1
void foo(int a, int b);              // #2
template<typename T> void foo(T t);  // #3
concept A<typename T> {
  void bar();                        // #4
  void foo(int a, int b)             // #5
  {
    func();                          // Picks #1
    ::foo(0,1);                      // Picks #2
    foo(0,1);                        // Picks #5
    bar();                           // Picks #4
    foo(t);                          // Picks #3
  }
}
```

Figure 4. Parsing associated functions.

- associated requirements and concept refinements, also represented as declarations that we will discuss in Sect. 5.1.3.

For all these declarations, we reuse Clang's parsing procedures and structures to a major extent. Notably, for concept parameters and associated types, the use of Clang's infrastructure for multi-level parameter lists allows us to better reuse its template argument substitution mechanism. In addition, our reuse of Clang's scope mechanism helps us to avoid large memory footprints.

Concept Parameters and Associated Types. For parsing and storing concept parameters, we reuse Clang's template parameters parsing and representation. Clang stores template parameters in multi-level lists, where each level corresponds to a nested template, and each template parameter is uniquely identified. Concept parameters are stored in the first level, and we represent associated types as a second level of template parameters. The representation of associated types as nested template parameters allows direct reuse of Clang's type checking discussed in Sect. 5.2.

Parsing Scope When a concept declaration is parsed, we enter a *Scope*. *Scope* is a transient data structure used by Clang when parsing the program, and it assists in resolving identifiers to appropriate declarations. When a *Scope* is created, it uses flags to indicate the kind of scope that is being parsed; some examples of possible flags are *FnScope*, *ClassScope*, or *DeclScope*. The flags influence how name lookup is performed. For a concept scope, we set three flags: *DeclScope* and two flags we added to Clang, *ConceptDefnScope* and *RestrictedScope*.

DeclScope indicates that a concept is a declaration which contains other declarations that should be used in name lookup. *RestrictedScope* indicates that name lookup should be performed in a constraints environment. In the case of a concept, the constraints environment consists of concept refinements and associated requirements, which are added to the scope as they are encountered. Fig. 4 illustrates the results of lookup from a default definition of an associated function. The default definition has its own *Scope*, but scopes are chained, and the lookup proceeds in the parent scope of the concept, and then in the scope surrounding the concept. *ConceptDefnScope* indicates that default definitions of functions should allow unconstrained templates, as shown in Fig. 4. This is an experimental feature of ConceptClang that does not conform with concepts specification [3], but thanks to our implementation with *Scope* flags, it is easy to change.

To handle restricted scopes, ConceptGCC injects the associated entities from refined concepts into the refining concept, as a concept post-processing step. This results in larger memory footprints of the compiler due to repetitions of identical entities in memory [17]. ConceptClang does not explicitly inject names into restricted scopes. Instead, it uses the *RestrictedScope* flag, avoiding repetitions.

```
...
concept PA<typename T>  {
  void bar();               //#4
}
concept A<typename T> : PA<T> {
  void foo(int a, int b)   //#5
  {
    ...
    bar();                 // Picks #4
  }
}
```

Figure 5. Parsing associated functions of concept definitions, with refined concepts.

Name-Lookup in Restricted Scopes. Name lookup is driven by the *CppLookupName* and the *LookupName* procedures. When a restricted scope is encountered, *LookupQualifiedName* is used to explicitly look names up in the declaration contexts that have been added to the scope (e.g. refinements and associated requirements), recursively traversing the concept refinement hierarchy. We illustrate the resulting behavior in Fig. 5, which is adjusted from Fig. 4. We discuss the details of restricted-scope name lookup in Sect. 5.2.1.

Associated Requirements vs. Concept Refinements. One of the differences between refinements and associated requirements given in Sect. 4.5 is that associated requirements are not used in name lookup in concepts. Currently, in ConceptClang, both refinements and nested requirements participate in name lookup, as a basis for experiments (although the behavior is easily modifiable.) In the remainder of this paper, we only refer to associated requirements when we treat them differently than refinements. Otherwise, discussion of refinements implies a corresponding treatment of associated requirements.

Storage of Concept Maps. Each concept declaration holds a set of concept maps and another set of concept map templates. The separation is necessary as each set requires a different kind of processing. These sets are the only way that one can access maps of the given concept.

For fast lookup and comparison, we reusing Clang's mechanism for unique identification. The concept maps stored in each concept are uniquely identified as *FoldingSetNode*s, stored in a an LLVM's *FoldingSet*. This storage mechanism allows to efficiently check whether a given concept map exists.

5.1.2 Concept Map Declaration

Like concept declarations, a concept map declaration, *ConceptMapDecl*, is a template declaration and a declaration context. However, the template parameter list serves a different purpose in a concept map than in a concept. When the list is non-null, a *ConceptMapDecl* represents a concept map template declaration. Otherwise, a *ConceptMapDecl* represents a concrete concept map declaration, with concrete type arguments. In addition, the list has only one level.

A *ConceptMapDecl* holds a pointer to the concept that it maps, as well as implementations of declarations specified in its mapped concept. The implementations are represented as declarations in the following ways:

- associated types definitions as *TypedefDecl*s,
- associated function and function template definitions as *FunctionDecl*s and *FunctionTemplateDecl*s respectively, and
- concept maps for associated requirements and refined concepts as lists of *ConceptMapDecl*s.

In parallel with the two-level structure of the template parameters lists in concept declarations, concept map declarations maintain a two-level template arguments list. The first level holds the template arguments of the concept map, and the second level holds the type values of each associated type. The first level of arguments is constructed before parsing the concept map body, while the second level is constructed after parsing the map declaration, during type checking and after associated type definitions are processed.

As with concept declarations, there is a *Scope* associated with each concept map declaration. Analogously to the *ConceptDefnScope* flag, ConceptClang adds the *ConceptMapScope* flag to indicate that one is parsing a concept map declaration. Thus, the following flags are set on the scope of a concept map: *ConceptMapScope*, *DeclScope*, and *RestrictedScope*.

The meaning of scope flags and the behavior of the name-lookup process are similar to that described for concept declarations. One exception is that what is looked up in the associated requirements and concept refinements in a concept, is looked up in the corresponding implementations in concept maps. To enable the name lookup of associated declarations in the mapped concept while parsing a concept map, ConceptClang adds a pointer to the mapped concept in the scope immediately after the scope is entered.

Unique Identification. A *ConceptMapDecl* is also a *FoldingSetNode*. In Clang, this requires that the implementation of the *ConceptMapDecl* structure be extended with a an implementation of *profile* methods to specify ways in which concept maps are different from one another. The profile methods compute a unique identifier based on the following properties of a concept map: the mapped concepts, arguments lists, and declaration contexts.

Type-checking of a Concept Map. Once a concept map declaration is parsed, it is then checked that implementations correspond to associated declarations, as discussed in Sect. 4.3. We refer to this procedure as *concept map generation*, which currently checks the correspondence more zealously than required. ConceptClang verifies that associated types have only one definition, but the rules we discussed in Sect. 4.3 require that conflicting definitions be allowed if they are in different concept maps for the same concept (due to refinement) and if they specify the same type. The more permissive check makes it easier to write concept maps for refined concepts.

In addition, ConceptClang requires an exact match between the signatures of associated functions and the signatures of implementations in a concept map, while concepts specification allows a less exact match that generates "candidate sets" containing functions that differ only in `constant` or `volatile` (*cv*) qualifiers. However, implementations for associated functions are already stored as overload sets rather than a single function, and the modification to match the behavior described in Sect. 4.3 will not require drastic changes to the current infrastructure.

The Concept Map Generation Procedure. The lookup of implementations for associated entities of concepts is roughly implemented according to the following procedure. The procedure takes a concept map as input and modifies the map as an effect:

1. For each associated declaration (signatures for functions and function templates) in the mapped concept:
   (a) Copy the associated declaration, substituting template arguments for template parameters.
   (b) Look for an implementation in the map that matches the substituted copy. If not found,
      i. look for a default implementation in the mapped concept. If that fails as well, then
      ii. look in the immediate surrounding scope.
   (c) Reject the declaration if the implementation is not found.
   (d) Otherwise, modify the concept map appropriately.
      i. Add the substituted copy of the associated declaration to the concept map, if it was not already there.
      ii. For functions and function templates, store a pointer to the actual implementation—the body of the declaration of the associated function or function template.
2. Generate or find concept maps for refined concepts, after substituting the template arguments into the refinements.
   (a) Search for required concept maps in the lists of concept maps stored in the *ConceptDecl*s for refined concepts.
   (b) If some refinements remain unsatisfied, call this procedure recursively with the declarations from the current concept map. If any declarations remain unused in the current concept or its refined concepts, return in error.

```
concept PA<typename T>  {
  T bar();
  T foo();
}
concept A<typename T> : PA<T> {
  T foo() ;
}
concept_map A<int> {
  int bar() {...}
  int foo() {...} // for A<int> and PA<int> maps
}
/*  The above map internally generates this:
concept_map PA<int> {
  int bar() {...}
  int foo() {...}
}
concept_map A<int> {
  int foo() {...}
}
*/
```

Figure 6. Concept map generation.

Fig. 6 shows an example of the interaction between concept maps and refinement. For the guaranteed success of the necessary type substitutions, Step 2 of the concept map generation procedure is invoked earlier in the process: before processing associated types during Step 1 for refinements and after processing associated types for associated requirements.

5.1.3 Concept Map Archetype

ConceptClang generates internal representations of concept maps, called concept map archetypes (CMA), similarly to archetypes described in Sect. 4.4.1. While we generate CMAs, ConceptClang implements more concepts functionality as a part of lookup procedures than ConceptGCC or than the concept wording suggests. Two important differences are that we do not generate type archetypes and that we traverse concept map archetypes in name lookup instead of directly injecting their members into scopes.

Because template parameters can be directly treated as types in Clang, we do not implement type archetypes explicitly. In addition, there are a number of type support structures for storing information about types such as *cv* qualifiers, using *QualType*, or type locations, using *TypeSourceInfo*.

A CMA conveys minimal information about a concept map. CMAs differ from concept maps in the following way:

- Their type arguments do not have to be concrete; for example, they may be type-representations of type parameters.
- They do not hold the actual values for associated types and instead directly use type parameters (*TemplateTypeParmDecl*) representing associated types in concepts (see Sect. 5.1.1).

- For associated functions and function templates, they hold only substituted copies of signatures from concept declarations and no function bodies.
- For associated requirements and refined concept maps, they store the corresponding CMAs.
- Most notably, they are not added to the listing of maps held by the mapped concept.
- They do not have a *Scope* associated with them since they are automatically generated by the compiler and not parsed.

The Use of CMAs. In general, ConceptClang uses CMAs to facilitate name lookup where a concrete concept map is not available. In concept declarations, CMAs are used to represent associated requirements and concept refinements. In constrained templates, including concept map templates, each constraint is represented as a CMA. For concept map templates in particular, CMAs are used as placeholders for associated requirements and refinements that depend on template parameters. Likewise, when type-checking nested calls to constrained templates from a constrained template, they are used as concept maps. In Sect. 5.2.1, we discuss in more detail how CMAs facilitate the type checking and instantiation of templates.

5.1.4 Constrained Concept Map Templates

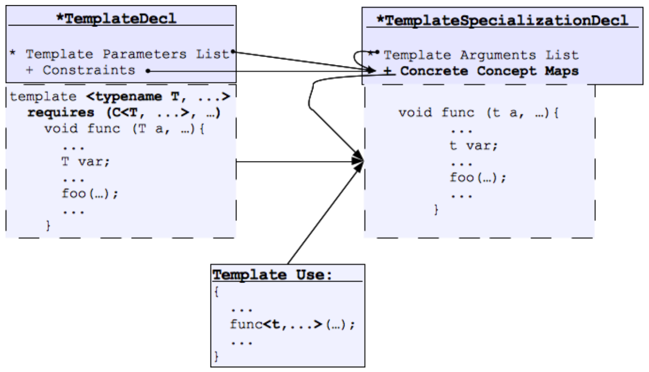
Concept map templates are always constrained, and the body of a concept map template is checked against its constraints, which can be empty. During parsing, all members from the constraints are available for lookup, but this is an automatic result of the *RestrictedScope* flag set for parsing the declaration. During type-checking, Concept map templates follow the map generation procedure described in Sect. 5.1.2, with a modification in Step 2, in which concept maps for refined concepts are generated. That is, as a preliminary step of Step 2 of the generation procedure, concept maps for refinements are looked up in the constraints of the concept map templates.

For each associated requirement or refinement *R*, where *R* is the requirement or refinement with the arguments of the concept map template substituted for its parameters, the constraints are searched one by one for matching concept maps. First, each constraint is compared to *R* using *structural equality* in Clang's lingo. Structural equality only compares the name of the concept and the template arguments, ignoring other properties such as *DeclContext* since *R* and constraints are declared in different contexts. If a constraint does not match *R*, we recursively test for structural equality throughout the refinements and requirements of the constraint, using the *isAscendantOf* procedure of ConceptClang. If no matching map ends up being found, Step 2 proceeds as normally. Otherwise, the constraint is used as placeholder for the matching map.

5.1.5 Implicit Concepts and Alternatives

Implicit (or "auto") concepts and explicit concepts share the same implementation in ConceptClang, with an indicator flag set for implicit concepts. According to the concepts wording, when a concept map for an implicit concept is needed, the compiler should attempt to implicitly generate the map if it was not defined explicitly. The implementation we provide can be used with both kinds of concepts, and can be easily adjusted for various extensions such as the use of associated axioms, or alternative proposals such as explicit refinement [32] and valid-expressions [9].

For example, for the three examples herein, one would have to modify the parsing of concept definitions and the concept map generation procedure, to account for syntactic and semantic differences. In addition, for explicit refinement, one can set explicit concept flags only for explicit refinements. For axioms, one can add related declarations in the *ConceptDecl* structure. And, for

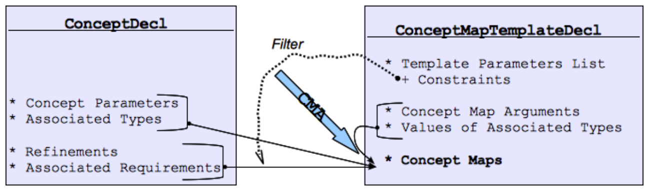

*TemplateDecl
* Template Parameters List
+ Constraints
*TemplateSpecializationDecl
* Template Arguments List
+ Concrete Concept Maps
Template Use:

```
template <typename T, ...>
  requires (C<T, ...>, …)
    void func (T a, …){
      ...
      T var;
      ...
      foo(…);
      ...
  }
```

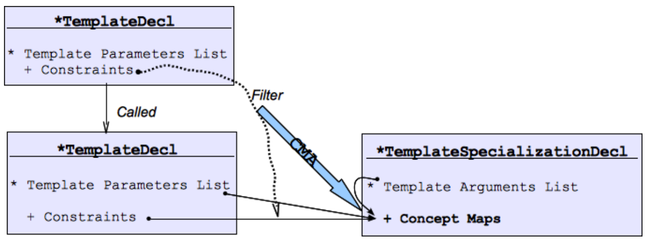
```
    void func (t a, …){
      ...
      t var;
      ...
      foo(…);
      ...
    }
```

```
{
  ...
  func<t,...>(…);
  ...
}
```

Figure 7. Templates example: definition, use and specialization.



(a) For concept map template generation


*TemplateDecl
* Template Parameters List
+ Constraints
Called
Filter
CMA
*TemplateDecl
* Template Parameters List
+ Constraints
*TemplateSpecializationDecl
* Template Arguments List
+ Concept Maps

(b) For type checking of constrained template reference

Figure 8. Re-uses of the constraints check procedure.

valid-expressions, one can replace *ConceptDecl*'s *FunctionDecl*s and *FunctionTemplateDecl*s with *Expr* objects.

5.2 Constrained Templates

In Sect. 4.4.1, we introduce separate type checking of constrained templates. When constraints are specified on a function, a class, or a concept map template definition (with the *requires* keyword), the constraints form its *constraints environment*, and its body must be checked against the environment.

The next part of the type checking occurs when a template is used with some arguments. When this happens, the compiler substitutes the arguments for template parameters and checks that the necessary concept maps exist (or can be generated) for each constraint of the template.

After the compiler ensures that all necessary concept maps exist or can be generated, the template can be instantiated. Fig. 7 illustrates the overall behavior: When a constrained *TemplateDecl* is used, a specialization is generated that can be instantiated at some later point. The requirement satisfaction procedure finds concrete concept maps for each constraint of the template. The maps later aid in the instantiation of the template.

Class templates are instantiated immediately, at the point of use. For function templates, Clang generates a template specialization to be instantiated at the end of the translation unit. ConceptClang then stores the concept maps for the constraints of the template in

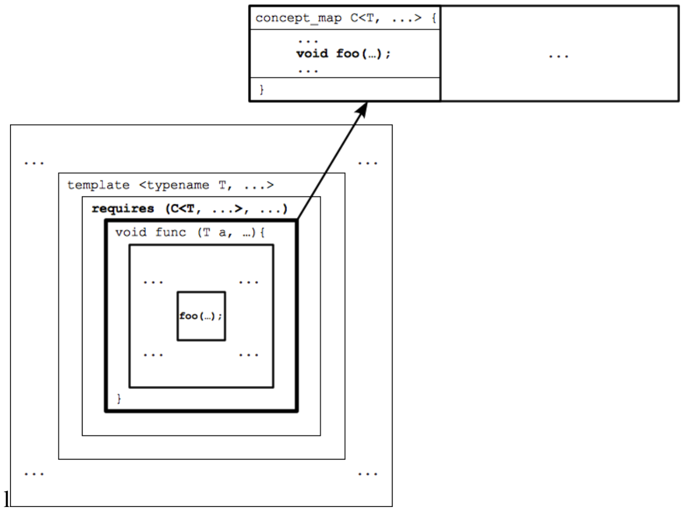


Figure 9. Name-lookup in restricted context

the object representing the specialization, for them to be used at instantiation time.

For concept map templates, the first part of the type checking consists of the concept map generation procedure, as discussed in Sect. 5.1.4. In essence, as Fig. 8(a) illustrates, the constraints act as a filter over the normal concrete concept maps generation procedure that causes the production of CMAs in places of concrete concept maps. The second part of type checking occurs when a concrete concept map declaration needs to be generated from a concept map template and concrete concept maps must be found or generated to substitute for concept map archetypes created in the first step. The instantiation of concept map templates occurs immediately during requirements satisfaction of templates.

Thus, there are three main implementation points as steps to consider when type checking constrained templates, whether the last two steps are separate or not: the template definition, the requirements satisfaction, and the instantiation of template specialization. In this section, we will be describing how ConceptClang approaches those steps, particularly for function templates.

5.2.1 Type Checking of Template Definitions

To type check a template definition, a *RestrictedScope* is first entered for checking of the template body. Then, a concept map archetype is created for each constraint, as discussed in Sect. 5.1.3, and added to the scope for name lookup.

Name lookup searches through scope chains, from the scope that initiates the lookup up to the translation unit scope. When a restricted scope is encountered, we "hijack" Clang's standard lookup path, and direct it to perform the lookup in the archetypes that form the constraints environment and, for some exceptions, in the surrounding scopes (cf. Fig. 9). We reuse most of Clang's name lookup procedure, with changes to deal with situations that only arise in restricted scopes.

First, we allow the lookup of template parameters outside of the restricted scope, when they are referred to from within a restricted scope. To do this, we modified *CppLookupName()* so it would continue following the normal lookup path, but only through those scopes for which the *TemplateParmScope* flag is set.

When a qualified name is encountered (a name of the form $X::Y$ where Y is to be looked up in the scope of X), ConceptClang checks if the qualifier is correct in a constrained scope (currently the check described here is implemented only partially). If the name qualifier is not a concept id, then it is handled as it normally would be. If the qualifier is a concept id, it is first looked up in the constraints, as described in Sect. 5.1.4. If it is not found in the constraints, then it is allowed as long as it names an existing concept map or a concept map template (requirements satisfaction check is performed for concept map templates before they are used).

Another exception is made for calls to function templates that depend on template arguments of a constrained template. For these calls, ConceptClang first checks that the called template is itself a constrained template. Then, it checks that the constraints of the calling template satisfy the constraints of the called template in a process similar to checking concept map templates (Sect. 5.1.4, Fig. 8(b)): template parameters, template arguments, and constraints of the called template are analogous to parameters, arguments, and refinements of the constrained concept map template. And the constraints of the calling template are analogous to the constraints of a constrained concept map template.

Finally, if lookup failed in the restricted scope after first parsing a non-dependent call expression within the restricted context, we temporarily unset the *RestrictedScope* flag to repeat the name lookup process as we would do for normal function calls.

5.2.2 Template Requirements Satisfaction

The requirements satisfaction check finds or generates concept maps for each constraint of a constrained template. If the map found is a template, then it tries to generate a concrete map from it, or, if the generation fails, it produces an error. The generation of a concept map template may fail just like any other concept map may, due to errors in implicit and default definitions, but, as a constrained template, it may also fail due to issues discussed in Sect. 4.4.2. When the failure occurs due to late checking related to optimizations, ConceptClang produces an error.

Finding Concept Maps When a map is requested for a concept, ConceptClang looks the map up in the *FoldingSet* of maps held by the concept. If the search is unsuccessful, it looks through the *FoldingSet* of map templates for the closest match. If the search is still unsuccessful, it may be the case that the map is simply not in the immediate declaration context in which the constrained template is used. In that case, the search is repeated with the value of the context updated to the surrounding context, up to the translation unit context. Only namespace contexts and translation units are considered in this iterative process.

When no concrete concept map exists for a given constraint, there may be more than one concept map templates available. Concept map templates are ordered just like normal class templates in C++, and the best template is chosen reusing Clang's implementation for ordering class templates. If there is an ambiguity, concept map lookup fails, and ConceptClang prints out all the candidate template maps along with the reason for the failure of each one of them.

Generating a map from a map template. When a concept map template is used, appropriate template arguments, corresponding to the 1-level template parameters list of the map template, must be deduced for final substitution. To deduce the template arguments, ConceptClang reuses a modified version of Clang's procedure for deducing the arguments for partial specializations of class templates. After argument deduction is completed, the requirements of a concept map template are checked. Then, a new concrete map incorporating these arguments is created and the members of the concept map template are type-substituted and copied over to the concrete map. Finally, the concrete map is stored in the list of concept maps for the mapped concept.

Requirement Satisfaction and Concept Map Generation. In ConceptClang, the procedures for checking constraints and generating concept maps are mutually recursive, and reused often (cf. Figs. 7and 8).

The constraints check procedure is re-used to check that a concept map satisfies its associated requirements and refinements when type-checking the map. The procedure for generating map archetypes reuses the map generation procedure, but does not attempt to find existing implementations. The procedure for generating maps from map templates consists of logic similar to the map generation procedure, but it greatly differs in the implementation details. Thus, the two procedures are implemented separately. To get a better idea of how these procedures work, the reader is encouraged to check out the implementations of the *ActOnConceptMapMembers*, the *CheckTemplateConstraints* and the *BuildConceptMapFromPartialMap* procedures [34].

5.2.3 Template Instantiation

After the uses of templates are checked, the specializations of templates must be instantiated to generate specialized code, based on the maps generated from requirement satisfaction. At this time, expressions and statements constituting the bodies of the templates must be appropriately substituted, i.e., references to members of concept map archetypes, resulting from name-lookup resolution during the type checking of templates, must be updated to refer to members of corresponding concrete concept maps.

To this end, ConceptClang has adjusted the use of Clang's *SubsStmt()* procedure to account for the list of concept maps as well. The procedure thus uses the list to invoke the necessary transformations.

For call expressions to associated functions of concepts, ConceptClang simply "re-points" the calls corresponding members of concrete maps. Variable declarations and references to associated types receive a similar treatment.

Re-pointing Call Expressions. Given a function call and the concrete template arguments from a template specialization, we substitute the arguments into the archetype that the call points to. Then, we use the substituted archetype to find its corresponding concept map in the provided list of maps. With the appropriate map found, ConceptClang repeats name lookup for the function name, transforms the arguments of the call substituting template arguments for template parameters, and rebuilds the call expression. The lookup is performed directly in the map only, via temporarily setting a ConceptClang scope flag *RestrictedInstantiationScope* . Call rebuilding goes through overload resolution again. The rebuilt call expressions are further processed using Clang's standard implementation.

Candidate Sets or Not? The concepts wording requires that concept maps store candidate sets for implementations of associated functions. ConceptClang currently implements a part of that functionality. Concept maps indeed store overload sets for each associated function, and overload resolution is performed at instantiation time. However, our implementation of concept map generation always puts a single implementation in the candidate set that corresponds to the best match or the *seed* of a candidate set.

Re-pointing References to Types We transform types similarly to the way we transform call expressions. When we detect a type that is introduced by a concept map archetype, we substitute the type arguments into the archetype, find the appropriate map in the available list of maps, and use the arguments list of that map to appropriately transform the type with a new instantiation object.

5.3 Concept-Based Overloading

To pick the best function overload based on constraints, all overloads must be compared pairwise, checking if constraints of one overload require constraints of the other. This mechanism is similar to calling a constrained function template from another constrained function template, which suggests a reuse of the requirements satisfaction procedure described in Sect. 5.2.2. However, to reuse the procedure, one must have access to the template arguments of the function call expression that prompted the overload resolution process. Unfortunately, at the time when the best viable candidate must be chosen, Clang no longer has access to template arguments. Therefore, we cannot reuse our constraint check procedure. ConceptClang extends the mechanism by which Clang finds the most specialized function. Given two function templates, their constraints are compared. If the list of constraints for the first template contains all the constraints of the second template, then the first template is *at least as specialized* as the second one.

Currently, our implementation of overload resolution always picks a constrained template over unconstrained ones if a function is called in an unconstrained context. The concepts wording requires that for calls in unconstrained contexts, unconstrained templates should be chosen if their signatures are a better match for the call. We plan to implement overload resolution required by the concept wording, and select between the two implementations with compiler flags, providing a platform for experiments.

5.3.1 Requirement Satisfaction Failure Is Not An Error

Another element inherited from C++'s function overloading mechanism is the proper assessment of failures of overload resolution. When no viable candidate has been found, the compiler must communicate what the candidates actually were, and why they failed.

ConceptClang must also be extended this way. Clang treats template argument deduction as Substitution Failure Is Not An Error (SFINAE) contexts. If a deduction failure occurs, its results are collected in a *TemplateDeductionInfo* structure. ConceptClang introduces a *ConstraintsCheckFailureInfo* structure to collect error diagnosis for templates that were not substituted because of unsatisfied constraints. When substitution error occurs in a constrained template, the *TemplateDeductionInfo* object is updated with the resulting *ConstraintsCheckFailureInfo* object. When no viable candidate is found, Clang uses the information from the *TemplateDeductionInfo* to inform the programmer which templates were considered but were rejected because of SFINAE. In this way, ConceptClang has notably improved error diagnostics through its support for separate type checking of templates.

6. Current State and Future Work

In this section, we explore the current state of our implementation and discuss our plans for future work. We start with a basic example highlighting the improvement of error diagnostics as expected from working with concepts. We also present a more sizable example of a minimal implementation of the Boost Graph Library [26] based on the implementation by Garcia et al. [11].

6.1 Basic Example: Apples-to-Apples

The *Apples−to−Apples* program (Fig. 10) defines a generic algorithm *pick()* for comparing objects of the same type. The types of the algorithm are constrained by the *Comparable* concept, which specifies an associated function *better()*.

For illustration of ConceptClang's diagnostics, we are interested in three scenarios:

1. When a model implementation for *orange* is not provided.
2. When the model is provided but is missing an implementation of an associated function.
3. When a constrained template calls a function that cannot be found in its constraints.

```
Concept Comparable<typename T> {
  bool better (const T &a, const T &b);
}
concept_map Comparable<int> {
  bool better(const int &x, const int &y) {
    return x > y;
  }
}
template <class T>
requires (Comparable<T>)
const T& pick(const T& x, const T& y) {
  if (better(x, y))  return x;
  else  return y; // return foo(y); (Fig. 13)
}
int main(int argc, char **argv) {
  int i = 0, j = 2;
  orange o1("Navel"), o2("Valencia");

  int k = pick(i, j);
  orange o3 = pick(o1, o2);  // error in Fig. 11
}
```

Figure 10. Apples-to-Apples example program.

```
Test.cpp:34:17: error: no matching function for
                       call to 'pick'
    orange o3 = pick(o1, o2);
                ^~~~
In file included from Test.cpp:10:
./Test.h:15:10: note: candidate template ignored:
      constraints check failure [with T = orange]
const T& pick(const T& x, const T& y) {
         ^
./Test.h:14:21: note: Concept map requirement not met.
requires (Comparable<T>)
                    ^
./Test.h:14:1: note: Constraints Check Failed: pick.
requires (Comparable<T>)
^
```

Figure 11. Undefined model.

```
./Maps.h:33:13: error: Concept map generation Failed:
      Comparable.
concept_map Comparable<orange>
            ^
In file included from Test.cpp:10:
In file included from ./Test.h:10:
In file included from ./Maps.h:15:
./Concepts.h:18:10: note: Associated declaration is
      undefined in concept map for concept 'Comparable'
    bool better (const T &a, const T &b);/* {
         ^
In file included from Test.cpp:10:
./Test.h:16:9: error: use of undeclared identifier
      'better'
    if (better(x, y))
        ^
Test.cpp:35:17: note: in instantiation of function
      template specialization 'pick<orange>' requested
      here
    orange o3 = pick(o1, o2);
                ^
```

Figure 12. Error defining concept map.

Figs. 11, 12, and 13 show the errors that ConceptClang produces in each scenario. It is worth noticing how notes following each error message point directly to the source of the problem whenever the error message itself is not informative enough.

6.2 Practical example: Mini-BGL

For a more practical example, we consider a minimal version of the Boost Graph Library which implements the breath first search algorithm with concepts, as illustrated in Fig. 14. The full program, with all the necessary concepts, concept maps and data structures, is available online [34].

```
./Test.h:19:16: error: use of undeclared identifier
      'foo'
        return foo(y);
               ^
```

Figure 13. Error detection at definition time.

```
template <typename G, typename C, typename Vis,
          typename QueueType>
requires (VertexListGraph<G>, IncidenceGraph<G>,
       ReadWriteMap<C>, Color<ReadWriteMap<C>::value>,
       BFSVisitor<Vis,G>,  Bag<QueueType>)
void breath_first_search(const G& g, vertex s, C c,
                         Vis vis, QueueType& Q) {
  pair<vertex_iter,vertex_iter>
                        iter_pair = vertices(g);
  for(vertex_iter iter = iter_pair.first,
               iter_end = iter_pair.second;
    iter != iter_end; ++iter)  {
    vertex u = *iter;
    initialize_vertex(vis, u, g);
    set(c, u, white());
  }
  // call to graph_search(g, s, vis, c, Q)
  // is inlined here in the current version
}
```

Figure 14. The breath-first search algorithm with concepts.

6.3 Known issues

As expected of work in progress, our implementation is not complete or free of errors. Although we have a working state of the breath first search algorithm, our regression tests are still minimal. We are not yet supporting some important features such as same-type constraints, associated function templates, and explicit refinements.

Our handling of concept ids used as name qualifiers can currently cause infinite loops in the compiler whenever the a concept id refers the same context in which it is used. The fix for this issue will involve adding proper safeguards in our concept map archetype structures, but we have make sure that the safeguards do not disable some valid qualifiers.

The version of Clang that our implementation is currently based on is not the most recent. In fact, in the version of Clang that ConceptClang is based on, template type parameter types are not yet uniquely identified by the context from which they were introduced, in addition to the other criteria listed earlier (e.g. depth, index, etc.). As a result, instantiations of constrained function templates called from other constrained templates, as well as instantiations of associated functions of concept maps built from concept map templates, are faulty due to ambiguities in the unique identification of the types. The current version of Clang incorporates template type parameter declarations (the declaration contexts of which introduce the types) in the definition of the types and their unique identification.

7. Conclusion

The main contribution of this paper is ConceptClang, a compiler for concepts-enabled C++. ConceptClang is based on Clang, a modern compiler frontend for the C family of languages. ConceptClang fills an important role in the C++ community. Concepts were one of the most expected features proposed for C++0x, the next standard of C++. Late in the standardization process, however, concepts were removed from the upcoming standard. The main reason for the removal was the lack of agreement on the impact that concepts would have on an average C++ programmer, and that impact could not have been properly understood without a compiler. The previous experimental compiler, ConceptGCC, was not complete enough to allow experiments with more advanced features of concepts. Concept-

Clang is based on a more transparent and modern compiler, and we believe that it will provide a transparent and comprehensible infrastructure to experiment with different designs of concept features. In addition, we present examples of diagnostics produced by ConceptClang for common scenarios of programming errors, and we provide an example based on the Boost Graph Library.

References *

[1] R. Anisko, V. David, and C. Vasseur. Transformers: A C++ program transformation framework. Technical Report 0310, EPITA/LRDE, 2003.

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

[3] P. Becker (ed.). Working Draft, Standard for Programming Language C++. Technical Report N2914=09-0104, ISO/IEC JTC1/SC22/WG21—The C++ Standards Committee, June 2009.

[4] J. Bernardy, P. Jansson, M. Zalewski, and S. Schupp. Generic programming with c++ concepts and haskell type classes—a comparison. *Journal of Functional Programming*, 20(Special Issue 3-4):271–302, 2010.

[5] CGAL. Computational Geometry Algorithms Library (CGAL). http://www.cgal.org/, Aug. 2008.

[6] Clang. Clang: a C language family frontend for LLVM. http://clang.llvm.org/, May 2011.

[7] V. David and M. Haveraaen. Concepts as syntactic sugar. In *Proc. 9th IEEE International Workshop on Source Code Analysis and Manipulation (SCAM)*, pages 147–156. IEEE Computer Society, 2009.

[8] B. Dawes and D. Abrahams. The Boost initiative for free peer-reviewed portable C++ source libraries. http://www.boost.org, May 2011.

[9] G. Dos Reis and B. Stroustrup. Specifying C++ concepts. In *Proc. 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 295–308. ACM Press, 2006.

[10] G. Dos Reis and B. Stroustrup. Templates aliases. Technical Report N2258=07-0118, ISO/IEC JTC1/SC22/WG21—The C++ Standards Committee, Apr. 2007.

[11] R. Garcia, J. Jarvi, A. Lumsdaine, J. Siek, and J. Willcock. An extended comparative study of language support for generic programming. *J. Funct. Program.*, 17(2):145–205, Mar. 2007.

[12] GCC. Gcc, the gnu compiler collection. http://gcc.gnu.org/, May 2011.

[13] D. Gregor. Concepts Go "To Core". http://conceptgcc.wordpress.com/2007/04/18/concepts-go-to-core/, Apr. 2007.

[14] D. Gregor. ConceptGCC: Concept extensions for C++. http://www.generic-programming.org/software/ConceptGCC/, Sept. 2008.

[15] D. Gregor. Type-Soundness and optimization in the concepts proposal. Technical Report N2576=08-0086, ISO/IEC JTC1/SC22/WG21—The C++ Standards Committee, Mar. 2008.

[16] D. Gregor. What happened in Frankfurt? C++-Next, The next generation of C++, http://cpp-next.com/archive/2009/08/what-happened-in-frankfurt/, Aug. 2009.

[17] D. Gregor and J. Siek. Implementing concepts. Technical Report N1848=05-0108, ISO/IEC JTC1/SC22/WG21—The C++ Standards Committee, Aug. 2005.

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

[19] J. Järvi, J. Willcock, and A. Lumsdaine. Concept-Controlled polymorphism. In F. Pfennig and Y. Smaragdakis, editors, *Proc. 2nd Int. Conf. on Generative Programming and Component Engineering (GPCE)*, volume 2830, pages 244, 228. Springer, 2003.

[20] D. Kalev. The state of the language: An interview with bjarne stroustrup. C++ founding father assesses the language on the eve of its new standard. http://www.devx.com/SpecialReports/Article/38813, Aug. 2008.

[21] D. Kalev. Bjarne stroustrup expounds on concepts and the future of C++. danny kalev asks bjarne the hard questions about concepts and C++'s future. http://www.devx.com/cplus/Article/42448, Aug. 2009.

[22] D. Kapur and D. Musser. Tecton: a framework for specifying and verifying generic system components. Technical Report RPI-92-20, Department of Computer Science, Rensselaer Polytechnic Institute, Troy, New York, 1992.

[23] LLVM. The LLVM Compiler Infrastructure. http://llvm.org/, May 2011.

[24] D. R. Musser, G. J. Derge, and A. Saini. *STL Tutorial and Reference Guide. C++ Programming with the Standard Template Library*. Addison-Wesley, 2nd edition, 2001.

[25] J. Siek and A. Lumsdaine. Concept checking: Binding parametric polymorphism in \cpp. In *Proc. 1st Workshop on \cpp Template Programming*, Erfurt, Germany, 2000.

[26] J. Siek, L.-Q. Lee, and A. Lumsdaine. *The Boost Graph Library: User Guide and Reference Manual*. Addison-Wesley, 2002.

[27] J. G. Siek and A. Lumsdaine. The matrix template library: Generic components for high-performance scientific computing. *Computing in Science and Engineering*, 1(6):70–78, Nov/Dec 1999.

[28] A. A. Stepanov and M. Lee. The Standard Template Library. Technical Report HPL-94-34, Hewlett-Packard Laboratories, May 1994. revised in October 1995 as tech. rep. HPL-95-11.

[29] B. Stroustrup. *The Design and Evolution of C++* . Addison-Wesley, Mar. 1994.

[30] B. Stroustrup. Evolving a language in and for the real world: C++ 1991-2006. *Proc. 3rd ACM SIGPLAN conference on History of Programming Languages (HOPL)*, pages 4–1–4–59, 2007.

[31] B. Stroustrup. The C++0x "Remove Concepts" Decision. *Dr. Dobb's*, July 2009.

[32] B. Stroustrup. Simplifying the use of concepts. Technical Report N2906=09-0096, ISO/IEC JTC1/SC22/WG21—The C++ Standards Committee, June 2009.

[33] H. Sutter. Trip report: Exit concepts, final iso c++ draft in 18 months. http://herbsutter.com/2009/07/21/trip-report/, July 2009.

[34] L. Voufo, M. Zalewski, and A. Lumsdaine. ConceptClang Project. http://www.generic-programming.org/software/ConceptClang/, May 2011.

* C++ standardization committee papers mentioned here can be found online at http://www.open-std.org/jtc1/sc22/wg21/docs/papers/