Computer Science and Information Systems 11(4):1361–1379 DOI: 10.2298/CSIS131209053M

SimpleConcepts: A Lightweight Extension to C++ to Support Constraints on Generic Types

Reed Milewicz[1], Marjan Mernik[2], and Peter Pirkelbauer[3]

[1] University of Alabama at Birmingham
Birmingham, AL 35294
rmmilewi@cis.uab.edu
[2] University of Maribor, Slovenia
marjan.mernik@uni-mb.si
[3] University of Alabama at Birmingham
Birmingham, AL 35294
pirkelbauer@uab.edu

Abstract. Generic programming plays an essential role in C++ software through the use of templates. However, both the creation and use of template libraries is hindered by the fact that the language does not allow programmers to specify constraints on generic types. To date, no proposal to update the language to provide concepts has survived the committee process. Until that time comes, as a form of early support, this paper introduces SimpleConcepts, an extension to C++11 that provides support for concepts, sets of constraints on generic types. SimpleConcepts features are parsed according to an island grammar and source-to-source translation is used to lower concepts to pure C++11 code.

Keywords: Generic Programming, C++ Templates, C++ Concepts

1. Introduction

Generic programming is made possible in C++ through the use of templates [19]. Templates are language constructs that operate with generic types and that are instantiated as needed during compile-time [17]. Templates are ubiquitous in many C++ libraries and systems, most notably the Standard Template Library (STL), which provides generic implementations of commonly used containers and related algorithms. Essential to the STL are concepts, which are sets of constraints on types [1]. A type is called a model of a concept if that type satisfies all of its requirements, and templates can impose these concepts on their arguments; this is done to ensure type safety. For example, to be able to sort a list, its elements must support the < operator, and this requirement is expressed by the concept LessThanComparable. If the concepts associated with a template class or function are not satisfied, then the instantiation of that template code will fail, and a compile-time error will result. It is important to note that concept checking as implemented by many STL implementations is not directly supported by the C++11 language, but is rather the product of what Gregor *et al.* refer to as a "grab-bag of template tricks" [7]. The first issue with concepts as they are currently known is that they do not lend themselves to informative error messages. Violations cannot be reported without exposing the programmer to the details of the implementation. This means that compile-time errors can lead to dense

```
template<typename T>
void f(std::list<T> lst){
        std::sort(lst.begin(),lst.end());
        /* ... */
}

int main(){
        std::list<int> lst;
        f(lst);
}
```

Fig. 1. An example of C++11 code that violates implicit constraints

barrages of esoteric error messages that give the programmer little insight into what went wrong. A canonical example of seemingly innocuous code that exhibits this phenomenon can be seen in Fig. 1. The function f takes a list of elements of type T and, prior to working with the data, calls another templated function, std::sort, to sort the elements. At compile time, the compiler will attempt to instantiate the function f, substituting `int` for T. Doing so requires that the compiler also instantiate the sort function to take iterators of a `std::list<int>` type as input, but there is a problem: list iterators do not support the – operator which is expected by the sort function. The result is that the compiler produces 114 lines of error messages, traversing layers upon layers of template code. The error is not expressed to the programmer in a clear and consise way.

The second issue is that the complex nature of template metaprogramming makes it difficult to map concepts as described in the documentation to the specifics of their implementation. For those who create and maintain generic libraries, this means that it can be extremely difficult to detect bugs and other issues in their code. The root problem is that the C++ language lacks a construct to perform a vital function, and this forces the developers of generic libraries to resort to bricolage, cobbling together a functional equivalent from whatever materials they have at hand. This satisfies the immediate needs of the developers. However, in that concepts do not formally exist, they cannot be formally reasoned about or analyzed. For users, this is what leads to incomprehensible error messages when they misuse templates. For developers, this means that writing and maintaining template code becomes unnecessarily burdensome. The main contribution of this work is the introduction of SimpleConcepts, a lightweight extension to the C++ language to facilitate that development, maintenance, and usage of generic libraries. SimpleConcepts introduces several useful abstractions that perform the same role as concepts via metaprogramming while being easier to write, read, and formally analyze.

This paper is an extension of our previous work[12]. In this paper, the design of SimpleConcepts was extended to support the use of concepts in template overloading. In other words, constraints on generic types can now be checked as part of the process of overload resolution. A subsection was created in the implementation section that covers how this new feature works and gives detailed examples to illustrate that point. Additionally, new diagrams were added to help explain the source-to-source translation approach used in our approach. Finally, the discussion section was extended to cover the trade-offs of signature-based constraint checks.

This paper is organized as follows. In §2, we characterize the previous work that has been done to help modernize concepts in C++. In §3, we describe the problem domain

by investigating how concepts work in generic libraries such as the STL, and in §4 we introduce SimpleConcepts, give justifications for our approach, and show how the functionalities provided by concepts via metaprogramming map to the new model. In §5, we provide formalisms for the syntax and semantics of SimpleConcepts. In §6, we compare our approach to previous and contemporary alternatives. Finally, in §7 and §8 we provide discussion, conclusions, and plans for future work.

2. Background

The first comprehensive attempt to provide high level language support for concepts was the development of Tecton, a domain-specific language (DSL) for generic programming that was conceived by Stepanov, Kapur, and Musser in the late 1970s [10]. The work that was done on that language fed into the development of the STL by A. Stepanov [18]. After the Hewlett-Packard implementation of the STL was made publically available in 1994, the language evolved into one that specialized in concept specification, as seen in Musser's technical report in 1998 [14]; Tecton sought to provide a language-independent means of describing constraints on generic types. While it did not see widespread adoption, it did provide a formal framework through which concepts could be understood, laying the foundation for further developments. 1998 also marked the birth of the Boost libraries for C++ [4], and two years later the collection was extended to include the Boost Concept Check Library (BCCL), an effort spearheaded by Siek [15]. The BCCL is meant to provide clean and accessible mechanisms for programmers to use concepts in generic code, an improvement upon concepts as known in the STL. In the last decade, the goal of researchers shifted away from providing support for concepts in C++ by means of DSLs and library support and towards the incorporation of concepts into the C++ language as an extension [11]. A detailed review of this period has been provided by Voufo and Lumsdaine [23]. The most successful of these movements was ConceptC++, a project which culminated in a proposal to the C++ standardization committee that was not accepted in 2009 for want of simplification and more testing [6] [9]. It is from ConceptC++ and the work done in the intervening period that SimpleConcepts draws much of its inspiration.

3. Domain Analysis

Here we shall provide a description of concepts as they are known in the C++ Standard Template Library. As previously described, the term concept refers to a set of constraints or requirements on types. For instance, the concept EqualityComparable defines what it means for a type T to be comparable for equality. In C++, this means that the expressions `a == b` and `a != b` be valid for any two values `a` and `b` of type `T`. In the STL, concepts are implemented as assertions with the help of macros and made use of through statements such as

```
__STL_REQUIRES(X,_EqualityComparable);
```

which checks to see whether the type of a template parameter X models the concept EqualityComparable. Aside from requiring that a type support certain operations, concepts can also require that certain functions be supported or class members exist. For example, the concept Container requires that its models implement a size() method. Additionally, concepts may require certain associated types be defined (e.g. a size_type for

the value returned by a call to a Container's size() method). Lastly, the documentation for a concept may also state that certain invariants must be satisfied (e.g. identity or transitivity), but these are assumed to be the case, and no actual verification occurs. Concepts can be defined as a refinement of any number of previously existing concepts. A refined concept adopts the requirements defined by the concepts that it is refining. For instance, the concept ForwardContainer is a refinement of the concepts Container, EqualityComparable, and LessThanComparable (if its elements model LessThanComparable). That is, a ForwardContainer is a Container that also requires that its elements be comparable to one another.

In order to produce an extension to the C++ language that captures what these concepts do, we have to provide a formalism that allows us to understand concepts independently of their realizations. It would be a mistake to begin with a deconstruction of the syntax and semantics of STL concepts, because we are less interested in what they are; rather, we seek to describe what they are meant to be. We ought to begin with seeing the task of identifying and expressing concepts as a problem domain that happens to intersect with the task of generic programming. This domain is not complete or self-contained as we cannot speak of constraints on types without dealing with the particulars of some type system. However, concepts, as entities in their own right, can be reasoned about. Following the work of van Deursen and Klint [5], we shall give a formal description of STL concepts using the Feature Description Language (FDL). From attempting to derive a formal description of concepts from our informal description, we can discern several truths. First, concepts cannot be anonymous. They must exist prior to and independently of the circumstances in which they are used, and therefore must be named. Second, concepts must have one or more generic type parameters. Concepts connect constraints to types, and a concept that does not do this is an invalid concept. Third, concepts are allowed to have no constraints, that is, they can be empty. A concept with no constraints is trivially satisfied; all types are models of an empty concept. This may seem puzzling at first, but consider that a concept that introduces no new constraints can still be useful if that concept is a refinement of two or more concepts, because it implicitly expresses the union of those concepts. Fourth, a concept may refine one or more other concepts. To refine a concept is to implicitly add its constraints to the concept.

Lastly, comparing the informal description of STL concepts and the diagram shown in Fig. 2, one may note that invariants are not listed as a feature of concepts, and this is deliberate. Invariants are the consequences of satisfying both the syntactic and semantic requirements of a concept. To state that invariants are a feature of concepts means that there exists some has-a relationship between the two, when in fact this is not the case. For example, the STL documentation states that the reflexivity of the == operator is an invariant of the concept EqualityComparable, that is, `x == x` for all `x` of type `T` that is a model EqualityComparable. From a mathematical perspective, the invariant naturally follows if we assume the conventional definition of equality. However, in that C++ is a language that allows for operator overloading, knowing that a type supports the == and `!=` operators does not tell us whether those operators behave in some prescribed way. From this, we can conclude that there must be a limit to the enforceability of concepts with regards to their semantics. At the very least, finding the means to do so goes beyond the scope of this paper.

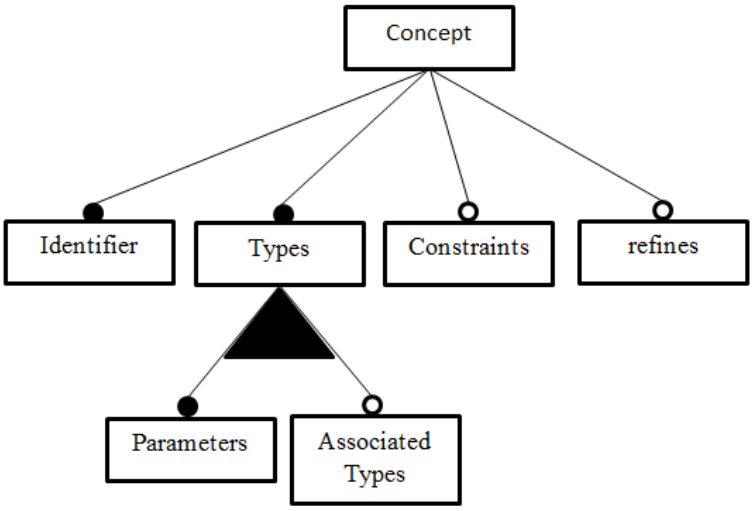

Fig. 2. FDL Diagram for Concepts

```
concept EqualityComparable<typename T> {
        bool T::operator==(T rhs);
        bool T::operator!=(T rhs);
}
```

Fig. 3. SimpleConcepts concept definition

4. Concepts

In SimpleConcepts, concepts are first-class representations of constraints on type parameters of templates. The concepts of SimpleConcepts obviate the need to use template metaprogramming to specify what a template requires of its type parameters. We designed SimpleConcepts with the following goals in mind:

1. To provide the same functionality as STL concepts while allowing programmers to write concept code that preserves readability and allows the compiler to produce meaningful error messages.
2. To make our extension to the C++ language as lightweight and undemanding as possible, providing more expressive power yet preserving the efficiency of template programming.

A concept definition consists of a declaration and a body containing concept member specifications. Fig. 3 depicts the definition of the concept EqualityComparable.

This concept definition establishes what it means for a type T to be comparable for equality. Once a concept has been defined, it can be used in template code by means of a

```
template<typename R> requires EqualityComparable<R>
bool foo(R x, R y) { /* ... */ };
```

Fig. 4. Requires clause

"requires" clause as is shown in Fig. 4. The requires clause places a restriction on the type R to being one which supports the == and != operators. Attempting to use the function foo with any x and y of a type that does not support these operators will lead to a compile-time error, informing the programmer that the EqualityComparable concept was not satisfied.

5. Implementation

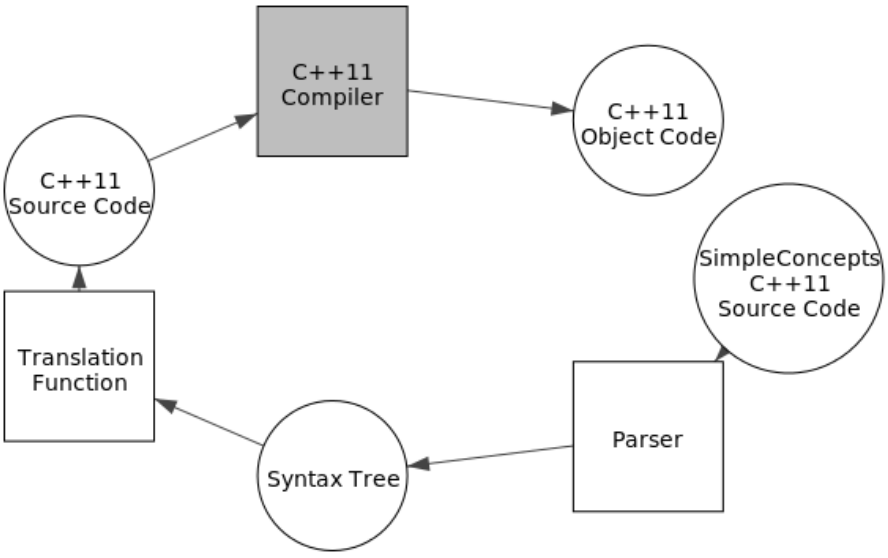

Fig. 5. The SimpleConcepts approach. Circles indicate input and output artifacts, and rectangles indicate processes. Pre-existing components are colored gray.

As was stated previously, a major consideration in the design of SimpleConcepts was to produce an undemanding extension, one that did not require significant overhaul or that could break existing code. Our approach can be summarized as follows.

First, the source code is parsed according to an island grammar that allows us to identify SimpleConcepts features [13]. The parser produces an abstract syntax tree that is then passed to the translation routine. The translator traverses this AST and replaces

Table 1. Abstract syntax and syntactic domains of SimpleConcepts

Type Variables	$\alpha \in TyVar$
Concept Names	$s \in CName$
Member Names	$f, a_t \in MemName$
Concept	$C \in \mathbb{C} := \text{concept} C_{id}\ \{B\}; \mid \text{concept} C_{id}\ \text{requires} R\ \{B\}$
Concept Identifier	$C_{id} := s < P >$
Concept Parameters	P := $\alpha P \mid \alpha$
Refinement Clause	R := $C_{id} R \mid C_{id}$
Concept Body	B := $MB \mid \epsilon$
Concept Member	M := $f\ func; \mid \text{typename}\ a_t$
Constrained Template	$T \in \mathbb{T}_c := \text{template}\ < params >\ \text{requires}\ R\ template\_body$

Here $\mathbb{C}$ refers to the set of all concepts, and $\mathbb{T}_c$ refers to the set of all constrained templates.

SimpleConcepts constructs with semantically equivalent C++11 constructs and then reconstitutes the C++11 source code. The resulting source code is then passed to an ordinary compiler, which generates the object code. A diagram illustrating this process can be found in Fig. 5.

There are several advantages to this approach. First, this approach allows us to handle both the syntactic and semantic analysis of concepts without need to modify an existing compiler. Second, using an island grammar allows us to analyze the syntax of concepts in a context-free fashion. Lastly, by means of translational semantics, we are able to express the meaning of concepts in terms of language features whose semantics are already well-known [3]. The remainder of this section addresses the details of this compilation model.

5.1. Abstract Syntax of SimpleConcepts

SimpleConcepts extends the C++ language to provide two new language constructs: concepts and constrained templates. A concept specifies a set of member function declarations and associated types, and a list of other concepts that are refined by it. A constrained template is one that has a concept requirement clause, dictating what restrictions are placed on the template's parameters. Table 1 gives the abstract syntax and the syntactic domains for SimpleConcepts.

5.2. Concrete Syntax of SimpleConcepts

An island grammar is a context-free grammar which describes some subset of the features of a language and uses catch-all productions to ignore all else; the name of this technique is derived from the view that the features we are interested in capturing with the grammar are "islands" amidst a vast sea of other language features [13]. There are several reasons why an island grammar is why we consider an island grammar to be an ideal choice for SimpleConcepts.

First, SimpleConcepts, as a language extension, intersects with C++11 in a very limited way. Concept definitions are unrelated to any existing C++11 language features, and

Table 2. Concrete syntax of SimpleConcepts

<concept-id> := <concept-name> "<" <concept-parameter list> ">"
<concept-name> := **<identifier>**
<concept-definition> := "concept" <concept-id> <requires-clause>? <concept-body> ";"?
<concept-body> := "{" <concept-member-specification>? "}"
<concept-member-specification> := <concept-member-specifier> <concept-member-specification>?
<concept-member-specifier>:= <associated-function> | <associated-type>
<associated-function> := **<function-definition>**
<associated-type> := **<typename-specifier>**
<requires-clause> := "requires" <requirement-list> | "requires" "(" | <requirement-list> ")"
<requirement-list> := <requirement> "&&" <requirement-list> | <requirement>
<requirement> := <concept-id>
<declaration> := <concept-definition>
<template-declaration> := "template" "<" **<template-parameter-list>** ">" <requires-clause>?
<concept-parameter list> := **<template-parameter-list>**

Non-terminals that refer to pre-existing C++11 features are in bold for clarity.

constraints on templates are merely addenda on template definitions. Since we are only interested in parsing concept definitions and their uses so that we can perform the necessary translations, using a grammar that allows us to identify salient features while skipping the rest is ideal.

Second, using an island grammar means that any implementation of our approach remains adaptable as the C++ language continues to evolve. The alternative, extending an existing parser to support SimpleConcepts, would entail frequent maintenance to stay current. An island parser sidesteps this problem by exploiting the loose coupling between the language extension and the original language, allowing us to handle the latter in a generic way.

One drawback to this genericity is that lexical and syntactic errors that fall outside of the purview of the island grammar will go undetected and will be preserved by any transformations. As a consequence, when the compiler reports the errors, it will expose the transformed source code to the client, which may not align well with the original source code that they provided. However, in our particular case this would only be a problem if the errors occurred within a template declaration or body.

Table 2 gives an EBNF grammar for SimpleConcepts.

5.3. Summary of Translation Model

Our approach is based on the work done by David and Haveraaen [22], which describes a means of converting ConceptC++ code to pure C++03 code by translating the concepts of ConceptC++ into sets of class templates. Here we shall use a toy problem as a vehicle to explore the translation scheme. Consider the code fragment in Fig. 6.

```
concept MutualExclusion<typename M>{
        void M::lock();
        void M::unlock();
}

template<typename L, typename V> requires MutualExclusion<L>
V read(L* lock,V* ref){
        lock->lock();
        V value = *ref;
        lock->unlock();
        return value;
}
```

Fig. 6. Example of SimpleConcepts in action

```
template<typename M>
struct MutualExclusion {
        MutualExclusion_Requirements<M> rq0;
}

template<typename L, typename V>
V read(L* lock,V* ref){
        while(false) { MutualExclusion<L>(); }
        lock->lock();
        V value = *ref;
        lock->unlock();
        return value;
}
```

Fig. 7. The first layer of the translation

A type M models the concept MutualExclusion if it provides two member functions, lock and unlock, that take no arguments and whose return type is void. In other words, M can provide a means of mutual exclusion; M could be a mutex type as defined in the standard libraries, a user-defined mutex type, or a type that provides an interface to a mutex. Below the concept definition we see an example of a function template that uses the concept MutualExclusion. The function read guards access to a value stored in a shared memory location by means of a mutex supplied by the client.

We shall describe, step by step, the translation process. The translation from concept code results in what can be seen as three distinct layers of template code. Fig. 7 shows the first layer of the translation.

The struct MutualExclusion has, as a member, another struct representing the requirements associated with the concept MutualExclusion. The requirements are separated from the concept itself in order to support refinement; a concept that refines another concept "inherits" the requirements from its ancestor, and the ancestor's requirements struct is listed there as well. Attempting to instantiate the MutualExclusion template will require the instantiation of the template MutualExclusion_Requirements. If that instantiation fails, the code will fail to compile. To implement constraints on template functions, we trigger the instantiation of MutualExclusion in the template function as seen in the template function code. The call to the constructor of MutualExclusion is placed in an unreachable block of code to guarantee that no run-time overhead will result. Now we examine the second layer of the translation: expressing the requirements enumerated by

```
CREATE_MEMBER_FUNC_SIG_CHECK(lock,void (M::*)(void));
CREATE_MEMBER_FUNC_SIG_CHECK(unlock,void (M::*)(void));
template<typename M>
struct MutualExclusion_Requirements {
        static_assert(has_member_func_lock<M>::value,
            "The member function 'lock' is not available
              or does not match signature.");
        static_assert(has_member_func_unlock<M>::value,
            "The member function 'unlock' is not available
              or does not match signature.");
};
```

Fig. 8. The second layer of the translation

```
template<typename T, T>
struct match_signature : std::true_type {};

template<typename M, typename = std::true_type>
struct has_member_func_lock : std::false_type {};
template<typename M>
struct has_member_func_lock<M, std::integral_constant
< bool , match_signature<void (M::*)(void), &M::lock>::value >>
: std::true_type {}

template<typename M, typename = std::true_type>
struct has_member_func_unlock : std::false_type {};
template<typename M>
struct has_member_func_unlock<M, std::integral_constant
< bool , match_signature<void (M::*)(void), &M::unlock>::value >>
: std::true_type {}
```

Fig. 9. The innermost layer of the translation.

a concept. A Requirements struct contains a set of static assertions that express the requirements that must be fulfilled by a type or set of types in order to model a concept. These assertions are checked when the compiler attempts to instantiate the MutualExclusion_Requirements template. The code that we generate for the definition of MutualExclusion_Requirements is provided in Fig. 8.

The macro CREATE_MEMBER_FUNC_SIG_CHECK provides template code necessary to check the existence of a member function that matches both the name and the signature specified by the concept. That is, the instantiation of has_member_func_lock will always succeed, but its member "value" will be true if and only if M has a lock method that matches the signature specified in the concept definition. This, in turn, determines whether the corresponding static assertion succeeds or fails. The code generated by the pre-processor is shown in Fig. 9.

This then is the third and innermost layer of the translation. Our mechanism verifies the existence of the member functions lock and unlock in a way that give true or false values which are used in the static assertions. This allows us to report meaningful error messages.

5.4. Semantics of SimpleConcepts

We shall describe the translational semantics of SimpleConcepts. In that concepts and constrained templates are ultimately converted to template code, the semantics of concepts

$F_C(C \in \mathbb{C}) = \{C_{id}\,\{\{requirements_{C_{id}}\,\langle P\rangle\}\ \cup\ \{requirements_r\,\langle P_r\rangle\,\forall r \in R\} \in \mathbb{T}, requirements_{C_{id}}\,\langle P\rangle\,\{F_{m1}(B)\} \in \mathbb{T}), F_{m2}(B)\}$
$F_{m1}(B) = \{static\_assert(has\_member\_func\_fname\,\langle P\rangle :: value, errormsg)\forall f \in R\} \cup \{typedef a_t \forall a_t \in R\}$
$F_{m2}(B) = \{CREATE\_MEMBER\_FUNC\_SIG\_CHECK(fname, fsignature)\forall f \in B\}$

Fig. 10. A formal description of the translation function

```
// function member checking template macros here,
// one for each associated function f ∈ B
template<P>
struct ConceptName_Requirements {
  // A list of static assertions, one for each associated function f ∈ B,
  // and a list of typedefs, one for each associated type definition a_t ∈ B
};
template<P>
struct ConceptName {
        ConceptName_Requirements<P> rq_c;
        id_0_Requirements<P_0> rq_0;
        // ... for every concept refined by this concept ...
        id_n_Requirements<P_n> rq_n;
};
```

Fig. 11. Summary of the output of the translation function

are a subset of the semantics of templates. With that in mind, we adapt the work done by Siek and Taha [16] to provide formalisms to describe the semantics of C++ templates. By providing a mapping from the abstract syntax of SimpleConcepts to that of C++ template code, we can then make the jump to the semantics. We represent our translation function as a set of functions that map SimpleConcepts code to C++11 code. The first subset of these translation functions, defined in Fig. 10, describes the translation of concepts and their members (note that $\mathbb{T}$ refers to the set of all C++11 templates).

In terms of the concrete syntax, this translation amounts to the code fragment depicted in Fig. 11.

The second subset of the translation functions transform constrained function and class templates into C++11 legal templates. These can be summarized as follows: for a constrained function template, our translation moves the concept requirements into the body of the function as calls to constructors; for a constrained class template, the concept structs are made members of the class. The result of this translation scheme is that instantiations of constrained class templates amount to a chain of instantiations that perform the necessary checks to confirm that the types involved are models of the concepts required. Concept instantiations lead to requirement template instantiations, and those in turn lead to member function checking template instantiations. With that, the only way to make use of a constrained template class or function is to supply it with valid types, or compile-time errors will result. As for the concepts themselves, whose capabilities are limited to checking the existence of function members, we know that our translation does exactly that and nothing more.

```
concept A<typename X>{
        void X::foo();
}

concept B<typename X> requires A<X>{
        void X::bar();
}

template<typename T>
void f(T v){
        /*...*/
}

template<typename T> requires A<T>
void f(T v){
        /*...*/
}

template<typename T> requires B<T>
void f(T v){
        /*...*/
}
```

Fig. 12. An example of template overloading with SimpleConcepts

5.5. Extending SimpleConcepts to support template overloading

In this section, we show how the SimpleConcepts approach can be extended to support template overloading based on constraints. This will allow us to selectively instantiate different versions of template functions and classes depending on whether or not certain concepts are modelled. In Fig. 12, we have two concepts, A and B, where B is a refinement of A, and three template functions, one which imposes no constraints on any of its parameters, one which requires A, and finally one which requires B. Ideally, we would like to instantiate the third template if type T models B, and the second if A but not B, and the first if neither A nor B hold. As is, a C++11-compliant compiler would fail to compile the translated SimpleConcepts source code, due to an unresolvable ambiguity between the three template functions; we move the concept checks into the bodies of the functions, yet the template declaration gives no indication of this.

To solve this, we must first find an ordering for these templates according to the extent of their restrictiveness. For any two constrained templates t_a and t_b, t_a is considered to be less specialized than t_b if the set of its required concepts, including all concepts that are refined by concepts in that set, is a proper subset of the set of required concepts of t_b.

A set of overloaded templates is unambiguous if we can find a unique best match for the arguments supplied in the client code. For all templates whose constraints are satisfied, there should exist a single template among them that has the strongest constraints; this then is the unique best match. Given that we have an ordering among templates according to the strength of their constraints, the algorithm for deciding the unique best match becomes simple: eliminate all templates except for those whose constraints are satsified and whose successors' constraints are unsatisfied. If a set of overloaded templates remains ambiguous, this will cause a compile-time error. Otherwise, the unique best match will be instantiated.

Our approach takes advantage of the "substitution failure is not an error" (SFINAE) principle [8]. The candidate set may contain many templates whose parameters will fail to

```
CREATE_MEMBER_FUNC_SIG_CHECK(foo,void (T::*)(void));
template<typename T>
struct A_Requirements {
static_assert(has_member_func_foo<T>::value,
"The member function 'foo' is not available or does not match signature.");
};

CREATE_MEMBER_FUNC_SIG_CHECK(bar,void (T::*)(void));
template<typename T>
struct B_Requirements {
static_assert(has_member_func_bar<T>::value,
"The member function 'bar' is not available
or does not match signature.");
};


template<typename X>
struct A {
        A_Requirements<X> rq1;
};

template<typename X>
struct B {
        B_Requirements<X> rq0;
        A_Requirements<X> rq1;
};

namespace f_0 {

template<typename T,
class = typename std::enable_if<
!((has_member_func_foo<T>::value && !(has_member_func_bar<T>::value))
|| (has_member_func_foo<T>::value
&& has_member_func_bar<T>::value) )>::type>
void f(T v){
        /*...*/
}
};
using f_0::f;

namespace f_1 {
template<typename T,
class = typename std::enable_if<
has_member_func_foo<T>::value && !(has_member_func_bar<T>::value)>::type>
void f(T v){
        while(false) {A<T>();}
        /*...*/
}
};
using f_1::f;

namespace f_2{
template<typename T,
class = typename std::enable_if<
(has_member_func_foo<T>::value) && (has_member_func_bar<T>::value)>::type>
void f(T v){
        while(false) {B<T>();}
        /*...*/
}
};
using f_2::f;
```

Fig. 13. The product of translating previous example of template overloading

be satisfied. In C++, the failure to find valid subsitutions for the parameters of a template causes the compiler to drop that template from the candidate set. We aim to cause the compiler to discard all templates whose constraints are unsatisfied or whose constraints that are weaker than any alternative.

To encode this algorithm and make it available to the compiler, we inject the necessary information into the type parameter list of each template by means of an `enable_if` clause. `enable_if` is a compile-time metaprogramming construct that allows us to selectively enable template functions and classes. Its declaration contains a boolean expression b and a type t (default `void`). If b holds true, then the `enable_if` struct will contain a public typedef member equal to t, and if b is false, then the struct will be empty. We can use this signal as a way of distinguishing the correct template to instantiate.

Here we construct a boolean expression using the signature checks associated with each concept that expresses the notion we described previously. The one template whose `enable_if`'s boolean expression holds true will be instantiated, and the remaining templates will be dismissed. Occasionally, some compilers may still treat this solution as ambiguous because they do not first evaluate the boolean expressions contained in the `enable_if` declarations. To ensure that they are properly evaluated, we can move each template into a unique namespace.

An example of this can be seen in Fig. 13. When a client calls `f(v)` for some `v` of a given type `X`, there are three possible outcomes:

1. If `X` models `B`, then `f_2::f` is instantiated.
2. If `X` models `A` but not `B`, then `f_1::f` is instantiated.
3. If `X` models neither `A` nor `B`, then `f_0::f` is instantiated.

6. Related work

As has been stated in previously, the absence of concepts in C++ is not a new problem; each attempt at a solution has built upon the groundwork laid by predecessors. In this section, we shall attempt to compare and contrast past and current approaches with SimpleConcepts.

6.1. Concepts Lite

At this time, there exists another proposal to provide support for C++ concepts by Sutton and Stroustrup known as Concepts Lite [20]. In this section, we shall attempt to compare and contrast that approach with SimpleConcepts.

Sutton and Stroustrup, the creators of Concepts Lite, have summarized their vision as "concepts = constraints + axioms" [21]. Concepts are abstract predicates that represent sets of requirements on generic types. These requirements can either be constraints or axioms. Constraints are syntactic requirements on the properties of generic types, which are checked at compile-time, and axioms are semantic requirements, analogous to the invariants of the STL documentation. As of the latest proposal, Concepts Lite supports constraints, but does not yet support axioms or concepts. With the understanding that the specifics of this proposal may be changed in the near future, we note the key differences between the two approaches.

```
template <typename T>
constexpr bool Addable()
{ return __is_valid_expr{bool={declval<T>() + declval<T>()}}
```

Fig. 14. Example of the syntax of Concepts Lite

First, while both SimpleConcepts and Concepts Lite share a similar notion of constraints, they differ greatly in terms of their implementation. In Concepts Lite, a constraint predicate is defined as a function template that contains a constant expression, referred to as a "use pattern". A type or set of types satisfies a constraint if the template can be legally instantiated, which is to say that the constant expression is valid. For example, a type `T` is Addable provided that the expression `a + b` is valid for any `a` and `b` of type `T`. In the syntax of Concepts Lite, this constraint might be expressed as can be seen in Fig. 14.

This approach differs from that of SimpleConcepts in three ways. First, Concepts Lite decouples concepts and constraints, which allows us to define Addable as a stand-alone constraint; in SimpleConcepts, which keeps constraints and concepts coupled, Addable would be expressed as a concept with a single constraint member. Second, Concepts Lite requires extensions to the compiler to support new intrinsics such as `__is_valid_expr`; SimpleConcepts requires no compiler extensions. Third and most importantly, whereas Concepts Lite uses constant expressions to define constraints, SimpleConcepts relies on function signatures.

Next, in contrast to Concepts Lite, this paper rejects the inclusion of axioms in its definition of concepts as going outside of the role that concepts are intended to fulfill, which is to provide compile-time support for templates. According to its authors, axioms are not statically evaluable, which implies that the question of whether a generic type truly models a concept, both syntactically and semantically, cannot be decided at compile-time. While we recognize the fundamental relationship between the two kinds of requirements, and we appreciate the simplicity and beauty of an approach that unifies them, we do not see a pressing need to incorporate axioms.

6.2. ConceptsC++

SimpleConcepts draws inspiration from the ConceptsC++ proposal, and in a certain sense can be seen as an evolution of it. In ConceptsC++, as in SimpleConcepts, concepts are sets of requirements, which can be expressed as signatures and associated types, and these concepts can be refined from other concepts, and they can be imposed upon generic types through the use of requirement clauses. Key to ConceptsC++ is its emphasis on retroactive modelling, that is, the ability to extend types to model concepts without modifying those types. This is accomplished through the optional use of concept maps, which detail how a type satisfies the requirements of a concept; concept maps can contain implementations of the functions required by the concept, either providing new functionalities or overriding exisiting ones. The idea has merit in that it provides a work-around to the problem of superficial syntax mismatches, but it demands that concept maps be given a considerable amount of expressive power, as they are able to define functions on their own and that they can themselves be templated. In contrast, SimpleConcepts does not support retroactive modelling, and does not support concept maps. In this way, we strive to be parsimonious,

offering the simplest and most targeted solution to the problem of constraining generic types.

There are also several other differences between the two approaches, which we shall list here:

- ConceptsC++ allows for non-member associated functions. The implementations of these functions can either be defined via a concept map or a default implementation can be provided in the concept itself. SimpleConcepts, meanwhile, requires that all associated functions be member functions.
- ConceptsC++ distinguishes between refinement clauses, and associated requirements, both of which allow a concept to "inherit" requirements from other concepts. In SimpleConcepts, no such distinction is made; a concept can have a requires clause that lists all of the other concepts that it draws requirements from.
- ConceptsC++ supports axioms. As was explained in the previous subsection, SimpleConcepts does not support this language feature.

6.3. Comparisons to Features of Other Languages

Java interfaces and SimpleConcepts concepts are mechanisms to impose constraints on types. Both can require that a set of declared functions be implemented by a type, and both can inherit requirements from one or more other interfaces or concepts. The key difference between the two is that interfaces are explicitly modeled, and SimpleConcepts concepts are implicitly modeled. In Java, classes are bound to interfaces through the use of the `implements` keyword; this allows for an object reference to be of an interface type. ConceptsC++ shares this notion of explicit binding in its use of concept maps. Meanwhile, in SimpleConcepts, types incidentally fulfill the requirements of concepts without self-affiliation with those concepts; references cannot be treated as being of a concept type.

This means that, ignoring refinement and inheritance, empty interfaces and empty concepts are semantically inequivalent. Whereas an empty concept is trivially modeled by all types, Java classes can be bound to empty interfaces which distinguishes those classes from those that do not implement the interface. Typically this is done in Java to express that a type obeys certain semantics (e.g. `Serializable`). SimpleConcepts does not allow the use of concepts as tags for the same reason that it does not allow for axioms: concepts should only express properties that are statically evaluable.

SimpleConcepts concepts also share much in common with the type classes of Haskell; a detailed treatment of the relationship between C++ concepts as they were known in ConceptsC++ and Haskell type classes can be found in Bernardy et al. 2010 [2]. As with concepts, type classes are parameterized constructs that express a sets of constraints on types. Type classes can be subclassed in the same way that concepts can be refined. In Haskell, types are associated with type classes through the use of user-defined instance declarations, which allows for retroactive modelling; SimpleConcepts does not have an analogous construct, instead relying on automatic modelling.

Like concepts, type classes can have multiple type parameters (this is not supported by the Haskell 98 standard, but many popular compiler implementations do support this feature). However, concepts and type classes diverge from one another on the use of functional dependencies. In the context of multi-parameter type classes, functional dependencies allow programmers to specify that type parameters are functionally related, that is,

that a type parameter may uniquely determine another type parameter. For example, a class Container might have two type parameters, ContainerType and ElementType, with a clause indicating that ElementType can be inferred based on the choice of ContainerType. These relationships are explicitly stated to aid in resolving ambiguities in the process of type inference. In SimpleConcepts, this feature is not necessary because C++ is a statically typed rather than a dynamically typed language.

Next, by virtue of the translational semantics of SimpleConcepts, there are several properties of C++ templates made available to concepts that are not available to Haskell type classes. For example, SimpleConcepts allows for concept parameters to have default values and value-level parameters, features that are not available to type classes. Most importantly, SimpleConcepts allows for concept-based overloading of templates. Haskell, meanwhile, does not provide any kind of overloading mechanism outside of type classes, and type classes alone are not sufficient to allow for this kind of specialization.

7. Discussion

Our approach is not without limitations. First, when translating refined concepts or constrained templates, it is assumed that the concepts that are being refined or used already exist, but this is not guaranteed to be the case. Referencing a non-existent concept will lead to a compile-time error, but that error is reported in terms of the translated code rather than the original source code, which could be problematic.

Second, one complication that signature-based constraints entail is the fact that a function's signature can be written in many superfically different ways. Within the context of a body of code produced by a single team, this is less of a concern, because defining concepts first helps to enforce conformity of style. However, when incorporating code produced by third parties, slight differences in signatures can lead to incompatibilities with pre-existing concepts.

While it would be trivial to extend SimpleConcepts to have an OR operator that would allow users to list a series of signatures that could satisfy a constraint, this could encourage bad practices; having to modify concepts over time to expand their coverage creates clutter and necessitates documentation for a feature that is intended to be self-documenting. Another possible solution would be for users to write new concepts to cover types introduced by foreign code, and to have an OR operator for `requires` clauses (i.e. `requires C<T> || C_alternate<T>`. However, this moves the problem of having to modify existing code to the template definitions, which is undesirable. Rather than include a feature that cannot fully or effectively deliver, we have chosen to leave out such extensions from SimpleConcepts as described in this paper.

This having been said, conservativeness is our aim, and the features that SimpleConcepts does provide are intended to be sufficiently expressive and intuitive to represent concepts in C++11.

8. Conclusion and Future Work

It has been shown that C++ lacks language support to specify constraints on generic types, and that this lack is the underlying cause of many difficulties for both the users and developers of generic libraries. To that end, we introduced SimpleConcepts, an extension to the

C++ language to provide such support. Our approach uses source-to-source transformations to provide an extension that is intended to be compatible with pre-existing C++11 compilers, while providing simple but powerful abstractions to aid in the design and use of C++ template libraries.

Moving forward, if the completed Concepts Lite is incorporated into the next iteration of the C++ language, then we shall turn our attention towards providing a formal analysis of such C++ concepts. In the short term, we hope to release a compiler front-end to provide experimental support for SimpleConcepts.

References

1. Austern, M.: Generic programming and the STL: using and extending the C++ Standard Template Library. Addison-Wesley (1998)
2. Bernardy, J.p., Jansson, P., Zalewski, M., Schupp, S.: Generic programming with c++ concepts and haskell type classes: A comparison. J. Funct. Program. 20(3-4), 271–302 (Jul 2010), http://dx.doi.org/10.1017/S095679681000016X
3. Bryant, B.R., Gray, J., Mernik, M., Clarke, P.J., France, R.B., Karsai, G.: Challenges and directions in formalizing the semantics of modeling languages. Comput. Sci. Inf. Syst. 8(2), 225–253 (2011)
4. Dawes, B.: Proposal for a C++ library repository web site (1998), http://www.boost.org/users/proposal.pdf
5. van Deursen, A., Klint, P.: Domain-specific language design requires feature descriptions. Journal of Computing and Information Technology 10, 2002 (2001)
6. Gregor, D., Stroustrup, B.: Proposed wording for concepts (revision 3) (N2421=07-0281) (10/2007 2007), http://www.open-std.org/JTC1/sc22/wg21/docs/papers/2007/n2421.pdf
7. Gregor, D., Stroustrup, B., Järvi, J., Reis, G.D.: Concepts: Linguistic support for generic programming in C++. In: SIGPLAN Notices. pp. 291–310. ACM Press (2006)
8. 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)
9. Kaley, D.: Bjarne stroustrup expounds on concepts and the future of C++ (2009)
10. Kapur, D., Musser, D.R., Stepanov, A.: Tecton: A framework for specifying and verifying generic system components (1983)
11. Mernik, M., Heering, J., Sloane, A.M.: When and how to develop domain-specific languages. ACM Comput. Surv. 37(4), 316–344 (Dec 2005), http://doi.acm.org/10.1145/1118890.1118892
12. Milewicz, R., Mernik, M., Pirkelbauer, P.: Simpleconcepts: Support for constraints on generic types in c++. In: Proceedings of the 2013 Federated Conference on Computer Science and Information Systems. pp. 1523–1528. IEEE Computer Society Press (2013)
13. Moonen, L.: Generating robust parsers using island grammars. In: The 8th Working Conference on Reverse Engineering. pp. 13–22. IEEE Computer Society Press (2001)
14. Musser, D.: Syntax of the tecton language (1998)
15. Siek, J., Lumsdaine, A.: C++ concept checking: A better practice for C++ programming (2001)
16. Siek, J., Taha, W.: A semantic analysis of C++ templates (2006)
17. Stepanov, A., McJones, P.: Elements of Programming. Addison-Wesley (2009)
18. Stevens, A.: Al Stevens interviews Alex Stepanov (1995)
19. Stroustrup, B.: The C++ programming language; 4th ed. Addison-Wesley (2013)
20. Stroustrup, B., Sutton, A., Voufo, L., Zalewski, M.: A concept design for the STL. Tech. Rep. N3351=12-0041, ISO/IEC JTC1/SC22/WG21—The C++ Standards Committee (January 2012)

21. Sutton, A., Stroustrup, B.: Concepts lite: Constraining templates with predicates (2013)
22. Valentin, D., Magne, H.: Concepts as syntactic sugar. In: SCAM. pp. 147–156 (2009)
23. Voufo, L., Lumsdaine, A.: A uniform terminology for C++ concepts. Tech. Rep. TR 703, Indiana University Technical Report (January 2013)

Reed Milewicz is a graduate research assistant at the University of Alabama at Birmingham. He completed his undergraduate studies at Birmingham-Southern College in 2011, at which he participated in undergraduate research initiatives in the field of computational aesthetics. He received his Master's in Computer and Information Sciences from UAB in 2013, where he is currently pursuing his PhD. His research interests include programming language theory, non-blocking algorithms, and static and dynamic analysis.

Marjan Mernik received his M.Sc., and Ph.D. degrees in computer science from the University of Maribor in 1994 and 1998 respectively. He is currently a professor at the University of Maribor, Faculty of Electrical Engineering and Computer Science. He is also a visiting professor at the University of Alabama in Birmingham, Department of Computer and Information Sciences, and at the University of Novi Sad, Faculty of Technical Sciences. His research interests include programming languages, compilers, domain-specific (modeling) languages, grammar-based systems, grammatical inference, and evolutionary computations. He is a member of the IEEE, ACM and EAPLS. Dr. Mernik is the Editor-In-Chief of Computer Languages, Systems and Structures journal since July 1, 2014.

Peter Pirkelbauer is an assistant professor at the University of Alabama at Birmingham. He received his PhD from Texas A&M University in 2010 and worked as postdoctoral researcher in the ROSE compiler group at the Lawrence Livermore National Laboratory. His current projects include scalable runtime error detection in parallel systems and the development of scalable and portable non-blocking data structures. His research interests include programming languages, source code analysis, transformation systems, and non-blocking programming techniques.

Received: December 9, 2013; Accepted: June 6, 2014.