View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by AMS Tesi di Laurea

Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea Magistrale in Informatica

CONCEPTS FOR C++: ANALYSIS OF ESTABLISHED AND NOVEL FEATURES

Tesi di Laurea in Linguaggi di Programmazione

Relatori:
Chiar.mo Prof.
COSIMO LANEVE
Chiar.mo Prof.
MAGNE HAVERAAEN

Presentata da:
MARCO POLETTI

Sessione III
Anno Accademico 2011/2012

Sommario

Questa tesi tratta dei "concept", una funzionalità per permettere una forma di polimorfismo limitato all'interno del linguaggio di programmazione C++. In particolare, si definisce concept un insieme di vincoli su uno o più tipi, che possono essere sia sintattici o semantici. Per vincoli sintattici si intende l'esistenza di specifiche operazioni su quei tipi, che vengono elencate nel concept; i vincoli semantici richiedono certe proprietà su queste operazioni, per esempio la commutatività di un'operazione binaria, e vengono formalizzati anch'essi dal programmatore all'interno del concept.

Dopo aver definito un concept è possibile utilizzarlo per definire funzioni e tipi polimorfi. All'interno del costrutto `template` del C++, che corrisponde ad una quantificazione universale, vengono inseriti i cosiddetti "concept requirement", che corrispondono ad un "tale che" nell'interpretazione matematica. Questi vincoli vengono utilizzati in un primo momento come interfaccia per risolvere le operazioni utilizzate nella funzione polimorfa e successivamente come predicato per determinare se l'ambiente chiamante può chiamare la funzione polimorfa. Certi sistemi di vincoli (compreso quello proposto nel lavoro di tesi) permettono un controllo di tipi separato del chiamante rispetto al chiamato. Non è invece possibile ottenere una compilazione separata del C++, a causa di alcune funzionalità avanzate del sistema di tipi, indipendentemente dai concept.

L'oggetto della tesi è lo studio di un linguaggio di vincoli adeguato, partendo da articoli esistenti sui concept in C++ e confrontando varie possibilità per ogni tipo di vincoli. Vengono inoltre proposte nuove caratteristiche del sistema di vincoli, come aggiunta o in sostituzione di quelle già esistenti. Viene dato particolare interesse al controllo di tipi separato e, nel contempo, al mantenere il più possibile la flessibilità tipica del polimorfismo in C++.

Viene fatta un'analisi approfondita dell'overloading basato sui concept, che porta a modifiche sul suo funzionamento e a linee guida sul suo utilizzo. La specializzazione di funzioni polimorfe, che sostituisce le soluzioni di ripiego utilizzate nel C++ odierno, viene integrata nel sistema di tipi senza impedire il controllo di tipi separato.

Inoltre, viene proposto un approccio per l'integrazione dei concept impliciti ed espliciti, uno dei principali punti di discussione fra i vari gruppi di ricerca sui concept. Tale approccio permette di ottenere un controllo di tipi più o meno stretto in base alle opzioni di compilazione, invece di annotare ogni concept con questa informazione come viene solitamente proposto. In questo modo è possibile

riutilizzare sia i concept sia il codice polimorfo che li utilizza in progetti che hanno esigenze diverse in termini di tempo di sviluppo e qualità del codice.

Le nuove funzionalità proposte comprendono gli "inner requirement", i moduli basati sui concept e l'aggiunta di implicazioni e quantificazioni esistenziali all'interno del linguaggio di specifica.

Gli inner requirement sono un costrutto per adattare i requisiti dichiarati da un'entità polimorfa a quelli dell'implementazione utilizzata, permettendo così la codifica dei morfismi che sono alla base della teoria delle istituzioni.

I moduli basati sui concept sfruttano la capacità di controllo di tipi separato per rimpiazzare l'inclusione testuale dell'intera implementazione di una libreria polimorfa all'interno del codice che la utilizza, che è l'unica possibilità nel C++ attuale.

Le implicazioni come parte del linguaggio di specifica ne aumentano l'espressività e permettono ad ogni componente di includere le relative parti della specifica, che altrimenti dovrebbero far parte di ogni modulo che istanzia il componente.

Le quantificazioni esistenziali permettono di specificare vincoli come "esiste una funzione f che prende un intero e restituisce un qualche tipo che soddisfa il concept C".

La tesi si conclude con una valutazione qualitativa del sistema di vincoli in base agli obiettivi in merito elencati in un articolo di Stroustrup, il creatore del C++, che è tuttora coinvolto nella discussione sui concept.

In appendice ci sono alcune funzionalità minori e la formalizzazione dei concept descritti informalmente nello standard C++ utilizzando il linguaggio di vincoli formato dalle funzionalità scelte durante la tesi.

Contents

Contents **3**

1 Introduction **11**
1.1 Concept literature . . . 11
1.2 Concepts . . . 12
1.3 Roles of concepts . . . 12
Concept as a set of requirements . . . 12
Concept as an API . . . 13
Concept as an implementation . . . 14
Discussion . . . 15
1.4 Uses of concepts . . . 16
Template parameter checking . . . 16
Domain engineering . . . 16
Documentation . . . 17
Testing . . . 17
Optimization . . . 17
1.5 Concepts vs interfaces . . . 18
1.6 Concepts vs programming by contract . . . 18
1.7 Tools for concepts . . . 19
1.8 Goals of the thesis . . . 19
1.9 Thesis structure . . . 20

2 Basic constraints **21**
2.1 Operation constraints . . . 21
Precise signatures . . . 21
Usage patterns . . . 21
Usage patterns with functionalization . . . 22
Fuzzy signatures . . . 22
Discussion . . . 23
2.2 Type specifications . . . 24
Simple type specification . . . 24
Associated types . . . 24
Discussion . . . 24

2.3 Default types and operations . . . 25
2.4 Constraints on constexprs . . . 26
2.5 Concept requirements . . . 27
2.6 Class specifier in concepts . . . 27

3 Requires clauses and concept maps **29**
3.1 Concept maps . . . 29
Syntax . . . 29
Non-trivial mappings . . . 29
Concept maps affecting visibility (API of a concept map) . . 30
Discussion . . . 31
Termination of concept map queries . . . 32
3.2 Requires clauses . . . 33
Syntax . . . 33
Requirements in the template clause . . . 34
Mixing checked and unchecked parameters . . . 35
Scope under a requires clause . . . 36

4 Axioms **37**
4.1 Axiom syntax . . . 37
Axioms as requirements . . . 37
Axioms as guarantees . . . 38
Discussion . . . 38
4.2 Checking satisfaction of concepts containing axioms . . . 39
Syntactical equality . . . 40
Based on the enclosing concept . . . 40
Discussion . . . 41
4.3 auto concepts . . . 42

5 Concept composition **45**
5.1 And and union . . . 45
5.2 And and disjoint union . . . 45
5.3 Or and intersection . . . 46
5.4 Xor and intersection . . . 48
5.5 Or as syntactic sugar . . . 48
5.6 Xor as syntactic sugar . . . 50
5.7 Not . . . 50
5.8 Implication . . . 51
5.9 Discussion . . . 53

6 Templated requirements and concept maps **55**

6.1 Checking templated concept maps 55
Fuzzy matching 55
Exact matching 56
Discussion 57
6.2 Templated concept maps and unconstrained templates . . . 57
6.3 Specializing constrained templates 59
6.4 Syntax of templated requirements 59
Declaration-like syntax 59
Instantiation-like syntax 60
Discussion 60
6.5 An explanation-only forall keyword 61
6.6 API of templated concept requirements 61
No name modification 62
Unconditional name modification 62
Name modification for functions only 63
6.7 Translating forall clauses back into template clauses 64
Functions 64
Types depending on all parameters 64
Types depending on only some parameters 65
6.8 Unconstrained template requirements 66

7 Same-type constraints **69**
7.1 Arbitrary type constraints 70
7.2 Limiting expressiveness 70
7.3 Discussion 73

8 Explicit and implicit concepts **75**
8.1 Explicit concepts 75
8.2 Implicit concepts 76
8.3 Implicit concepts with optional warnings 77
8.4 Discussion 78

9 Concept-based modules **81**
9.1 Multiple levels of visibility 83
9.2 Extension A: import of module source 85
9.3 Extension B: avoiding unnecessary rebuilds 89
9.4 Discussion 90

10 Inner requirements **91**
10.1 Discussion 96
10.2 Interaction with constexpr constraints 97

11 Concept-based overloading **99**
11.1 Approaches 99
Weaker requirements 99
Using Not as a concept combinator 100
Discussion 101
11.2 Guideline for concept-based overloading 102
11.3 Concept-based overloading for types 103
11.4 Ambiguous calls 104
Comparison with conventional overload resolution 106
11.5 Ambiguities as lack of a meet 107
11.6 Discussion 109
11.7 Concept-based specialization 110
Advantages 110
Simulating concept-based specialization 111
Considering new specializations 112
Disregarding new specializations 113
Interaction with modules 114
Interaction with inner requirements 115

12 Implementing concepts **117**
12.1 Flattening concepts 117
12.2 Converting concept requirements into template parameters . 121
12.3 Transforming constrained templates into unconstrained ones 122
Avoiding duplicates 125
Handling concept-based specialization 126

13 Additional features **129**
13.1 Existential quantifiers on types 129
Extended variant 130
Refinements 130
Discussion 131
13.2 Rvalue references and overload resolution 131
Discussion 132
13.3 Passing overloaded functions to templates 134
Discussion 135
13.4 Loops of function definitions 136
The problem 136
A solution 137
An escape hatch 138
Relationship with IDE support for concepts 139
13.5 Properties 139

13.6 Non-definable properties 141
13.7 Integrating axioms and properties 142
13.8 Integrating preconditions in requires clauses 142

14 Conclusion 145
14.1 Usability of explicit concepts 145
Who has the duty to write concept maps 145
Need to expose constraints used in the implementation . . . 146
Concept-based specialization 148
14.2 Evaluation of the concept design 149
A system as flexible as current templates 149
Enable better checking of template definitions 150
Enable better checking of template uses 150
Better error messages 150
Selection of template specialization based on attributes of template arguments 150
Typical code performs equivalent to existing template code . 151
Simple to implement in current compilers 151
Compatibility with current syntax and semantics 151
Separate compilation of template and template use 152
Simple/terse expression of constrains 153
Express constraints in terms of other constraints 153
Constraints of combinations of template arguments 153
Express semantics/invariants of concept models 153
The extensions shouldn't hinder other language improvements 154
14.3 Limitations of the concept design 154
14.4 Comparison with other concept designs 156
14.5 Future work 158
Feedback and discussion 158
Theoretical analysis 158
Implementation 158
Formalization of the standard library 158

A Other features 159
A.1 Uniform member syntax 159
Discussion 159
A.2 Late check 160
A.3 Substitution 160
Discussion 161
A.4 Methods obtained as replacements from functions 161
A.5 Implicitly-generated constraints for argument types 162

A.6 Delete syntax 163
A.7 Variadic concepts 164
A.8 Concept definitions in namespaces 164
A.9 Concept maps in namespaces 165
A.10 Other constraints 165
Namespaces 165
Friend declarations 166
Attributes 166
Enums 166
Global variables and public fields 166
Specialization 168
Templated axioms 168

B STL concepts 169
B.1 Basic operations 169
Callable 169
UnaryPredicate 169
BinaryPredicate 169
UnaryOperation 170
BinaryOperation 170
DefaultConstructible 170
Destructible 170
Swappable 170
MoveConstructible 171
CopyConstructible 171
ConvertibleTo 171
AssignableTo 172
MoveAssignable 172
CopyAssignable 172
ValueSwappable 172
Incrementable 173
Decrementable 173
SemiRegular 173
Regular 173
B.2 Ordering relations 174
Equivalence 174
EqualityComparable 174
StrictWeakOrder 175
LessThanComparable 175
TotalOrder 176
Comparable 176

B.3 Algebraic concepts 176
Functionalization 176
Commutative 177
Associative 177
Monoid 177
Group 178
CommutativeGroup 178
Ring 178
CommutativeRing 178
IntegralDomain 178
OrderedIntegralDomain 179
Field 179
Arithmetic 179
StrictArithmetic 181
Integer 182
UnsignedInteger 183
StrictInteger 183
B.4 Iterators 184
NullablePointer 184
Iterator 185
PointerReference 185
InputIterator 186
MutablePointerReference 186
OutputIterator 187
ForwardIterator 187
MutableForwardIterator 188
BidirectionalIterator 188
MutableBidirectionalIterator 188
RandomAccessIterator 189
MutableRandomAccessIterator 190
StrictPointerReference 190
B.5 Containers 191
Container 192
ReversibleContainer 194
EmplaceConstructible 194
MoveInsertable 195
CopyInsertable 195
Allocator 195
AllocatorAwareContainer 199
SequenceContainer 200
StrictSequenceContainer 203

FrontAccessSequenceContainer 204
BackAccessSequenceContainer 205
RandomAccessSequenceContainer 205
FrontInsertionSequenceContainer 206
BackInsertionSequenceContainer 206
AssociativeContainer 207
UniqueAssociativeContainer 209
MultipleAssociativeContainer 210
MappingContainer 210
UnorderedAssociativeContainer 210
UniqueUnorderedAssociativeContainer 215
MultipleUnorderedAssociativeContainer 215
UnorderedMappingContainer 216
UnorderedSetContainer 216
B.6 Misc concepts 217
BitmaskType 217
Hash 218
UnaryTypeTrait 218
BinaryTypeTrait 218
TransformationTrait 218
Ratio 219
Clock 219
TrivialClock 219
CharTraits 220
SeedSequence 222
UniformRandomNumberGenerator 223
RandomNumberEngine 223
RandomNumberEngineAdaptor 224
RandomNumberDistribution 224
BasicLockable 225
Lockable 225
TimedLockable 226
Mutex 226
TimedMutex 226

Bibliography **227**

1. Introduction

Several papers related to concepts in C++ have been published in the last decade. This thesis builds on the existing concept literature (especially [Str-03], [SGJ+-05] and [SR-05]), comparing alternative features and discussing known features, new variants and new features altogether.

The main new features that we discuss are inner requirements (chapter 10), the integration of concepts with modules (chapter 9), a compromise between implicit and explicit concepts that allows each project to choose its own policy (chapter 8) and a semantic for same-type constraints with interesting properties (chapter 7).

We thoroughly analyze concept-based overloading (chapter 11), giving guidelines for concept-based overloading of functions (section 11.2) and finding a serious issue of concept-based overloading of types (section 11.3). We also present an improved concept-based specialization that doesn't break separate typechecking (section 11.7).

1.1 Concept literature

Two research groups have been focusing on C++ concepts since 2003: one at Texas A&M University (including B. Stroustrup and G. Dos Reis) and one at Indiana University (including A. Lumsdaine, J. Järvi, R. Garcia, J. Willcock and J. G. Siek). They first worked separately, publishing several papers, and then joined together. The collaboration resulted in [GS-06], a compromise design.

The concept proposal was merged in the C++ standard draft in 2008, but it was removed in 2009 because concepts were considered to make C++ harder to learn, and the only prototype compiler for concept-enabled C++ (a fork of GCC named ConceptGCC) was very slow.

See [Sie-12] for more details on the history of concept papers for C++.

A comparison of language support for generic programming in several popular programming languages (including functional languages like ML, OCaml and Haskell) can be found in [GJL+-07].

1.2 Concepts

We'll use the same terminology as in [SS-11]; specifically, we call *syntactic constraint* a syntactic property of a set of types and operations; we call *axioms* the semantic properties, and we call *concept* a collection of syntactic constraints and axioms.

We call *entity* any declaration that can be specified in a concept. The simpler entities are types and operations, but later on we'll discuss other ones (global variables, fields and namespaces). Note the relationship between an entity and a constraint: an entity is defined in (i.e. required by) a constraint.

For example, requiring the existence of a `+` operator that takes an `std::string` and a `char` and returns an `std::string` is a syntactic constraint. `char`, `std::string` and `+` are entities. The requirement that `(s + c).size() == s.size() + 1`, for every value `s` of type `std::string` and every value `c` of type `char`, is an axiom.

When defining concepts, we often don't write the above properties on specific types; we instead write a parameterized concept, for example:

```
concept String<string, C> {
    ...
}
```

and we use the generic types for the syntactic constraints and the axioms. In this way, we can reuse the same requirements for multiple implementations.

This allows us to state that `String<std::string, char>` holds, but also, for example, `String<my_string, char>`. Even more generally, we could require `String<std::basic_string<T>, T>`, for every type `T`. These satisfaction statements are called concept maps.

The same `String` concept can be instantiated with any user-written or third-party string class that has the same syntax and semantics.

1.3 Roles of concepts

A concept can have different roles in different parts of the code, and also depending on the semantics of concepts that's being used. In this section we try to make these roles explicit, so that we can discuss the concept features from the point of view of each of this roles.

Concept as a set of requirements

This is the role that many concept papers focus on: a concept is a predicate (a set of requirements) on template parameters. This role of concepts has

been in the concept literature from the earliest papers (see [SL-00, Str-03]). This role is important for checking the instantiation of a constrained template. For example:

```
template <typename RandomAccessIterator>
void sort(RandomAccessIterator first,
          RandomAccessIterator last);
...
void do_work() {
    ...
    sort(v.begin(), v.end()); // Ok?
}
```

This role can already be emulated in the current C++ language, using `enable_if` together with traits (for semantic properties) and exploiting SFINAE (for syntactic properties). Even though this can already be done in current C++, including concepts as a language feature can greatly simplify the definition and use of such requirements, that currently require advanced knowledge of C++ templates.

Concept as an API

Another role of concepts is for typechecking the constrained template at the point of definition, allowing errors to be detected before the instantiation of the constrained template.

This is closely related to the notion of archetype found in the earliest concept papers (see [SL-00]) but has received less attention in later papers. [SGJ+-05] is one of the few papers that describes and gives importance to this role.

From this point of view, the `requires` clause provides an API that can be used in the constrained template. For example:

```
template <typename T>
void f(T);

template <typename RandomAccessIterator>
void sort(RandomAccessIterator first,
          RandomAccessIterator last) {
    ...
    typedef Itr::value_type vt; // Ok?
    ...
    ++first; // Ok?
    ...
    f(*first); // Ok?
}
```

The API must be expressive enough to specify types and operations, and also semantic properties (for example, guaranteeing that `first != last` can be replaced with `last != first` without changing the behavior).

Moreover, a concept should be able to guarantee that some concept instantiations hold (for example, above we have to check whether or not `RandomAccessIterator<Itr>` guarantees `Regular<Itr::value_type>`).

In this role, a concept is a contract between the implementation of the constrained template and the client code.

This role can also be partially emulated in current C++ using type traits, but in a different way than before: instead of checking syntactic properties using metaprogramming, we now have to use type traits as `namespace`-like classes that contain wrappers of all types and operations of the API.

When using traits for requirements only, the client code doesn't have to know about them (the template parameters are the same that would be used if the template was unconstrained), while for emulating this feature the template parameters are the traits, and for each call the client code has to specify the types of the trait classes to use.

Concept as an implementation

Let's focus on the syntactical part of the previous role. The syntactical part of the API is an equivalent in static polymorphism of the interfaces used in dynamic polymorphism.

In dynamic polymorphism it's often useful to provide part of the implementation together with an interface, producing an abstract class. In the same way, it can be useful to provide implementation code in a concept.

This allows to write code once in the concept, rather than duplicating it in all classes that model the concept. For example, a `Comparable` concept that provides all relational operators can provide implementations for `<=`, `>=`, `!=` and `>` in terms of `==` and `<`.

As for implementations in abstract classes, in some cases the implementation is provided just for convenience (and a different one may be provided by a specific instance), while in other cases we want to guarantee that specific behavior, preventing changes (as `final` does for virtual methods).

We consider this a different role from the previous one, because we can only decide which implementation should be used at instantiation time, while the API provided by the concept must be independent from the uses, so that we can typecheck the constrained code in isolation from specific instantiations.

Also, the implementation part can be seen as the "glue" between the requirements and the API: by containing implementation code, a concept may provide a richer API without increasing the requirements.

Discussion

In several papers there's a strong focus on the first role, but the use of concepts in the other two roles is not analyzed in detail.

We consider each of these roles very important, and in the rest of this paper we will analyze the various features from all three points of view.

To avoid confusion, we'll use the word "concept" for a description of a scope, "requirements" or "predicate" for the first role, "API", "signature" or "interface" for the second, and "definitions in a concept" for the third.

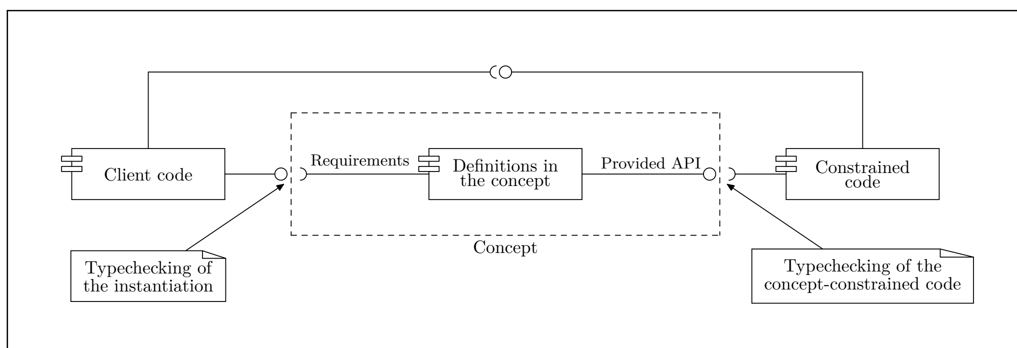
In many concept proposals, a concept is a description of template arguments, and our interpretation is a generalization of that idea. Instead of thinking, for example, `Convertible<T, U>` as a property of the two types `T` and `U`, we think of it as requiring that there is a conversion from `T` to `U` in the current scope (first role) and providing such conversion in the API (second role). `Convertible` does not contain implementation code, so in this case we don't need to generate any glue code (third role).

This is semantically equivalent when only types are involved, but there are things that can be expressed with this interpretation and not with the former — for example, the following concept:

```
concept UniversalHasher<f> {
    template <typename T>
    size_t f(const T&);
}
```

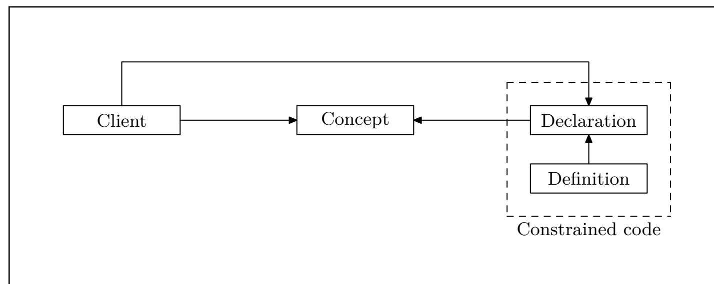
specifies that a template function `f` is able to compute a hash for every type. Note that such a function can't be used as a template argument, because `f` is neither a type nor a value.

We think that this interpretation simplifies reasoning about concepts, both for users and for discussing the various alternatives.



The above diagram provides a visualization of a concept as a component, where the provided API is mapped into the requirements by the definitions inside the concept. Typechecking corresponds to checking whether the socket and its lollipop fit together.

This doesn't mean that concepts should be used for any component (replacing classes) — the implementation part of a concept is meant to be used mostly as syntactic sugar (for example, defining `>` in terms of `<`), rather than a place for all the implementation code.



This diagram shows the dependencies between the three components. Note that the client does not depend on the definition, so this model allows separate typechecking of the client code from the definition of the constrained template.

1.4 Uses of concepts

Template parameter checking

The typical use of a concept is to constrain the set of types that can be used as template arguments of a template function or class, based on their properties. For example, in the following code:

```
template <typename T>
requires { C<T>; }
void f(T x);
```

We declare a function `f` for each type `T` such that `C<T>` (where `C` is a concept). In the current C++, there are templates (the "for each" part) but there is no natural way to express the "such that" part.

However, this is only one of the advantages of concepts.

Domain engineering

By writing the specification of an API using concepts (especially axioms), the programmer can easily find out if the set of constructors, methods and functions is enough for that abstraction, or if instead something is missing (in particular, observer methods and functions).

There's a tradeoff here: the more powerful a specification logic we use, the fewer observer methods we need. We think that by using the programming language itself as a specification logic we can get a reasonable set of observers; this also allows existing programmers to easily write and understand axioms. In this way, the programmers will be more motivated to write axioms even for rapidly-changing code, and for new abstractions that haven't been studied as much as the classical mathematical structures.

Documentation

Concepts can also be useful for specifying APIs: by explicitly stating the syntactic and semantic properties of the provided code (and for template libraries, also the required syntactic and semantic properties of the template parameters) both the developers of the library itself and the users of such library can get a better understanding of the behavior.

Also, in this way when the maintainers of the library change these properties, it's more clear that this may break the client code, since there is a clear distinction between the expected behavior (specified using concepts) and the implementation-specific properties (that are not specified).

If concepts are not used, and the developers of the client code read the provided header files in order to know these properties, they won't know which properties will be stable across releases and which ones are instead considered implementation details, so they should not be depended upon.

This will be more and more important as the use of templates increases.

Testing

If the axioms are expressed using a suitable syntax, for example the one provided by the Catsfoot library [Catsfoot website], it's also possible to test them with a subset of the possible parameters (an exhaustive testing with all possible parameters is often impossible or too slow to be feasible).

The axiom syntax that is currently being considered for the next C++ standard at the time of this thesis, [SS-12], does not allow testing.

Optimization

Axioms can also be used by a concept-aware compiler for optimization purposes, by replacing an expression with another when an axiom guarantees that they are equivalent.

The syntax in [SS-12] seems to be designed for this purpose, as it allows to state the equivalence of expressions that have side-effects, while the Catsfoot syntax can't. Also, Catsfoot compares values using `operator==()`, but since it is defined by the user, unless the compiler can prove that it's a congruence (and this is unlikely) replacing an expression with the other may change the program behavior, so it wouldn't be a sound optimization.

1.5 Concepts vs interfaces

Some of the features of concepts can be emulated in most programming languages using interfaces (in C++, classes where all methods are pure virtual) and implementing these interfaces in concrete classes. However, this approach has several limitations compared with concepts.

As with concepts, it is possible to require methods with a specific signature in interfaces, but an interface can only require methods from a single class, while a concept can have requirements on multiple classes, if it takes more than one type template parameter. When using interfaces, the methods often have informal behavior specification, while with concepts it's possible to have axioms in code, which can be useful for various purposes (see the previous section).

In C++, often some non-member function are provided together with a class; concepts allow to specify them, but interfaces don't. The same also applies to static methods and functionality accessed through type traits.

Another advantage of concepts is that by using concept maps it's possible to bind concepts to a class after its definition. This means that if we define a new concept we can add a concept map for a third-party class, as long as we know that it's satisfied. Using interfaces, instead, we would have either to modify the class itself (but this may not be possible) or to create a derived class that also implements the new interface and use the new class instead of the old one (this may also not be possible if the old class appears in the public API of our library).

1.6 Concepts vs programming by contract

As a specification of the syntax and semantic of an API, concepts are akin to programming by contract. However, there are important differences: a concept specifies an API as a whole, including the interaction between different operations, while programming by contract often focuses on a single operation at a time. This is especially apparent for semantic properties: in concepts they are specified as axioms (that can relate to multiple types and operations), while the pre- and post-conditions used in programming by contract are tied to a specific operation (but can often involve other operations, typically observers).

Every post-condition can be described as an axiom instead, but this is not the case for pre-conditions. Translating axioms into pre/post-conditions is in general impossible. This suggests that concepts are not an alternative to programming by contract, but rather a supplement. We think that post-conditions should be expressed as axioms, but it would be useful to have

a way to formalize preconditions in C++. This feature is independent from concepts, and can be integrated with them by extending the syntax of signatures. We describe a possible syntax and semantics inspired by axioms in section 13.5.

1.7 Tools for concepts

Even though concepts aren't included in the C++ standard, some C++ tools related to concepts have already been developed, for various reasons.

A very important tool is the implementation of a concept-enabled C++ compiler: this allows to test the feasibility and performance of concepts in the real world. Incorporating in the C++ standard a concept design which actually is not feasible to implement may lead to different compiler-specific concept dialects.

In 2004, a fork of GCC named ConceptGCC was started, to support C++ with concepts. Unfortunately, the performance was not optimal, and this led to further criticism of C++ concepts in general.

In 2009, work started on a fork of Clang named ConceptClang (see [ConceptClang-09]). This version was last updated in 2011 (see [ConceptClang-11]), but it's still being developed.

A tool called Catsfoot (see [Catsfoot website]) has been developed at the University of Bergen (Norway). It's not a compiler, but a library that tries to implement concepts in current C++. This approach prevents it from constraining the names used in templates, but still allows to check the syntactic constraints of concept maps and to generate tests from the axioms written in concepts.

1.8 Goals of the thesis

Our goal is the definition of a concept design that:

- Allows separate typechecking. This guarantees that error messages will be caught immediately, rather than being delayed at instantiation time. In turn, this is a prerequisite of some of our new features, especially inner requirements (chapter 10) and concept-based modules[1] (chapter 9).

[1] As we'll see in chapter 9, concept-based modules make it easy to specify the API of a library using concepts. This motivates users to write such specifications (that are basically concept maps) and allows integration of libraries using implicit concepts with libraries using explicit ones.

- Maintains backward compatibility: unconstrained code must still work as before and it should be able to interact with constrained code.
- Is done at compile-time: we don't consider solutions based on vtables and similar approaches.
- Is a suitable replacement of unconstrained templates: the existing C++ template tricks (type traits, tag types, SFINAE, `enable_if`, CRTP, etc.) must be replaceable with corresponding constrained code and such code should be "cleaner" and easier to understand than the previous implementation.
- Is concise: writing concept-aware code should not be significantly more verbose than usual C++ code.
- Makes it easy to port code to concepts: a library should be able to switch to concepts incrementally and with small changes. We can't expect template libraries to be rewritten in order to switch to concepts.
- Reduce compilation time: the use of template libraries in current C++ slows down the compilation process considerably. Concept-aware code should compile faster rather than slower.[2]

1.9 Thesis structure

In the next chapters we will discuss various design decisions for concepts, first describing the possible choices, then comparing their pros and cons and finally suggesting the use of one or more of them based on these considerations.

Appendix A contains some minor concept features together with the ones we rejected. Appendix B contains formalizations of the STL concepts using the concept features chosen in the previous chapters.

[2] This is a stated goal of our design and influenced our design decisions (especially regarding concept-based modules). However, the lack of an implementation of our design means that we don't have any quantitative measures yet.

2. Basic constraints

2.1 Operation constraints

There are several possible ways to express that a certain operation exists. In this section we analyze them taking into account the concept roles explained in the previous chapter.

Precise signatures

With this syntax, we write the prototypes of the operations that we require. The syntax is different, but the behavior is similar to the function match approach in [Str-03], except that we compose concepts with And and union rather than And and disjoint union (see chapter 5).

```
concept Number<N> {
    ~N::N();
    N::N(unsigned long);
    N::operator unsigned long();
    N operator+(const N&, const N&);
}
```

A scope satisfies the concept if the corresponding operations have been declared (not necessarily defined). The API of the concept is (trivially) the collection of prototypes declared inside the concept.

Usage patterns

This is a style similar to the one used in some C++ libraries for concepts, for example the Boost Concept Check Library, and also discussed in [Str-03]. The required operations are specified using normal C++ expressions; the concept is satisfied in any scope where the expression is defined.

```
concept Number<N> {
    constraints(const N& x, const N& y) {
        N(42UL);
        unsigned long = x;
        N = x + y;
    }
}
```

When we write `U = e`, with `U` type and `e` expression, we mean that `e` must typecheck and have a type that is convertible to `U`.

Note that `x` and `y` have been declared in the header of the `constraints` block, so that we are not requiring that they are default constructible.

Regarding the provided signature, since the requirement specification doesn't naturally map to a signature, we can't allow the code using this concept to do anything even slightly different from the specified expressions.

For example, knowing that `x + y` is convertible to an `N` doesn't mean that `x + y + z` is a valid expression, because the expression in the constraints may be using a conversion both on the parameters and on the return type, and in `x + y + z` we would have to apply two conversions to the `(x + y)` subexpression, which is not allowed if they involve user-defined conversions.

Usage patterns with functionalization

One possible solution is to generate a pseudo-signature based on the expressions: for example, the constraint `N = x + y` generates the signature `N operator+(const N&, const N&)`. The template code constrained with this concept will call wrapper functions with the generated signatures, and that are implemented in terms of the actual operations provided by the caller's scope, adding conversions if needed.

This introduces a limitation: if `T` and `U` are different types, we can't have both `T = e` and `U = e` in the requirements, because this would generate two functions that differ only in the return type.

For more complex expressions (for example `x * y + z`), we need to introduce unnamed types for temporaries. Such unnamed types can be implemented as implicit existential types (see section 13.1).

Fuzzy signatures

Fuzzy signatures, also known as pseudo-signatures, were first described in [Str-03], even though they originally had a different syntax, and were later adopted in the concept proposal for the C++ standard.

This syntax looks very similar to the one for precise signatures, but the semantics is very similar to usage patterns with functionalization, restricted to simple expressions. Instead of writing the expressions, we now write the pseudo-signature and we let the compiler do the translation to usage patterns when checking if the concept is satisfied in a certain scope.

```
concept Number<N> {
    ~N::N();
    N::N(unsigned long);
    N::operator unsigned long();
    N operator+(const N&, const N&);
}
```

So with the same syntax we now have a weaker predicate: any scope where, for instance, `operator+()` returns a type *convertible* to `N`, is allowed. Such conversions are allowed both on parameters and on return types, as happens for usage patterns.

The API, on the other hand, is the set of the specified signatures, as with precise signatures.

Allowing conversions when checking the requirement imposes the need of wrapper functions that do such conversions. Such implicit wrapper functions can be inlined, to avoid the performance penalty.

As in the generation of pseudo-signatures to compute the API of usage patterns, we can't require the same function twice, with different return type. However, when using this syntax this seems a more natural limitation, since it's a known rule for C++ declarations.

Discussion

Both the fuzzy signatures approach and usage patterns with functionalization seem feasible. It's mainly a matter of syntax, since the compiler needs both the signature (for typechecking constrained templates) and the expressions (to check the constraint), so the other is automatically generated.

The usage patterns approach needs/allows the introduction of unnamed existential types. However, existential types can also be introduced in the fuzzy signatures approach (see section 13.1).

Even though they have the same expressive power, the use of fuzzy signatures is more natural if concepts are thought as high-level abstractions (`Number`, `Vector` and so on), while use patterns are more natural for describing the minimal set of requirements for a specific code fragment. For this reason we favor pseudo-signatures — we think that concepts should describe complete APIs, rather than enumerating the primitives used in the implementation. This allows a more high-level thinking (both for the user and for the implementation) and also to more freely refactor the constrained code without changing the requirements. Of course, "fat" APIs increase the burden on the client code, so the trade-off should be evaluated in each case.

2.2 Type specifications

Another interesting property of a scope is the existence of a type.

For example, for the `Vector` concept, we need to require/provide several inner types: `iterator`, `value_type` and so on.

Simple type specification

We simply write the type that we want to specify. For example:

```
concept Vector<V, T> {
    ...
    class V::iterator;
    class V::value_type;
    ...
}
```

Note that we use the `class` keyword here, but the types may be typedefs. Another possible keyword is `typename`, but we use it below for associated types. If associated types are not proposed for introduction in the standard, we may consider using `typename` for simple types instead, because it behaves quite differently from the `class` keyword in declarations.

Associated types

Associated types have been introduced as a language feature in [Str-03] and since then they have been consistently adopted in C++ concept papers. In some proposals, the types provided by a concept are split in two classes: (normal) types and associated types.

Associated types behave differently with respect to composition of concepts: it's possible for a concept to inherit from another one (and then inheriting all the associated types) or to require another concept, and in this case the associated types are not inherited. To distinguish associated types, we use the `typename` keyword instead of `class`.

Discussion

A disadvantage of associated types is that, even though they are not in the signature when they are hidden, they are still in the requirements, so the compiler must know what type is used in the constraints. This means that either they are always paired to a default type (see the next section) or the user of the composed concept must know about them and specify the corresponding types in the concept map.

They don't seem to be much useful, since types that are relevant to the interface of a class are usually exposed as inner types (or typedefs), for example `value_type` in STL containers.

Nothing prevents us to also attach inner types to builtin types, like `int` and `T*`, as long as we provide a definition in the concept (this is essentially the "Member types" feature from [SR-03]).

In the rest of this paper we will use the `typename` keyword for simple type specifications.

2.3 Default types and operations

Often, a type or operation in a concept can be defined in terms of other entities in the same concept. In this case, it's very convenient to include such definitions in the concept itself. This feature has been proposed in [SGJ+-05]. For example:

```
concept EqualityComparable<T> {
    bool operator==(const T&, const T&);
    bool operator!=(const T& x, const T& y) {
        return !(x == y);
    }
}
```

The semantics of this definition is that `operator!=()` is added to the provided signature, but not to the requirements of the concept.

We think that we should distinguish two cases: either the definition is semantically binding (any other definition provided by the scope must be semantically equivalent) or it's not. By making the definition of `operator!=()` above semantically binding, we allow the code that requires `EqualityComparable` to consider `(x != y)` semantically equivalent to `!(x == y)` (even though it may be more efficient).

We think that this semantics should be the default. To mark the non-semantically-binding definitions we will use the `default` keyword, as in the following:

```
concept Container<C> {
    typename C;
    typename C::value_type;
    default typename C::reference = C::value_type&;
    ...
}
```

So `reference` is `value_type&` if it's not already defined in the class, but if it is, the user-defined type is used instead. So the code that requires `Container` can't assume that `reference = value_type&`.

A non-default operation definition is equivalent to a default one together with an axiom. A non-default type definition, instead, is equivalent to a

default one together with a same-type constraint (see chapter 7). The following concept is equivalent to the one at the beginning of this section:

```
concept EqualityComparable<T> {
    bool operator==(const T&, const T&);
    default bool operator!=(const T& x, const T& y) {
        return !(x == y);
    }
    axiom(const T& x, const T& y) {
        (x != y) <=> !(x == y);
    }
}
```

In other words, a default definition affects the implementation role of the concept (since the type/operation in the API can be mapped to the definition in the concept) and the predicate role (that is weakened by removing the requirement on the type/operation). A non-default definition also changes the API part, adding the semantic equivalence as an axiom or same-type constraint.

`default` type and function definitions don't require the user-provided implementation (if any) to be equivalent, but they do require the in-concept definition to satisfy the other semantic requirements in the concept.

2.4 Constraints on constexprs

As in some concept proposals (e.g., [SGJ+-05]), we will allow constraints based on boolean constexprs, for example:

```
constexpr is_prime(size_t n) { ... }
concept PrimeSized<T> {
    is_prime(sizeof(T));
}
```

Clearly, the predicate part of this constraint involves evaluating the expression and checking that it is `true`. Unlike other constraints, though, no API is exposed, except the guarantee that such expression is `true`. Therefore, even though this constraint is able to express the predicate part of many other kinds of constraints (for example, using type traits), it behaves very differently in the API role.

We don't attempt to prove equivalences between expressions, we'll use a purely syntactic matching. For example, even though `false || bool(a)` always has the same value as `bool(a)`, the matching will not take this into account.

The matching can be improved to some degree, but in general we can't always detect that two expressions containing unknown values and functions

are equivalent. The choice of whether we should improve the matching, and to what extent, is left to future work. We will only introduce a slight improvement together with the feature of inner requirements, see 10.2.

2.5 Concept requirements

As functions allow to reuse the same code in many situations by calling a common function, it's natural to define complex concepts by composing simpler ones. This feature is fundamental for concepts and has been present since the earliest concept papers (see for example [Str-03, SR-03b]).

We'll use the following syntax for concept requirements:

```
concept A<T, U> {
    B<T>; // A<T, U> refines B<T>.
    ...
}
```

In the C++ literature, concept requirements are usually preceded by the `requires` keyword. We prefer the syntax above because it's more concise, it's similar to our concept map syntax (see section 3.1) and allows us to use the `requires` keyword for constrained concept requirements instead (see section 5.8). Also, this allows to more easily think of a concept as an interface (that is, to focus on the provided signature) when needed, instead of always thinking it as a predicate (the requirements on the scope).

We don't distinguish concept refinements from concept requirements. This has been traditionally done in the C++ literature, but [GS-06] already suggested that their unification should be investigated.

2.6 Class specifier in concepts

Often, a concept describes many types and operations provided by a class. Such class may be a template with its own `requires` clause, and it's very verbose to repeat both the `template` clause and the requirements for each method. This feature allows `class` blocks inside concepts, as a shorthand for the more verbose notation. For example, this code:

```
concept StackTemplate<stack> {
    template <Regular T>
    class stack {
        stack();
        void push(const T&);
        const T& top() const;
        void pop();
    }
}
```

Will be expanded to:

```
concept StackTemplate<stack> {
    template <Regular T>
    typename stack<T>;

    template <Regular T>
    stack<T>::stack();

    template <Regular T>
    void stack<T>::push(const T&);

    template <Regular T>
    const T& stack<T>::top() const;

    template <Regular T>
    void stack<T>::pop();
}
```

Inside a `class` pseudo-definition, we can use the full concept syntax that we can use outside, except namespace definitions (for obvious reasons). All such declarations will be interpreted as declarations inside the class, except declarations from required concepts, that are unaffected. The only advantages of putting a required concept inside a class pseudo-definition is that to instantiate it, we can omit the name of the class as qualifier for the entities that we have already specified, and we also get the enclosing `template` and `requires` clause as usual. So the following:

```
concept A<T> {
    void f(T);
}
concept IFoo<Foo> {
    template <Regular T>
    class Foo {
        typename value_type;
        A<value_type>;
    }
}
```

is equivalent to:

```
concept IFoo<Foo> {
    template <Regular T>
    class Foo {
        typename value_type;
    }
    template <Regular T>
    void f(Foo<T>::value_type);
}
```

3. Requires clauses and concept maps

3.1 Concept maps

Concept maps allow the programmer to state that a concept is satisfied in the current scope. They allow the compiler to assume the semantic properties stated in the concept, and they produce a compilation error if some of the syntactic properties are not satisfied.

Syntax

We'll omit the `concept_map` keyword used in the C++ literature to declare concept maps. For example:

```
concept CopyConstructible<T> {
    ...
}

CopyConstructible<int>;

template <CopyConstructible T>
CopyConstructible<std::vector<T>>;
```

Specifies that `int` is `CopyConstructible` and that, for every `T` that is `CopyConstructible`, `std::vector<T>` also is.

We use the same syntax for concept maps and concept requirements, and this is not a coincidence: both are specifying the same property — the only difference is that a concept map specifies that such property holds in the current scope, while a concept requirement specifies that such property holds for all models of the concept.

Non-trivial mappings

In most proposals (see for example [SGJ+-05]), a concept map can have a body containing the definition of some types and operations required by the concept that aren't available in the current scope.

For example:

```
typedef double my_type;

void do_stuff(int);

concept C<T> {
    typename myType;
    void doStuff(T);
}

C<int> {
    typename myType = my_type;
    void doStuff(int x) {
        do_stuff(x);
    }
}
```

Concept maps affecting visibility (API of a concept map)

We may decide that a concept map, in addition to the behavior described above (it is used to prove concept map queries and to check the syntactic requirements), also has a third feature: it extends the current scope with the types and operations defined in the concept. For example:

```
concept EqualityComparable<T> {
    typename T;
    bool operator==(const T&, const T&);
    bool operator!=(const T& x, const T& y) {
        return !(x == y);
    }
    ... // Axioms
}

struct point {
    int x, y;
};
bool operator==(const point& p, const point& q) {
    return p.x == q.x && p.y == q.y;
}

EqualityComparable<point>;

void do_stuff(const point& x) {
    if (x != point{0, 0}) // Allowed.
        ...
}
```

With this feature, `operator!=()` is visible in `do_stuff()`, otherwise it would have been visible only in functions that require `EqualityComparable` in their `requires` clauses.

This also allows to “add” inner types and methods to a class after its definition (note that such code can’t access the private and protected parts of the class, so this doesn’t break encapsulation). As always with concepts, we forbid adding virtual methods (since we can’t modify the vtable), and adding fields (since we can’t modify the size of the objects of that type). It’s not allowed to remove or change the behavior of existing methods, but the behavior of a function call can still change because one of the additional overloads may be chosen instead of an original one.

Discussion

We think that concept maps should affect visibility. In this way, the boilerplate methods of a class can be written just once, in a concept that describes that kind of class.

Since the user is motivated to write a concept map in order to factor out boilerplate code, we think that this will encourage users to write them for production code (even non-templated code), getting the specification of some semantic properties of the class for free.

We think that this is an important point: even though we don’t think that concept maps should be compulsory (see chapter 8), they are a very powerful tool, and we want to encourage users to write them as much as possible by making them even more powerful. In chapter 9 we’ll see how to integrate them with modules.

Having concept maps that affect visibility, non-trivial mappings in concept maps lose much of their importance, since most mappings can just be moved before the concept map, as usual definitions, and others can be specified by writing an extended concept that contains them (as well as requiring the original concept) and modifying the concept map to point to the extended concept.

In some concept proposals (e.g. [SGJ+-05]) a concept map can have a body and the concept instantiation can be used as a qualifier, for scope resolution. This feature is not so useful when concept maps affect visibility, since we can just move such definitions before the concept map, so that the qualifier isn’t needed anymore.

Multiple concept maps for the same concept instantiation and with different implementations are not allowed in this case, since we would get several incompatible type and function definitions in the same scope. In such cases, we prefer to distinguish the two concept instantiations by adding parameters when we want to express a satisfaction statement (for example, see the algebraic concepts in section B.3) and to use the feature of inner

requirements (see chapter 10) when the concept map was used for syntax adaptation.

If the feature of concept maps affecting visibility is used, we probably want to forbid the definition of types and operations in `auto` concepts. Note that non-`default` operation definitions should be forbidden anyway, since they are just syntactic sugar for a `default` operation definition together with an axiom.

If we both allowed concept maps affecting visibility and definitions inside such concepts, defining an `auto` concept would add those defined entities to the current scope, and we think that in this case the user should be required to make this explicit with a concept map.

Termination of concept map queries

The search of concept maps won't terminate if we don't add additional limitations. For example:

```
concept A<T> {
    void f(const T&);
}
concept B<T> {
    void g(const T&);
}

template <typename T>
requires { B<T>; }
A<T> {
    void f(const T& x) {
        g(x);
    }
}
template <typename T>
requires { A<T>; }
B<T> {
    void g(const T& x) {
        f(x);
    }
}

template <typename T>
requires { A<T>; }
void do_stuff_helper(const T& x) {
    f(x); // Ok.
}
void do_stuff() {
    do_stuff_helper(42); // Allowed?
}
```

Here, a naïve algorithm for checking would try proving `A<T>`, find the first concept map, recursively try to prove `B<T>`, then `A<T>` again, and so on.

This is similar to what can happen with recursive template instantiations, even though it can also happen in cases like the above, that couldn't be expressed in C++ without concepts. So, we could use the same solution that is used to limit template instantiations: an arbitrary limit to the recursion (say, 1024), that makes the search fail when reached.

3.2 Requires clauses

A `requires` clause is usually preceded by a `template` clause (but not necessarily) and specifies the conditions under which the following is defined. This formalizes the purpose of `enable_if`. Such conditions describe the needed properties of a scope, so it's natural to use a concept to describe them.

Syntax

```
template <typename T>
requires { Regular<T>; }
class vector {
    ...
};
```

In most concept papers for C++, only concept instantiations are allowed. Being able to use the full syntax available in a concept body allows to distinguish syntactic requirements (expressed as types, operation signatures or syntactic-only concepts) and requirements that include semantic properties (expressed as concepts).

We don't allow axioms in requires clauses, because it would be impossible to provide concept maps for such axioms. If axioms were needed, the user can still write a (named) concept with the relevant requirements and axioms, and then use it in the requires clause. We think that in most cases that concept will be reused for multiple functions, so the rewriting as a concept will actually simplify the code.

Because of these design choices, we prefer a braced notation for the requires clause, with each requirement followed by a semicolon. This is slightly more verbose for very simple cases, but we think that it improves readability for more complex ones, and allows to write more complex requirements (templated requirements, requirements with their own requires clauses, etc.).

Compare:

```
template <typename T>
requires Regular<T>
void f();

template <typename Itr, typename OItr>
requires InputIterator<Itr> && Regular<Itr::value_type>
        && OutputIterator<OItr>
        && Convertible<Itr::value_type, OItr::value_type>;
void copy(Itr first, Itr last, OItr dest);
```

with:

```
template <typename T>
requires { Regular<T>; }
void f();

template <typename Itr, typename OItr>
requires {
    InputIterator<Itr>;
    Regular<Itr::value_type>;
    OutputIterator<OItr>;
    Convertible<Itr::value_type, OItr::value_type>;
}
void copy(Itr first, Itr last, OItr dest);
```

Requirements in the template clause

Often, we have to constrain the parameters of a template separately, with a unary concept. The following syntax has already been proposed in many concept papers, for example [SR-03].

```
template <Regular T>
void f();
```

Is equivalent to:

```
template <typename T>
requires { Regular<T>; }
void f();
```

Note that even though the `requires` clause can be omitted in some cases, the template is still considered a constrained template — this is only syntactic sugar.

The above syntax is often extended to concepts taking more than one parameter, as in the following:

```
template <Regular T, Collection<T> C>
void f();
```

which is equivalent to:

```
template <typename T, typename C>
requires {
    Regular<T>;
    Collection<C, T>;
}
void f();
```

Mixing checked and unchecked parameters

Some concept papers (for example [SGJ+-05]) allow a template to have a mix of checked and unchecked template parameters, so that an unconstrained function can be only partially constrained.

If at least one of the parameters is not constrained, then the whole template is considered unconstrained and can't be typechecked separately from the instantiation. The amount of typechecking at the point of definition increases, but it's not enough to guarantee separate typechecking.

The C++ literature, especially in the earlier papers, considered concepts as properties of single types, but since we want to fully typecheck the constrained template with respect to its requires clause, we prefer to consider the API provided by the requirements as a whole.

[SR-05] shows the following example:

```
template <C T1, typename T2>
void f(T1 t1, T2 t2) {
    ++t1;
    ++t2;
    t1 + t2;
}
```

Note that here `T2` is meant to be an unconstrained parameter, while `C` is constrained.

Lacking concept requirements on `T2`, we can check the first statement but not the other two, since we have no information on `T2` and resolving the `+` is impossible, because even if there is an available definition of `+` that can be used for `t1`, it may not be unique and we have no way of picking the best candidate, or even to check that any of them can be used wih `t2`.

Combining the arguments together is not an esoteric operation, we can expect it to happen often (and early) in partially constrained functions, therefore limiting the amount of additional typechecking compared to an unconstrained template.

We don't think that this feature is very useful, and prefer using unconstrained templates in the cases when one or more parameters must be unconstrained.

Scope under a requires clause

The template arguments (if any) and the names specified by the requires clause should of course be in scope. But we also need to allow the use of declarations (and concept maps) of the enclosing scope, so that a constrained function provided by a library may be implemented using a lower-level library that the end user does not need to care about (i.e. that doesn't appear in the requires clause).

```
// sort.h
#include <list>

template <Regular T>
void sort_list(std::list<T>& l);

#include "sort.cpp"
```

```
// sort.cpp
#include <vector>
#include <algorithm>

template <Regular T>
void sort_list(std::list<T>& l) {
    std::vector<T> v(l.begin(), l.end());
    std::sort(v.begin(), v.end());
    std::copy(v.begin(), v.end(), l.begin());
}
```

In this example the user can see `std::list` in the function prototype, but doesn't know that `std::vector`, `std::sort` and `std::copy` are used. None of them has to appear in the `requires` clause, because they were defined in the enclosing scope of the definition. If the preprocessor is used to include the library, the implementation code must be included together with the declaration part, so this distinction is blurred. It's much more important when using modules (as we'll see in chapter 9), since a module can then expose only the declaration part and completely hide the implementation, so that the client code will have to include or import `vector` again, if needed.

4. Axioms

Most of this thesis is focused on describing syntactic properties of a scope in a concept. However, semantic ones are also very important. Axioms have been part of the C++ concepts' literature since [SR-03] (even though they weren't called "axioms" yet) and have been discussed in several papers, notably [RSM-09].

4.1 Axiom syntax

Axioms as requirements

This kind of axioms expresses properties in the form of an assertion on a boolean value. For a more detailed analysis, see [BDH-09].

For example, in this style we can write the axioms for an equivalence relation:

```
concept EqualityComparable<T> {
    bool operator==(const T&, const T&);
    axiom reflexivity(const T& x) {
        x == x;
    }
    axiom symmetry(const T& x, const T& y) {
        if (x == y)
            y == x;
    }
    axiom transitivity(const T& x, const T& y, const T& z) {
        if (x == y && y == z)
            x == z;
    }
}
```

We say that an axiom *holds* if any possible "call" of the axiom with valid values of the specified types succeeds. The name of the axiom is for documentation purposes only and it's ignored. It is allowed to have unnamed axioms, or several axioms with the same name.

Given a specific type `T` and a finite set of values of type `T`, the above axioms can be tested. This approach for testing allows to split the tests' code in axioms (expressed once in a concept and shared by all types that

implement the concept) and test data (expressed in a type-specific file as usual unit tests).

Such axiom-based testing can be already done in C++11 by using the Catsfoot library, see [BDH-11] for details.

Axioms as guarantees

This kind of axioms expresses guarantees as equivalences of expressions. Stating that two expressions are equivalent means that they can replaced with each other without changing the behavior of a program. This is very different from stating that the values are equal — the expressions may have side effects, or the user-defined equality may be too strong or weak.

```
concept RandomAccessIterator<iterator> {
    ...
    axiom(iterator& itr) {
        ++itr <=> itr += 1;
    }
}
```

We use the `<=>` operator to express the equivalence of expressions.

Note that this is *not* a generalization of the syntax of the previous paragraph — for example, the following concept:

```
concept EqualityComparable<T> {
    bool operator==(const T&, const T&);
    axiom reflexivity(const T& x) {
        (x == x) <=> true;
    }
    axiom symmetry(const T& x, const T& y) {
        if (x == y)
            (y == x) <=> true;
    }
    axiom transitivity(const T& x, const T& y, const T& z) {
        if (x == y && y == z)
            (x == z) <=> true;
    }
}
```

is not equivalent to the one we saw earlier: here we forbid side effects (for example: writing to a log file) that the original concept does not specify.

It's obvious that the reverse transformation is also not possible: we can constrain side effects with this syntax, but not with the previous one.

Discussion

The two kinds of axioms express different properties and both of them can be useful for documenting the intended behavior.

The first kind of axioms can also be used for testing, together with a data set or using other means to generate test data. The second kind can be used by programmers for refactoring code and also by language tools for various purposes (optimization, code analysis, etc.).

We think that both kinds of axioms should be allowed, and we agree with [RSM-09] that directing optimizers should not be the main goal of the axiom syntax (this would mean defining asymmetric axioms that would be used as user-defined optimization rules).

Using `==` in both cases, while simpler from the user's viewpoint, leads to unexpected specifications that will result in wrong optimizations for compilers that do axiom-directed optimization. Consider this concept:

```
concept CopyConstructible<T> {
    ...
    axiom (const T& x) {
        T(x) == x;
    }
}
```

This is a reasonable relationship between copy construction and equality, but replacing `T(x)` with `x` (or, even worse, the opposite) can change the program's behavior even for the most regular types. Consider what happens for `vectors` when changing `T(x).capacity()` into `x.capacity()`, or the other way around.

We will implicitly generate the second kind of axioms for non-`default` operation definitions in a concept, so that it's allowed to replace equivalent code fragments during refactoring.

4.2 Checking satisfaction of concepts containing axioms

Due to the undecidability of many semantic-related problems (checking if an axiom is satisfied, checking if two axioms are equivalent, etc.), we can't check semantic assumptions, unless an encoding of correctness proofs is proposed.

Since most programmers are not accustomed with writing proofs, we don't think that this is generally useful, even though this feature could be useful in domains where code correctness is extremely important.

So, how do we check whether an axiom is satisfied? [1]

[1] In this section, we will use the concept semantic known as "explicit concepts" in the C++ literature. The issue of explicit vs implicit concepts is analyzed in chapter 8.

Syntactical equality

This approach consists in comparing the axiom that we are currently checking with the ones known to hold (due to concept maps). If they are syntactically equal, then we are sure that the axiom is satisfied.

With this approach, when different concepts contain the exact same axioms the compiler will consider the two concepts semantically equivalent.

The predicate part of concept is the logical conjunction of the axioms and the predicate of the constraints.

With the following concept:

```
concept Fooable<T> {
    typename T;
    void foo(T, T);
    axiom (T x, T y) {
        foo(x, y) <=> foo(y, x);
    }
}
```

We get a predicate as follows:

$$\begin{aligned}\text{predicate}(\texttt{Fooable<T>}) &= \text{predicate}\begin{pmatrix}\texttt{typename T;}\\ \texttt{void foo(T, T);}\end{pmatrix}\\ &\quad\wedge \begin{pmatrix}\texttt{axiom (T x, T y) \{}\\ \quad\quad\texttt{foo(x, y) <=> foo(y, x);}\\ \texttt{\}}\end{pmatrix}\end{aligned}$$

The axiom is considered as an atomic predicate, that can only be proven from an identical one.

Based on the enclosing concept

This approach identifies axioms based on the concept that contains them. When we check whether a concept instantiation (say C<int>) is satisfied, we require a concept map for this instantiation to be in scope.

Unlike the previous approach, if we define the same concept twice, say as C and D, unless we use templated concept maps to state their equivalence, a concept map for C<int> will not match a requirement for D<int>.

With this interpretation, the predicate part of a concept is the conjunction of the requirements of the individual requirements (excluding axioms) together with a atomic property that identifies the specific concept instantiation. The predicate in the last example becomes:

$$\begin{aligned}\text{predicate}(\texttt{Fooable<T>}) &= \text{predicate}\begin{pmatrix}\texttt{typename T;}\\ \texttt{void foo(T, T);}\end{pmatrix}\\ &\quad\wedge\ (\texttt{Fooable<T>})\end{aligned}$$

Here the concept instantiation is used as atomic predicate.

Discussion

The first approach is a closer approximation of the ideal semantic (checking axiom equivalence), but it has several shortcomings.

Firstly, the way in which an axiom is written is significant, so if the formulation of an axiom in a concept is changed and there is any code relying on the equivalence of that axiom with one in another concept, such code will break. For example, splitting the concept `Fooable` above into the following two concepts:

```
concept Commutative<f, T> {
    typename T;
    void f(T, T);
    axiom (T x, T y) {
        f(y, x) <=> f(x, y); // Note the different order.
    }
}
concept Fooable<T> {
    Commutative<foo, T>;
}
```

can break the code using `Fooable` when using the first approach, but not when using the second one.

Another downside of the first approach is that it only handles formalized axioms, disregarding semantic properties expressed in comments. It's not reasonable to also compare comments, because comments should be ignored by the compiler. The second approach, even though comments aren't actually processed, never assumes that two concepts are the same just because they have the same axioms, so comments are treated in the same way as axioms are.

Additionally, by checking concept atomically we also save work during the satisfaction check: we only consider the concept maps for the current concept and for concepts that require it — we don't have to break the current concept into basic components and check them separately.

```
concept C<T> {
    ...
}
concept D<T> {
    C<T>;
    ...
}
D<int>;

template <C T>
void f(T);
```

```
f(1);   // Ok.
f(1.0); // Error.
```

For both calls, we first determine the concepts that require `C` (in this case, `C` and `D`) and then consider only the concept maps for these concepts. We only have to process the body of `D` for determining that it requires `C`, and we don't have to process the body of `C` at all.

For these reasons, we choose the second approach.

4.3 auto concepts

Sometimes, a concept contains only syntactic requirements and concept requirements, with no additional semantic properties.

This doesn't happen very often: most of the time, even if there are no apparent semantic properties, some comments may describe a specific behavior, that the compiler doesn't know about. For example, an `Assignable` concept may only seem to require `operator=()`, but it's expected that after the assignment the LHS will have a value "equivalent" to the one of the RHS.

When there are really no additional semantic requirements we can let the compiler know that this is the case, and we suggest to use the `auto` keyword for this purpose. We think that we need a keyword (and not just a check that there are no axioms in the concept) because some axioms may not be expressible, so a concept may have semantic requirements in comments, even if it has no axioms.

Basically, a concept marked `auto` is an abbreviation for its contents. So, while the concept itself can't add axioms (nor any other semantic constraints), it can require concepts that do. Also, this means that, even though the assumption about the concept itself is considered safe and doesn't need a concept map, any assumption about concepts that it contains are considered unsafe, unless a concept map for those concepts is found or they are also marked `auto`.

Under the interpretation of `auto` as an abbreviation, this is natural, unlike the interpretation that marks the concept as implicit, where this behavior becomes a interoperability problem between implicit and explicit concepts, as discussed in [Str-09].

When a concept is defined as:

```
auto concept C<X1, ..., Xk> {
...
}
```

then `C<Y1, ..., Yk>` is equivalent to the body of `C`, where each $\mathtt{X}_i$ has been replaced with the corresponding $\mathtt{Y}_i$.

Readers familiar with the C++ literature should note the difference of these `auto` concepts compared to the ones in other concept papers. For example, the following code:

```
auto concept C<T> {
    void f(T);
}
auto concept D<T> {
    void f(T);
}

template <C T>
void g() {...}

template <D T>
void g() {...}
```

will generate an error for the second definition of `g` because another definition with the same requirements has already been defined.

This is in contrast with the usual proposed meaning of `auto`, that only affects matching and would interpret the above code as two distinct overloaded definitions of `g`.

5. Concept composition

This section discusses ways to combine concepts into another concept. Since the requirements of a concept are a predicate on scopes, it's natural to discuss the logical connectives, as long as we define what happens to the signature part of the concept.

5.1 And and union

The most natural way of composing two concepts is the And connective, where we define the signature of the composed concept as the union of the two signatures.

When we need an entity provided by both concepts, it's assumed that both concepts provide the same entity. If an operation has different signatures that can't be overloaded (for example, same parameter types but different result type), the call is considered ambiguous.

```
concept A<T> {
    ...
}
concept B<T> {
    ...
}
concept C<T> {
    A<T>;
    B<T>;
}
```

5.2 And and disjoint union

Since the interface composition for And is based on union, it's natural to conceive a different kind of And, that uses disjoint union on the APIs instead of the normal union.

In this case, when we use an entity provided by both concepts, we get an error, even if the two operations have the same signature, but we can disambiguate the use by qualifying it with the concept name.

Some papers use a mix of union and disjoint union for And, see for example [SGJ+-05].

The need of qualifiers may arise in some unexpected places. For example:

```
concept AddableNumber<N> {
    typename N;
    N::N(const N&);
    N::~N();
    N operator+(const N&, const N&);
}
concept MultiplicableNumber<N> {
    typename N;
    N::N(const N&);
    N::~N();
    N operator*(const N&, const N&);
}
concept Number<N> {
    typename N;
    AddableNumber<N>;
    MultiplicableNumber<N>;
    ...
}
...
template <Number T>
T f(const T& x, const T& y, const T& z) {
    return AddableNumber<T>::T(x * y) + z;
}
```

Here the user of `Number` must know how `Number` was implemented in order to properly disambiguate the copy constructor. Moreover, the disambiguation syntax must be more powerful than the usual syntax for name qualification (and it's hard to conceive a good syntax for some cases, for example for the implicitly-called destructors). Note that the example above still needs several fixes before it can work with this semantics: as far as the compiler knows, there are three types `T` with independent operations, so we have to add conversion operators between different `T`s in `Number`.

5.3 Or and intersection

Another possible connective is Or, used for example in [SR-05]. We don't know which of the two concepts holds, so we must take the intersection of the signatures. Note that, while computing the union (i.e., concatenation) of two signatures is a trivial operation, the intersection is not. In fact, in general we can't guarantee that every expression that typechecked in both signatures will typecheck in the intersection. For example, with the following code, the API of `C<U>` contains only `f`, because the two declarations of `g` are different.

```
concept A<U> {
    int f(U);
    void g(int);
}
concept B<U> {
    int f(U);
    void g(float);
}
concept C<U> {
    A<U> || B<U>;
}
template <A U>
void h1(U x) {
    return g(f(x)); // Ok.
}
template <B U>
void h2(U x) {
    return g(f(x)); // Ok, uses an implicit conversion.
}
template <C U>
void h3(U x) {
    return g(f(x)); // Error, g not found.
}
```

This connective has the odd property of requiring more than what it guarantees (guaranteeing more than what's required is not a problem, as in default type/operation definitions). This breaks the following code:

```
concept A<T> {
    void f(T);
}
concept B<T> {
    void g(T);
}

template <typename T>
requires { A<T> || B<T>; }
void do_stuff_helper(T x) {
    ...
}

template <typename T>
requires { A<T> || B<T>; }
void do_stuff(T x) {
    do_stuff_helper(x); // Error, neither A<T> nor B<T> hold
}
```

The call to `do_stuff_helper` is typechecked using the API of the `requires` clause (that is empty!) and the enclosing scope (that defines

`do_stuff_helper`). `do_stuff_helper` requires the existence of `f` and/or `g`, but they are not defined.

Checking the Or means checking separately the LHS and the RHS, and failing when both fail. If we instead define the checking by searching for a `A<T> || B<T>` assumption in the current scope, the above example typechecks. However, since we're not breaking the Or into subcomponents, if we switch the ordering in one of the two functions, the search will still fail.

We may patch this problem further by taking into account some properties of Or (commutativity, idempotence, distributivity with respect to And, etc.) but this would significantly slow down the search. Moreover, we would still have problems in some corner cases (depending on the chosen properties).

5.4 Xor and intersection

This is a stricter variant of Or. The signatures are still composed using the intersection, while the requirements on the concept are composed with Xor. This means that, in a scope where both concepts hold, their Xor will not hold. This can be motivated as an ambiguity between the two concepts. As for Or, we can't always provide an API where all expressions that typechecked in either APIs will typecheck.

5.5 Or as syntactic sugar

If we restrict the Or connective to `requires` clauses, we can define its semantics as just syntactic sugar for a set of declarations. For example:

```
concept C<T> {
    // Error, not in a requires clause.
    // A<T> || B<T>;

    template <typename T>
    requires { A<T> || B<T>; }
    void foo();
}

template <typename T>
requires { A<T> || B<T>; }
void bar() {
    ...
}
```

Would be translated into:

```
concept C<T> {
    template <typename T>
    requires { A<T>; }
    void foo();

    template <typename T>
    requires { B<T>; }
    void foo();

    template <typename T>
    requires { A<T>; B<T>; }
    void foo();
}

template <typename T>
requires { A<T>; }
void bar() {
    ...
}

template <typename T>
requires { B<T>; }
void bar() {
    ...
}

template <typename T>
requires { A<T>; B<T>; }
void bar() {
    ...
}
```

We are using concept-based overloading, that will be explained in chapter 11[1].

The intuition seems the same of the normal Or connective, but the typechecking is very different: now we split each declaration into all combinations of concepts that make the Or true, and we typecheck the body of `bar` three times, always allowing it to use either `A` or `B`, instead of just the intersection of their APIs.

The example that was problematic with Or and intersection now works:

[1]Due to the problems of concept-based overloading for types described in chapter 11, the Or-composition of concepts will only be allowed in constraints for template functions, not in the ones for template classes.

```
concept A<U> {
    int f(U);
    void g(int);
}
concept B<U> {
    int f(U);
    void g(float);
}
template <typename U>
requires { A<U> || B<U>;}
void h3(U x) {
    return g(f(x));
}
```

5.6 Xor as syntactic sugar

Similarly to the above feature for Or as syntactic sugar, we can do the same for Xor. We can use the `|` connective, inspired by BNF notation (note that unlike `||`, the `^^` connective is not currently in C++). The only difference is in how we split the declaration: now we don't generate a concept-based overload that requires both conditions. This makes any call to the above `bar` fail in any scope where both `A` and `B` hold.

Note that the check that `A` and `B` don't both hold doesn't have to be implemented as a special case: it will show up as an ambiguous overload — this is a consequence of the overloads that we didn't generate.

How do we handle `A | B | C`? In logic, the Xor will also be true when all three concepts are true. This is not much useful for concepts, since we use Xor when we *don't* want the conjunctions. However, the semantics explained earlier already works as we expect: from `A | (B | C)` we generate two definitions, respectively requiring `A` and `B | C`, and then we split the second one to get a total of three definitions, requiring `A`, `B` and `C` respectively.

This Xor connective is equivalent to the Or used in [GS-06].

5.7 Not

In some concept papers (e.g. [SR-05]), a Not combinator has been proposed, that negates the predicate and provides an empty signature.

The same issue that we saw for Or and Xor also arises with Not, even more seriously: from a function that requires `!A<T>` we can't possibly call another function with the same requirements — the API of `!A<T>` is empty, so we have no guarantees. As for Or and Xor, we can try to side-step the

issue by searching the negated requirement, but we get the same problems discussed above.

Moreover, Not only works under a closed-word assumption: when checking the Not we need to know all concept maps in the translation unit, while a positive requirement can be checked in the current scope.

In chapter 11 we discuss possible solutions for disambiguating concept based overloads without using Not.

5.8 Implication

Many concepts require operations that have their own `requires` clause. In the same way that a template clause corresponds to a "for all" in logic, a requires clause corresponds to a "such that", i.e., an implication.

For example, this is part of the `Vector` concept of the STL:

```
concept Vector<vector, T> {
    ...

    requires { EqualityComparable<T>; }
    void operator==(const vector<T>&, const vector<T>&);
}
```

Many papers have proposed this feature, but at the best of our knowledge a general form of implication has never been proposed yet. With this more general form, we could write the above code as follows:

```
concept Vector<vector, T> {
    ...

    requires { EqualityComparable<T>; }
    EqualityComparable<vector<T>>;
}
```

This means that not only `Vector` guarantees the existence of an `operator==()` in that case, but also guarantees the semantic properties of `EqualityComparable` (in this case, that such an operator is a congruence) and provides the default types and operations that it defines (in this case, `operator!=()`).

Adding the implication to the set of available connectives makes checking a concept significantly more complex, since the relevant requires clauses must be recursively checked, and giving rise to non-termination if we don't put some limitations on the checking (as discussed in section 3.1). This may seem a too high price to pay for having the most general form of implication in the language, but we have to do this for concept maps anyway.

The API of the implication is the intuitive one, while we must be careful when defining the requirements: if we define the implication's requirements as in propositional logic:

$$(A \to B) \Leftrightarrow_{\mathrm{Df}} (\neg A \vee B)$$

Then we must be able to check whether `A` is false, but with concepts we can only guarantee things to be true. This is because a declaration following the concept map may make a concept true, and then the following code could assume the concept map even though it would no longer hold.

Instead of this definition, we use one inspired by intuitionistic logic: in order to check `(A→B)`, we check `B`, but we allow all entities specified in `B` to have the additional requirement `A` in the `requires` clause. This is consistent with the semantics of requires clauses in constrained templates. For example:

```
concept VectorTemplate<vector> {
    template <Regular T>
    class vector {
        ...

        requires { EqualityComparable<T>; }
        EqualityComparable<vector>;
    }
}

template <Regular T>
class vector {
    ...

    requires { EqualityComparable<T>; }
    bool operator==(const vector& y) {
        // Uses == between values of type T.
        ...
    }
};

VectorTemplate<vector>;
```

When we check the concept map, it's not sufficient to notice that there is no `operator==` defined, so "the implication is vacuously true", because we can have declarations later on that would make the concept map false.

Instead, we check the requirements of `EqualityComparable<vector>`, adding `EqualityComparable<T>` in the `requires` clause of each entity (in

this case, `operator==`). So the definition of `operator==` in `vector` is accepted, even though `operator==` in `EqualityComparable` has no requires clause, and the concept map is satisfied.

This feature can be used instead of the "Encapsulated assertions" considered in [SR-05], extending it to functions and factorizing the specification in the concept rather than in model classes.

5.9 Discussion

We definitely need one of the two Ands and the first one seems the best choice, so that we don't have to qualify the entities that are common to various concepts. They wouldn't just be needed in some corner cases, they would be needed pretty much everywhere we combine two concepts, since some basic requirements (default constructor, destructor, etc.) are very common.

Or and Xor have several issues, especially the fact that they have guarantees weaker than their requirements. Instead, the limited versions of Or and Xor with the syntactic sugar semantic don't cause problems (this is a trivial consequence of them being just syntactic sugar) while still covering most of the use cases. Note that this interpretation also increases performance of the compiled code, since typechecking a function constrained with an Or using a richer API means that we can select more specific concept-based overloads.

The more general form of implication is definitely needed at least in concept maps, for example to express that `std::vector<T>` exists only as long as `T` is `Movable`. Therefore it doesn't add significantly more complexity in the compiler, and since it can be very useful (as we saw in the example above), we think that it should be included.

6. Templated requirements and concept maps

Many concepts require templated entities. For example, the `Vector` concept of the STL specifies a templated constructor that takes any range of `InputIterators`. Also, when specifying the semantic properties of templated code, templated concept maps are used.

In this chapter we try to define the semantics of these templated requirements, in a way that is consistent with the programmer's intuition in the different use-cases for templates.

A limited version of templated requirements was already presented in [SGJ+-05].

6.1 Checking templated concept maps

In this section we discuss how the parameters of a templated concept map can match the template parameters of a declaration in scope.

Fuzzy matching

There are cases where the template parameters are *not* expected to be specified explicitly, and the function is templated only to achieve a universal quantification on types. For example, given the following concept:

```
concept EqualityComparable<T, U> {
    bool operator==(const T&, const U&);
}
```

we expect that this concept map:

```
template <typename T>
requires { EqualityComparable<T, T>; }
EqualityComparable<vector<T>, vector<T> >;
```

will be satisfied by the following, more general, implementation:

```
template <typename T, typename U>
requires { EqualityComparable<T, U>; }
bool operator==(const vector<T>&, const vector<U>&) {
    ...
}
```

The matching algorithm is equivalent to performing overload resolution in the following code fragment, that can be automatically generated by the compiler from the concept map:

```
template <typename T>
requires { EqualityComparable<T, T>; }
bool __equal_to(const vector<T>& x, const vector<T>& y) {
    return operator==(x, y);
}
```

If the implementation of `operator==` above (or at least its declaration) is in scope, it will be one of the candidates for overload resolution.

Implicit instantiation is only allowed for functions, so this matching approach can't be used for types (and it would not make sense anyway).

Exact matching

In other cases, the template arguments are important to distinguish different functions, and the client code is expected to use explicit instantiation. Consider, for example, the following code:

```
concept Factory<factory, T> {
    typename T;
    class factory {
        T* create<T>();
    }
}

template <Regular T>
Factory<my_factory, T>;
```

A class `my_factory` that specifies a non-templated `create()` method does not satisfy the concept map, even if the return type of `create()` is convertible to every pointer type. In this case, the type used to instantiate `create()` is as important as a function argument.

Not all functions use this interpretation, but all types do, since template types must always be explicitly instantiated.

The implementation of this use-case needs a matching semantic that is quite different from the fuzzy matching explained earlier. The matching algorithm can still be implemented by typechecking an artificial function definition, but we will use explicit instantiation in the function body:

```
template <Regular T>
T* __create(my_factory& x) {
    return x.create<T>();
}
```

Note that the call to the `create` method is allowed to use conversions in the parameters (here only the implicit `this` parameter) and/or in the return type, as always in pseudo-signatures.

The artificial `__create` function can be an actual wrapper function generated internally by the compiler or can become a piece of inlined code — we only require its use to determine overload resolution.

The names beginning with a double underscore have only been chosen to simplify the explanation, but such types and functions should be unnamed (or have unique names), so that multiple concept maps for the same concept are possible.

Discussion

Since both uses of templates are widespread in C++, we will use both matching strategies. The second one is stricter, so we will use it first and then fall back on the first one (for functions only).

6.2 Templated concept maps and unconstrained templates

How do we check the requirement of a constrained template in a given scope? For example, if we have the following code:

```
concept Fooable<T> {
    void foo<T>();
}

template <Regular T>
Fooable<T>;
```

which of the following files satisfy the above concept map?

```
// A.h
template <Regular T>
void foo();
```

```
// B.h
template <Semiregular T>
void foo();
```

```
// C.h
template <typename T>
void foo();
```

```
// D.h
template <typename T>
void foo() {
    T();
}
```

```
// E.h
template <typename T>
void foo() {
    T::draw();
}
```

`A.h` certainly does satisfy the requirements, since they are the same as the ones of the concept map. `B.h` contains a definition of `foo` with a weaker `requires` clause, so it's also ok.

What about `C.h`, `D.h` and `E.h`? As we did for `B.h`, we must check that the template requirements of the declaration are weaker than (or equal to) `Regular<T>`. Since these are unconstrained templates, the requirements are implicit, so we will try to typecheck the unconstrained template against the API provided by `Regular<T>`. This is the same check that would have been done if we added `Regular<T>` to the requirements of these templates, making them constrained.

So, we can accept `D` but not `E`. Even for `D`, we must be careful, since in C++ a template can be specialized, and there may be some specializations later on. So we must check all specializations that are already defined, we will store the requirement `Regular<T>` together with the definition of `foo`, and we will check it for each specialization defined later on in the same translation unit.

We can do the same thing for `C`, the only difference is that we also have to check the definition as well as the specializations, and check that the definition is encountered before the end of the translation unit.

When implicit concepts are used, allowing template concept requirements to match unconstrained templates requires the compiler to backtrack after a sequence of successful template instantiations followed by an unsuccessful one, and then resuming compilation. This is considered hard to implement in existing compilers (see [SGJ+-05]).

6.3 Specializing constrained templates

Should we allow the programmer to define template specializations for a constrained template? If we do, we must decide which requirements are allowed for the specialization. We have the following options:

1. The specialization can have any requires clause.
2. The specialization must have the same requires clause as the general template.
3. The specialization must have a requires clause that implies the requirements of the general template.
4. The specialization must have a requires clause that is implied by the requirements of the general template.

These options are the possible solutions of a trade-off: the programmer implementing the specialization may want to use stricter requirements, because the additional requirements are "always" satisfied in that special case, while the compiler typechecking the client code together with the prototype of the general template (without necessarily knowing of all specializations) needs the requirements of the specialization to be at most as strong as the ones in the general template in order to guarantee that the call typechecks.

The combination of these two needs seems to exclude the options 1 and 4 above. However, there is another solution, similar to what we did in the previous section. If we use option 1 (the one giving the most freedom to the programmer) the compiler can't guarantee that the instantiation of the constrained template typechecks without looking at the specializations — but the C++ standard guarantees that all applicable partial specializations have been declared before use, so we can typecheck the instantiation first with the general template and then with all known specializations (we need only the prototypes, so we achieve separate typechecking). For total specializations of functions, the situation is different: they may be defined in another translation unit, so we have no way to detect them. Luckily, this is not an issue: in this case, the compiler will already fully typecheck the specialization (since there are no template parameters), and no further checks are needed.

6.4 Syntax of templated requirements

Declaration-like syntax

We can specify the existence of templated entities using the same syntax used to declare template classes and functions. For example, we can write:

```
concept Pair<pair, T1, T2> {
    class pair {
        ...
        template <typename U1, typename U2>
        requires {
            void operator=(T1&, U1&&);
            void operator=(T2&, U2&&);
        }
        pair& operator=(pair<U, V>&& p);
    }
}
```

Note that here the class pseudo-definition syntax is no longer just syntactic sugar, but disambiguates `pair<U1, U2>::operator=` from `pair::operator=<U1, U2>`.

Instantiation-like syntax

Another possible syntax is the following, where we write the `<U1, U2>` explicitly:

```
concept Pair<pair, T1, T2> {
    class pair {
        ...
        template <typename U1, typename U2>
        requires {
            void operator=(T1&, U1&&);
            void operator=(T2&, U2&&);
        }
        pair& operator=<U1, U2>(pair<U, V>&& p);
    }
}
```

Discussion

The use of pseudo-signatures to specify functions suggests the use of a declaration-style syntax.

Another reason to avoid the instantiation-like syntax is that it is redundant and this redundancy allows the user to write malformed code. For example, this specification has no meaning:

```
template <typename T, typename U>
requires {
    ...
}
void f<T>();
```

Finally, this syntax is already used in C++ for partial specialization of template classes, and this may generate further confusion.

6.5 An explanation-only forall keyword

The instantiation-like syntax, however, will be useful for explanation purposes in the rest of this chapter. To avoid confusion, we will use the `forall` keyword in these cases, as in the following code:

```
forall <typename T>
requires {
    ...
}
void f<T>();
```

Which means:

```
template <typename T>
requires {
    ...
}
void f();
```

Decoupling the two effects of the template clause will allow us to treat them separately. Firstly we will define the API of templated concept requirements in terms of code containing the `forall` keyword and then define the meaning of such code.

6.6 API of templated concept requirements

The two use-cases for template arguments (relevant type parameters or just universal quantifications on types) also affect the handling of templated concept requirements. By “templated concept requirements” we don’t mean “a template’s concept requirement”, but situations like the following:

```
concept VectorTemplate<vector> {
    template <Movable T>
    class vector {
        ...
    }

    ...

    template <Movable T, Movable U>
    requires { EqualityComparable<T, U>; }
    EqualityComparable<vector<T>, vector<U> >;
}
```

Here we want to specify that for every suitable pair of type `T` and `U`, `vector<T>` and `vector<U>` are `EqualityComparable`.

This section discusses how the entities in the API of the required concept are affected by the template clause. Due to the feature of concept maps affecting visibility explained in section 3.1, the following discussion also applies to the API of a templated concept map.

No name modification

In some cases no name modification is needed, typically when exact matching is desired.

```
concept Factory<factory, T> {
    typename T;
    typename factory;
    T* factory::create<T>();
}
concept RegularFactory<factory> {
    template <Regular T>
    Factory<factory, T>;
}
```

Here, the API of `RegularFactor` will contain of a type `my_factory` and, for every regular type `T`, of a method `factory::create<T>`. Using the `forall` keyword, we can express the API as follows:

```
concept RegularFactory<factory> {
    forall <Regular T> // ???
    typename factory;

    forall <Regular T>
    T* factory::create<T>();
}
```

The `forall` clause for `factory` has been commented with `???`, since in fact `factory` does not depend on `T` nor on any other type in the API of the concept constraints (in this case, `Regular<T>`, that provides no other types). We'll address this issue later on in this chapter. For now, note that there is no `factory<T>`, just `factory`.

Unconditional name modification

For fuzzy matching, on the other hand, we expect that the names are modified, adding the template parameters.

```
concept EqualityComparable<T> {
    typename T;
    bool operator==(const T&, const T&);
}
concept VectorTemplate<vector> {
    ...
    template <EqualityComparable T>
    EqualityComparable<vector<T>>;
}
```

The API of the above `VectorTemplate` concept is equivalent to:

```
concept VectorTemplate<vector> {
    ...
    forall <EqualityComparable T>
    bool operator==<T>(const vector<T>&, const vector<T>&);
}
```

Note the added `<T>` in the declaration of `operator==`.

Name modification for functions only

Since types always use exact matching, we can safely use the no-name-modification approach for them, while using the second approach for functions.

We will only add the template parameters that aren't already parameters of the function itself or of one of the enclosing types. For example:

```
concept Factory<factory, T> {
    typename T;
    typename factory;
    T* factory::create();
}
concept RegularFactoryTemplate<factory> {
    template <Regular T>
    Factory<factory, T>;
}
```

Leads to a `RegularFactoryTemplate` concept with the same API as:

```
concept RegularFactoryTemplate<factory> {
    forall <Regular T>
    typename factory;

    forall <Regular T>
    T* factory::create<T>();
}
```

While this modified version:

```
concept Factory<factory, T> {
    typename T;
    typename factory;
    T* factory::create();
}
concept RegularFactoryTemplate<factory> {
    template <Regular T>
    Factory<factory<T>, T>; // Note the <T>.
}
```

Leads to this API:

```
concept RegularFactoryTemplate<factory> {
    forall <Regular T>
    typename factory<T>;

    forall <Regular T>
    T* factory<T>::create(); // Not create<T>.
}
```

Using this semantics, the examples that we showed for the previous two approaches keep working as expected.

6.7 Translating forall clauses back into template clauses

Functions

All the template arguments to functions are guaranteed to be part of the name of the function (or of an enclosing type) as is in current C++, so we can directly map `forall` into `template` clauses.

Types depending on all parameters

When a type already depends on all the template parameters (either directly or indirectly through an enclosing type), like `factory<T>` in the last example, we can simply replace the `forall` with a `template` clause using the same parameters as the type.

This code:

```
forall <Regular T, Iterator I>
typename outer<I>::inner<T>;
```

Becomes:

```
template <typename I>
template <typename T>
requires {
    Regular<T>;
    Iterator<I>;
}
typename outer<I>::inner;
```

Types depending on only some parameters

Consider the following requirement:

```
forall <Regular T, Iterator I>
typename outer<I>;
```

In general we can't split the requirements for the different template parameter lists. This means that there is no general way to transform the above requirement into something such as:

```
forall <Iterator I>
typename outer<I>;
```

Since we can't deduce this declaration for the one above, we will require such a declaration to be in scope when the doubly-quantified one is encountered.

What we can do, however, is to check that the declaration in scope is consistent with the requirements of the overly-quantified declaration. This can be done by generating an artificial function and then typechecking it.

In our example, we will generate a function such as:

```
template <typename I, typename T>
requires {
    Regular<T>;
    Iterator<I>;
}
void __unnamed() {
    typedef outer<I> __outer;
}
```

Since the requirements in the declaration for `outer` above allow this use to typecheck, we accept the requirement, but we will not add anything more to the API. This semantics allows the definition of concepts such as:

```
concept EqualityComparable<T, U> {
    typename T;
    typename U;
    bool operator==(const T&, const U&);
}
concept VectorTemplate<vector> {
    template <Regular T>
    class vector { ... }

    template <Regular T, Regular U>
    requires { EqualityComparable<T, U>; }
    EqualityComparable<vector<T>, vector<U>>;
}
```

That has the following API:

```
concept VectorTemplate<vector> {
    template <Regular T>
    class vector { ... }

    forall <Regular T, Regular U>
    requires { EqualityComparable<T, U>; }
    typename vector<T>;

    forall <Regular T, Regular U>
    requires { EqualityComparable<T, U>; }
    typename vector<U>;

    forall <Regular T, Regular U>
    requires { EqualityComparable<T, U>; }
    bool operator==<T, U>(const vector<T>&, const vector<U>&);
}
```

After checking that the first definition of `vector` is more general than the following two `forall`s, the API of the concept is finally translated into:

```
concept VectorTemplate<vector> {
    template <Regular T>
    class vector { ... }

    template <Regular T, Regular U>
    requires { EqualityComparable<T, U>; }
    bool operator==(const vector<T>&, const vector<U>&);
}
```

6.8 Unconstrained template requirements

This feature allows a concept to require unconstrained templates. This can be useful to better interact with non-annotated code.

Constrained templates that require such a concept in their `requires` clause are not allowed to instantiate such unconstrained templates, because it's impossible to guarantee that the instantiation is allowed. This doesn't make this feature useless: as long as we are using concept maps that affect visibility, as we saw in section 3.1, all code after such a concept map (non-templated code, unconstrained templates and constrained templates) is allowed to instantiate such templates. For constrained code, we check that the unconstrained template instantiates correctly on the archetypes, and for non-templated code we do the usual checks on the concrete types.

To be able to instantiate the template after a concept map, we also need the definition of the unconstrained template (not just its declaration) unlike for requirements of constrained templates.

This feature becomes very important together with concept-based modules (see chapter 9), because it's the only way for a module to export unconstrained templates.

This is some example code:

```
concept C<> {
    template <typename T>
    void f();
}

template <typename T>
void f();

C<>; // Error, f is not yet defined.

requires { C<>; }
void nop() {
}

requires { C<>; }
void nop2() {
    nop(); // Ok, even though C is needed by nop() and we
           // can't access the definition of f, we are not
           // instantiating f. Note that we don't have to
           // look at the body of nop(): if it typechecked
           // we know that it doesn't instantiate f.
}

requires { C<>; }
void nop3() {
    f<int>(); // Error, can't instantiate a non-constrained
              // template provided by the requires clause
}
```

```
void nop4() {
    nop2(); // With implicit concepts:
            // Error, f is declared but not defined, so we
            // can't generate an implicit c.map.
            // With explicit concepts:
            // Error, no concept map for C found.
}

template <typename T>
void f() {
    T x;
    x.some_method();
}

C<>; // Ok now.

void do_nothing() {
    nop2(); // Ok.
}

requires { C<>; }
void do_something() {
    f<int>(); // Error, can't instantiate a non-constrained
              // template provided by the requires clause.
}

void do_something2() {
    f<int>(); // Ok, a concept map for C is found (for
              // implicit concepts the definition would be
              // enough, the concept map is not needed).
}
```

We think that both constrained templates and unconstrained ones should be allowed in concepts if concept-based modules are used. Otherwise, unconstrained template requirements become nearly useless.

7. Same-type constraints

Same-type constraints specify that two types are in fact the same type. They have been discussed in the C++ concepts' literature since 2005, starting with [SGJ+-05].

Some papers only use same-type constraints as requirements: when a concept is instantiated with concrete types, all same-type constraints are checked. However, they can be more useful if we also add this information to the signature (as done in [SGJ+-05], but not in some of the later papers). Consider this example with concepts from the standard:

```
concept AssociativeContainer<container> {
    class container {
        typename key_type;
        ...
        iterator find(key_type);
    };
};
concept SimpleAssociativeContainer<container> {
    class container {
        AssociativeContainer<container>;
        SameType<key_type, value_type>;
        ...
    };
};

template <SimpleAssociativeContainer container>
container::iterator
f(container& c, const container::value_type& x) {
    return c.find(x);
}
```

Our intuition tells us that `f` should pass typechecking.

We are calling the `container::find()` method with a `value_type`. `SimpleAssociativeContainer` guarantees that a `value_type` is the same as a `key_type`, and also that such a method exists for a `key_type`.

This kind of reasoning expects that `SameType` constraints are also exposed in the signature provided by the concept, so our main focus in this section will be to analyze how to do that. The checking part of a `SameType` constraint is trivial, so we won't discuss it further.

7.1 Arbitrary type constraints

In many papers, due to a greater focus on the requirements part than on the signature part of a concept, it's possible to state the equality of any two types. Even in [SGJ+-05], that discusses the use of same-type constraints in the signature part, no limitations on type equalities are proposed.

When templated concept maps and templated concept requirements involve same-type constraints, during typechecking we have to try proving such requirements to determine the available type equalities.

Unlike the undecidability of the search of concept maps, that we patched using an arbitrary limit to the recursion, we prefer to guarantee termination of the search of same-type equalities, by limiting the expressible constraints.

7.2 Limiting expressiveness

We impose the following limitations on the `SameType` constraints:

- The left-hand side of a `SameType` constraint must be of the form `T::I` or `T::I<T1, .., Tk>`, with `T` a namespace or a type and `I` an identifier. `T1`, ..., `Tk` can be anything that's allowed as a template parameter. We say that `T::I` is the type defined by the same type constraint. The `T::` prefix is optional.

- All the declarations of a type, with or without a same-type constraint, must have the same template arguments (if any) and `requires` clause. As an exception, two definitions that differ only in the requirements are allowed if the requirements of one imply the requirements of the other; in this case, the definition with the stronger requirements is ignored.

- We say that a type `T::I` defined in a `SameType` constraint *depends* on all the entities named in the right-hand side of the constraint or in `T`.

- The graph of the dependencies must have no cycles, except cycles of the form `SameType<T1, T2>`, ..., `SameType<Tk-1,Tk>`, `SameType<Tk, T1>` (including the case `SameType<T, T>`, with $k = 1$).

- If a type is defined in more than one same-type constraint, the RHS of these constraints must be the same type (taking into account the same-type constraints defining the types in the RHS and their dependencies).

Since (the specification of) a `SameType` constraint is no longer symmetric, we'll use the following notation instead:

```
typename T::I = U;
```

This helps to convey the intuition that `T::I` is defined in terms of `U`. Note that this is only relevant for the provided signature part; when we check a same-type constraint we just check that the two types are the same, no matter which `typedef`s or same-type constraints are involved.

The limitations above can prevent to compose two concepts with And, even in cases where both concepts in isolation are allowed. For example:

```
concept Semiregular<T> {
    ...
}
concept Regular<T> {
    Semiregular<T>;
    ...
}
concept A<vector> {
    template <Regular T>
    class vector {
        typename iterator;
        typename value_type;
        typename iterator::value_type = value_type;
        ...
    }
}
concept B<vector> {
    template <Semiregular T>
    class vector {
        typename iterator;
        typename iterator::value_type;
    };
}
concept C<vector> {
    A<vector>;
    B<vector>;
    // Error: B<vector> has a definition for
    // vector<T>::iterator::value_type that takes
    // any Semiregular, but we have a same-type
    // constraint for vector<T>::iterator::value_type
    // that has a different requirement (Regular).
}
```

We would obtain the same error if we just concatenated the bodies of `A` and `B` in a single concept. We can make `C` compile by adding a same-type constraint equal to the one in `A`, but that requires only `Semiregular`.

With the above limitations, it's possible to consider same-type constraints as (possibly templated) `typedef`s, as long as the signature part of the concept is concerned, and the no-cycle requirements ensure that with a topological order of the graph we can obtain an ordering of the `typedef`s where we only use previously-defined types.

The trivial cycles that are allowed as an exception to the no-cycle limitation can be handled by simply removing an edge from the cycle. The removed type equivalence will be guaranteed anyway by the remaining ones.

In this way, for each type `T` provided in the signature of a concept, we can identify a *canonical type* by simply expanding all the `typedef`s in `T`. The requirement that all `template` and `requires` clauses of the definitions of a type should be the same allows us to do this expansion ignoring the `requires` clauses completely — if a type `U` appears within `T`, then we know that the `requires` clause of the same-type constraints defining `U` (if any) is satisfied. Otherwise, we wouldn't be allowed even to name `U` within `T`.

The signature of a concept containing same-type constraints can then be transformed replacing each type with its canonical type (except in the LHS of same-type constraints), obtaining an equivalent signature. We still have to retain the same-type constraints, so that a function that requires such a concept can use interchangeably the aliases defined with same-type constraints or the types they are defined to.

Let's see how this works in the example we cited earlier:

```
concept AssociativeContainer<container> {
    class container {
        typename key_type;
        typename iterator;
        ...
        iterator find(key_type);
    };
};

concept SimpleAssociativeContainer<container> {
    class container {
        AssociativeContainer<container>;
        typename key_type = value_type;
        ...
    };
};
```

In this case, the type `container::key_type` depends on `container::value_type`, and there are no other dependencies (so there are no cycles). Both definitions of `container::key_type` have no

template and no requires clause, so AssociativeContainer can be combined with the additional requirements in SimpleAssociativeContainer.

After these checks, we can transform SimpleAssociativeContainer, obtaining an internal representation of the API similar to the following:

```
concept SimpleAssociativeContainer<container> {
    class container {
        ...
        // Signatures use canonical types only.
        iterator find(value_type);
        typename key_type = value_type;
    };
};
```

The function f defined earlier now typechecks correctly, since it finds an exact match in the signature. Within f, key_type and value_type can be used interchangeably, since any use of the first is expanded to the second while canonicalizing the types.

From the point of view of f, the equality relation between types is now an equivalence relation, since it's based on a syntactical comparison of the canonicalizations, so it's the equivalence relation induced by the canonicalization function.

7.3 Discussion

We think that it's important to have a same-type facility integrated with the signature part of a concept, so that such assumptions can be used by the typechecker for concept-constrained code.

The above set of limitation makes same-type constraints decidable, and still allows enough flexibility to express most constraints. For the type equalities that do not satisfy these limitations, we still keep the SameType built-in concept as used in the literature.

Since the non-symmetric syntax is a stronger requirement, such requirements must imply the same requirements written with SameType. This is the only reason why SameType must be builtin instead of being a concept in the standard library implemented using std::is_same in a constexpr constraint.

8. Explicit and implicit concepts

One of the main discussions regarding concepts in C++ is about the choice between explicit and implicit concepts. In [Str-09] Stroustrup argues that we should only allow one of the two kinds of concepts, but not both, because they don't interact well with each other. So, unlike most of the literature, we don't use the `auto` keyword to mark implicit concepts, nor `explicit` to mark explicit ones. Instead, we discuss whether `concept` means implicit or explicit. We use the `auto` keyword with a slightly different meaning, see section 4.3.

8.1 Explicit concepts

Using explicit concepts means that we never assume a concept map unless the user has explicitly written it. If a function requires a concept instantiation for which no concept map has been provided by the user, a compile error is raised. If a concept requires another one, a concept map for the "base" concept is implicitly generated when the user writes a concept map for the "derived" one.

```
concept Comparable<T> {
    typename T;
    bool operator<(const T&, const T&);
}

struct point {
    int x, y;
};
bool operator<(const point& p, const point& q) {
    return p.x < q.x || (p.x == q.x && p.y < q.y);
}

template <BidirectionalIterator Itr>
requires { Comparable<Itr::value_type>; }
void sort(Itr first, Itr last);

Comparable<point>; // Required!
```

```
void do_stuff(vector<point> v) {
    sort(v.begin(), v.end());
}
```

8.2 Implicit concepts

Using implicit concepts means that, when we don't find a matching concept map, before raising an error we check the syntactic requirements of the concept in the current scope. If they are, we assume that the concept map holds and no error occurs. This approach favors duck-typing over correctness: it makes semantic assumptions on the requirements implicitly, but such assumptions may not hold. This is similar to what happens in C++ without concepts, since only the (implicit) syntactic requirements are checked. For example:

```
concept Instrument<instrument> {
    class instrument {
        void play(string song_title);
    }
}

class Piano {
    ...
public:
    // Loads the specified song and plays it.
    void play(string song_title);
};

class HeartsGame {
    ...
public:
    // Starts a new game using the specified user name.
    void play(string user_name);
};

template <Instrument I>
void replay_last_song(const I& x) { ... }

void do_stuff() {
    Piano p;
    replay_last_song(p); // Ok, no concept map needed.

    HeartsGame g;
    replay_last_song(g); // No error detected.
}
```

8.3 Implicit concepts with optional warnings

A possible compromise between implicit and explicit concepts is to use implicit concepts, but optionally emitting a warning (which is switched on using a compiler flag) when we assume a concept map that the user hasn't written.

When the compiler flag is not specified the warning is disabled, and we get the same behavior that we would get with implicit concepts. On the other hand, if we use the compiler flag and we have a no-warning policy (that can be enforced with a flag like `-Werror` for GCC), we are using explicit concepts.

It's of course allowed to have the compiler flag enabled without `-Werror`, if we think that such unsafe assumptions are important enough to be displayed but not to interrupt the compilation.

The decision between implicit and explicit concepts is made per-compilation-unit instead of per-concept as it's common in the C++ literature, and this allows different projects to use the same concepts and generic code with different policies.

The use of `#include` directives to import libraries will produce these warnings not only for our own code, but also for other libraries that our code uses. It should be possible to hide these warnings with a compiler flag, for example with the (existing) `-isystem` of GCC[1].

A project that does not adopt a no-warning policy (i.e. it uses concepts as implicit) can switch to concepts even before the libraries it depends on do so. To use concepts as explicit, instead, more work is needed if the dependencies aren't concept-aware yet, and this discourages (but doesn't prevent) projects to switch to concepts with a no-warning policy before the libraries that they depend on do the switch.

For example:

```
// library_a.h
template <typename T>
void f() {
    ...
}
template <typename T>
class my_class {
    ...
}
```

[1] When using concept-based modules (see chapter 9) there is no need for `-isystem`.

```
// library_b.h
#include "library_a.h"
#include <library_a>

template <Assignable T>
void process_assignable();

template <Regular T>
void do_stuff {
    f(); // Ok, the definition of f is visible so we
         // can check it on the archetype.

    process_assignable<my_class>(); // Warning:
                                    // Assignable<my_class>
                                    // may not hold.
}
```

In this example, the library B has switched to concepts before A, and gets one warning that wouldn't be there if A had switched to concepts and specified a concept map for `Assignable<my_class>`[2]. So, for the library B to switch to concepts before A, it needs to write concept maps that describe the public interface of A. This is most easily done by using a concept-based module to wrap the existing implementation of A (see chapter 9). Only the entities in the public API of A need wrapping, while the ones used internally by A or in the API of libraries used by A don't. The code of A can be re-used because a concept map with a requires clause can match an unconstrained template, as long as the template definition is visible and the template typechecks against the requirements specified in the concept map.

8.4 Discussion

As we saw, both extremes have disadvantages: with implicit concepts we risk missing errors due to accidental matches, while with explicit concepts we have to explicitly write concept maps in order to use an entity that requires a concept.

Mixing implicit and explicit concepts by marking relevant concepts as `auto` is not a solution, since the different opinions on the implicit vs explicit concepts issue will mean different opinions on what concepts should be `auto`. For example, advocates of explicit concepts will want a non-`auto`

[2] By «switching to concepts» we mean not only that the templates have been constrained, but also that concept-based modules have been adopted (see chapter 9). Otherwise a library that decides to use implicit concepts may not specify such a concept map anyway.

`EqualityComparable` concept to make sure that `operator==` is an equivalence relation, while proponents of implicit concepts will prefer it to be `auto` so that the related concept maps aren't needed.

The compromise between safety and conciseness has very different outcomes depending on the project and it is an inherently subjective and context-dependent decision. This is a serious issue for the standard library already, but becomes even worse when third party libraries are considered, since each library decides independently from the others and forces such decision on all users.

If we use explicit concepts, in order to maintain backward-compatibility with client code even with a concept-enabled STL, we would need a compile-time switch that hides all the requires clauses.

In the above example about a vector of points, if this was written in C++ without concepts, there would be no concept map in the client code nor requires clause in `sort`, and it would compile. If we then add a requires clause for `sort` (in the STL), the client code won't compile anymore because it lacks a concept map.

Using macros to hide the requires clauses would significantly clutter the concept-enabled library code (both for the STL and for any other template library), so it would discourage users from switching to concepts. The best option in this case is probably to have a compiler flag that turns on the concept checking. However, this allows a library to become concept-aware only when all of its dependencies are, and would slow down the adoption of concepts.

If we use implicit concepts with warnings, instead, legacy code will still compile and work as before while an actively maintained library can choose to activate the warning and gradually reduce the number of warnings by adding the required concept maps, eventually being able to adopt a no-warning policy and get all the safety guarantees of explicit concepts (if desired).

Making concept maps not compulsory also helps teaching C++, since the STL can be taught before concepts and concept maps. Otherwise, the types and algorithms in the STL could only be used with builtin types and types defined in the STL itself, and not with user-defined `struct`s and classes.

Based on these observations, we favor implicit concepts with optional warnings because:

- Each project can independently choose implicit or explicit concepts, while still using the same concept definitions used by projects that did a different choice.

- Backward compatibility is maintained without needing to disable concept checking (so that we still get simpler error messages and a stricter typechecking on at least part of the code).
- Any library can become concept-aware independently from the client code, and somewhat independently also from the libraries that it uses.
- A library can be made concept-aware iteratively, and can be compiled/tested at each step.
- It is easier to use C++ for fast prototyping (i.e. with implicit concepts), and the tighter checks can be introduced later if needed.
- Learning C++ is easier since concepts can be taught later, when the student is already familiar with the STL.

We don't discuss the integration of implicit concepts with concept-based overloading here, because we do so in chapter 11.

Even though this design choice allows the use of implicit concepts, we recognize the power of concept maps, and we think that we should encourage their use in production code by making them more powerful (see section 3.1) and integrating them with modules (see chapter 9).

We would like many C++ programmers to *decide* to actively use concepts and concept maps, but we won't force them to do so. This is in the same spirit as the following (by Stroustrup, from [Str-09]):

«I will argue that "average programmers" should write concepts and (less frequently) concept maps, that it is good for C++ that they do so, and that they will only do so if they see benefits from doing so.»

9. Concept-based modules

C++ still lacks modules, even though they have been discussed for inclusion in the standard (see [Van-06]). In this chapter we discuss a different approach to modules for C++, which relies on concepts (a brief discussion of concept-aware modules can be found in [JWL-04]).

One of the roles of concepts is as a specification of an API. From this point of view, it's natural to think about using concepts to specify the entities that should be exported from a module. In this way, instead of having to split the code in a public and a private part, and having to duplicate some of this information in concept maps, we can reuse such concept maps for describing the API of the module.

A second advantage of concept-based modules is that for entities that have a definition in the concept, the implementation is free to drop its definitions and let the clients use the ones in the concept.

Finally, since the easiest way of exporting the required interface will be to specify a concept map for it in the export block, the programmer will be encouraged to explicitly state the semantic properties of his code, so that the client knows what properties he can depend on. These concept maps also allow different modules to use different policies regarding the implicit vs explicit concept issue, since all interactions between modules will happen through concept maps anyways.

Note that this requires all libraries, even the ones implemented with an implicit concept policy, to describe their public API using concepts. We think that this is not too high a burden, since the API has to be documented anyway, and concepts are the natural way to do so.

An `export` clause can contain anything that can be in a concept body. This means that it is allowed to directly specify function signatures in the `export` block, but such operations don't provide any semantic guarantees, so concept maps should be preferred. We will use the `import` keyword to import other modules or module partitions, and the `export` keyword to export declarations and concept maps.

```
// In stack_concept.h:
concept Stack<stack, T> {
    class stack {
        stack();
        void push(const T&);
        void pop();
        bool empty() const;
    };
}
```

```
// In my_stack.cpp:
#include "stack_concept.h"

export my_stack {
    template <Regular T>
    Stack<stack<T>, T>;
}

import std;

template <typename T>
requires { Regular<T>; } // Optional
class stack {
public:
    void push(const T& x) {
        v.push_back(x);
    }
    void pop() {
        v.pop_back();
    }
    bool empty() const {
        return v.empty();
    }

private:
    std::vector<T> v;
};
```

```
// In main.cpp
import my_stack;

int main() {
    my_stack<int> s;
    s.push(1);
    std::vector<int> v; // Error: std::vector is not declared
                        // in this scope.
    return 0;
}
```

When a module is compiled, after typechecking it, its `export` clause is processed, and all entities provided by its body are placed in a special section of the compiled module, that is structured in a way that allows fast queries without scanning the entire file.

When the `export` clause is encountered, all the entities it provides are added to the following scope — we only check that the needed types and operations have been defined when we reach the end of the translation unit. This allows the `export` clause to be close to the beginning of the file, since it will probably be much shorter than the implementation below. Also, circular dependencies between types and operations no longer need to be handled using additional declarations (for example, forward declarations for classes), since all names in the API of the concept will be available as if they were declared and not yet defined.

The tradition of splitting the API (the header files) from the implementation is respected, the only difference being that the API is now expressed using a concept (that can be defined more concisely by reusing other concepts) and an export clause. The private parts are now hidden from the header file, though, and we think that this provides a better separation of the API from the implementation.

9.1 Multiple levels of visibility

This syntax and semantics for modules also allows to have an unlimited number of levels of visibility. Usually, in C++ there are 3 levels (`public`, `protected` and `private`) together with the `friend` feature to make the levels more flexible. With concept-based modules, we can have any number of levels, for example:

```
// src/A.h
concept A<T> {
    void f(T);
}
```

```
// src/package_Foo/A.h
#include "../A.h"
concept package_Foo_A<T> {
    A<T>;
    void g(T);
}
```

```
// src/package_Foo/subpackage_Bar/A.h
#include "../A.h"
concept subpackage_Bar_A<T> {
    package_Foo_A<T>;
    void h(T);
}
```

```
// src/package_Foo/subpackage_Bar/A.cpp
#include "A.h"

export mylib::package_Foo::subpackage_Bar::A {
    subpackage_Bar_A<int>;
}

void f(int x) {
    ...
}
void g(int x) {
    ...
}
void h(int x) {
    ...
}
```

```
// src/package_Foo/A.cpp
#include "A.h"

export mylib::package_Foo::A {
    package_Foo_A<int>;
}

import mylib::package_Foo::subpackage_Bar::A;
```

```
// src/A.cpp
#include "A.h"

export mylib::A {
    A<int>;
}

import mylib::package_Foo::A;
```

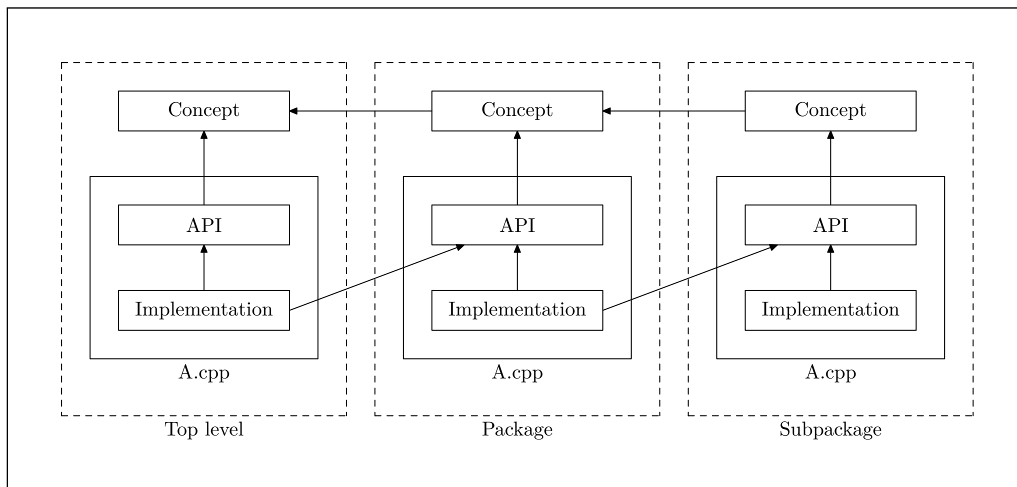
With the above code, a single implementation exposes a wide API to the code in the same subpackage, a tighter API to the code in the same package and a even more tight API to the code in other packages and to code outside the library. This approach can be iterated, providing an unlimited number

of levels of visibility. The additional levels can be very useful for several reasons — for example, exposing an extended API to unit tests.

We don't have to specify the common parts of the API each time: we can require a concept describing a part of the API and only write additional entities/properties, as in the code above.

Note that, unlike `public`, `private` and `protected`, we can also restrict the visibility of non-members.

From the following diagram:



we can see that the implementation of a level does not depend on the implementation of other levels.

In the diagram, we call "API" the part of the module up to and including the `export` clause, and "implementation" the rest of the module.

9.2 Extension A: import of module source

To allow cyclic dependencies between parts of a module (but not between modules), we can also allow a different import statement that imports a module source file rather than a compiled module. This is easier to explain on an example:

```
// song.h
concept ISong<Song> {
    class Song {
        ...
        void play() const;
    };
}
```

```
// song.cpp
#include "song.h"

export Song {
    ISong<Song>;
}

import Player;
... // Other imports/includes.

class Song {
    ...
public:
    void play() const {
        Player::current().play(*this);
    }
};
```

```
// player.h
import "song.cpp";

concept IPlayer<Player> {
    class Player {
        ...
        void play(const Song&) const;
        static const Player& current();
    };
}
```

```
// player.cpp
#include "player.h"

export Player {
    IPlayer<Player>;
}

... // Other imports/includes.

class Player {
    ...
public:
    void play(const Song& x) const {
        ...
    }
};
```

Note that there is a cyclic dependency between the `Song` and `Player` classes, but there is no cycle between `ISong` and `IPlayer`. Such a cycle

would not be allowed, because the definition of a concept is required before using that concept.

When `import` is used with a filename, it processes the specified source file up to and including the `export` block, but not further. In this way, all concepts (and other entities, if any) that are required for the `export` clause are processed, but the implementation is not. So the following `import` is not processed (this would cause an infinite recursion).

This situation would be tricky to handle in the current C++ language, because `song.cpp` would include `player.cpp` and vice versa. Even though there would be no infinite loop, due to the include guards, the compiler will not compile the resulting translation unit because, no matter which class is defined first in the preprocessed file, it will need the other which is defined later. In this situation both source files would have to be modified, so that `player.h` includes `song_declaration.h` before defining the `Player` class and `song_definition.cpp` afterwards.

When using the concept-based formulation above, `player.cpp` will be compiled first, then `song.cpp`. During the compilation of `player.cpp`, some low-level information on `Song` will be missing[1] (for example, `sizeof(Song)`) and must be resolved when the two module parts are linked together. The linker must check the dependencies between the entities declared in the various module parts, to ensure that there is no cycle in the dependencies.

In the last example, the types `Player` and `Song` don't depend on each other, since neither one has a field with the other's type. Instead, the following situation should cause a linker error:

```
// a.h
concept IA<A> {
    class A {
        ...
    };
}
```

```
// b.h
concept IB<B> {
    class B {
        ...
    };
}
```

[1] All entities in `player.cpp` have to be compiled as if all of them were code constrained with the export clause of `Song`.

```
// a.cpp
#include "a.h"

export {
    IA<A>;
}

import "b.cpp";

class A {
    ...
    B x;
};
```

```
// b.cpp
#include "b.h"

export {
    IB<B>;
}

import "a.cpp";

class B {
    ...
    A x;
    int n;
}
```

The problem is the cyclic dependency between the types `A` and `B`, that would define a type with an infinite `sizeof`, because there would be an infinite number of fields `n` in any object of type `A` or `B`. We only check the dependencies between types: methods don't induce such a dependency, since a method can depend on a type but a type can't depend on a method. A method may call another method, but this should not be flagged as an error, since we don't need to look at a method's body in order to typecheck the caller and generate the machine code for the call.

We have distinguished plain `#include`s from `import`s, but we can replace all includes with corresponding `import`s of the same file, if we add as a rule that `import`-ing a file with no `export` block is equivalent to the `#include` of that file. This may not be intuitive, so in this paper we stick to `#include`s.

9.3 Extension B: avoiding unnecessary rebuilds

The module syntax and semantic proposed in this section has a downside, compared with typical `.h`/`.cpp` splitting, especially for non-templated code: since the module source is a single file, a change in the implementation produces a rebuild of all the code that imports the module, unlike the previous situation where only changes to header files generate rebuilds of other translation units.

A naïve way to fix this would be to split the module source, putting the part up to and including the `export` block in a header file, and the rest in an implementation file. Unfortunately, this approach doesn't work, since the `export` clause by itself gives too few information to be able to compile the code that uses this module. For example, the size of exported types is also needed. A better solution is to have the compiler do the splitting when compiling the module, generating a "module interface" file that contains:

- The API of the `export` clause.
- The definition of the concepts involved in the `export` clause.
- The implementation of the template functions (constrained and non-constrained) named in the `export` clause, and the ones directly or indirectly instantiated by them.
- The implementation of inline functions (the ones that the compiler decides to inline, not necessarily all the ones marked as `inline`) exposed by the `export` clause.
- The size of types.
- Information on which classes have virtual methods, and on such methods (even if they are not exposed in the API of the module).
- The value of relevant `constexpr` variables and the implementation of constexpr functions and methods.
- Optionally, additional information useful to the optimizer (as suggested in the current C++ modules proposal). For example, whether or not a function writes to global memory.

If a module interface file already exists, the compiler should only update it if the current version is different. This prevents tools such as `make` from noticing a different modification time and rebuilding dependent TUs. In order for such tools to work, three kind of compile rules are needed:

- The compiled module (say, `my_stack.o`) should depend on the module source (say, `my_stack.cpp`), and the compiler should be invoked to update it, when needed. Such invocation may also change the module interface file, say `my_stack.mi`, but `make` doesn't know this.

- The code that imports the module should depend on `my_stack.mi`.
- `my_stack.mi` should depend on `my_stack.o`, but no command should be run because of this.

The third rule is a workaround to prevent problems — without it, `make` may compile the client code before (or during) the generation of `my_stack.o`, since the client code would not depend on it. By having an empty command, the correct order is used. Note that this does *not* produce an infinite loop: even if the `.mi` file remains older than the corresponding `.o` file, `make` will only run the empty command once, and then assume that the module interface is up to date.

Currently, the information in the module interface file is always recalculated based on the header file. Such recalculations increase the compile time of dependent TUs, and there is often no way in C++ of just specifying them without having to specify several additional ones. For example, lacking a way to specify the just size of a type in the header file as a number of bytes (for good reasons), the programmer is forced to also expose the private fields and recursively include the headers that define the relevant types. Such information is unnecessary, and produces repeated parsing and typechecking of the code in header files.

So, this extension doesn't just bring back the amount of recompilation to the current situation without modules, but is actually an improvement even if compared with the current situation.

Another advantage of this approach is that the programmer specifies all the code in one source file, unlike the current practice where the programmer has to remember which parts of the code have to be in the header file, and move the code between the header and the implementation file when doing certain changes (for example, templating code or marking a function as inline).

From another point of view, this transparent recompilation is a downside: it will be more difficult for the programmer to predict how much rebuilding will derive from her latest changes, even though it will be at most as much as in the current C++ without modules.

9.4 Discussion

For the above reasons, we think that modules should be combined with concepts. Extension B makes concept-based modules more useful and greatly reduces the amount of code rebuilt, so we think it should be used.

About extension A, we are uncertain whether or not the added complexity in the language is worth the fix of this not-extremely-widespread issue. More research and discussion is needed on this topic.

10. Inner requirements

In many concept proposals (e.g., [GS-06]), concepts maps are used as adapters between the syntax used in the concept and the actual syntax of the implementation.

For example[1]:

```
concept Shape<shape> {
    ...
    double area(const shape &);
}

concept ShapeWithGet<shape> {
    ...
    double getArea(const shape&);
}

// From a third-party library.
template <ShapeWithGet shape>
void do_work(shape s);

template <Shape shape>
concept_map ShapeWithGet<shape> {
    double getArea(const shape & s) {
        return area(s);
    }
}

template <Shape shape>
void f(shape s) {
    do_work(s);
}
```

The above code can be rewritten as:

[1] We use the `concept_map` keyword here to clarify the code, since the reader may otherwise have difficulties parsing the code, due to the body of the concept map.

```
concept Shape<shape> {
    ...
    double area(const shape &);
}
concept ShapeWithGet<shape> {
    ...
    double getArea(const shape&);
}

// From a third-party library.
template <ShapeWithGet shape>
void do_work(shape s);

concept ShapeAdapter<shape> {
    Shape<shape>;
    ShapeWithGet<shape>;
    double getArea(const shape& s) {
        return area(s);
    }
}

template <Shape shape>
ShapeAdapter<shape>;

template <Shape shape>
void f(shape s) {
    do_work(s);
}
```

Since `ShapeAdapter` requires `ShapeWithGet`, the concept map is also a concept map for `ShapeWithGet`. The implementation is now moved into a concept from the concept map body.

However, in both formulations of the code there is still a problem: we are requiring every class that models `Shape` to also model `ShapeWithGet` (using the given implementations of methods in `ShapeWithGet` in terms of the ones in `Shape`). This may seem a theorem that can be proven, as long as the semantic requirements are the same after the replacement of the methods with the adapted ones, but it's not: a model of `Shape` may have additional methods that are not specified in the `Shape` concept and whose behavior is not specified. In this case, if one of the methods happens to be named `getArea`, and it has a different semantics than `area`, then the above concept map is stating a false property. Of course, we must *not* add false concept maps, since the compiler will consider them safe assumptions even though they aren't, generating semantic errors at runtime.

To address this issue, we will now introduce a new feature, that we call *inner requirements*. It's syntactically similar to the feature proposed in

section 3.4.7 of [GS-06], but there are significant semantic differences.

Using this feature, we can write the above code as follows:

```
concept Shape<shape> {
    ...
    double area(const shape &);
}

concept ShapeWithGet<shape> {
    ...
    double getArea(const shape&);
}

// From a third-party library.
template <ShapeWithGet shape>
void do_work(shape s);

concept ShapeAdapter<shape> {
    Shape<shape>;
    ShapeWithGet<shape>;

    double getArea(const shape& s) {
        return area(s);
    }
}

template <Shape shape>
void f(shape s) {
    requires { ShapeAdapter<shape>; }
    do_work(s);
}
```

If the adapter is used only once, we can write it directly in the inner requirement, as follows:

```
concept Shape<shape> {
    ...
    double area(const shape &);
}

concept ShapeWithGet<shape> {
    ...
    double getArea(const shape&);
}

// From a third-party library.
template <ShapeWithGet shape>
void do_work(shape s);
```

```
template <Shape shape>
void f(shape s) {
    requires {
        ShapeWithGet<shape>;
        double getArea(const shape& s) {
            return area(s);
        }
    }

    do_work(s);
}
```

An inner requirement (in this case for `ShapeAdapter<shape>`) is a statement that the specification holds *in the current scope*, unlike a concept map, that would be used when it holds in the global scope. Note that this is different — when we instantiate the above function with a type `T`, only the entities in the API of `Shape` are visible in the function body, so we know exactly which entities are available and which aren't, unlike a concept map in the global scope.

In our example, if a model of `Shape` already has a `getArea` method with different semantic, this is not a problem here: such method is hidden, since the requirements of `ShapeAdapter` are only checked with respect to the API provided by `Shape` (and the enclosing scope, that in this case doesn't add any entities), not with respect to the concrete type with which `f` will be instantiated.

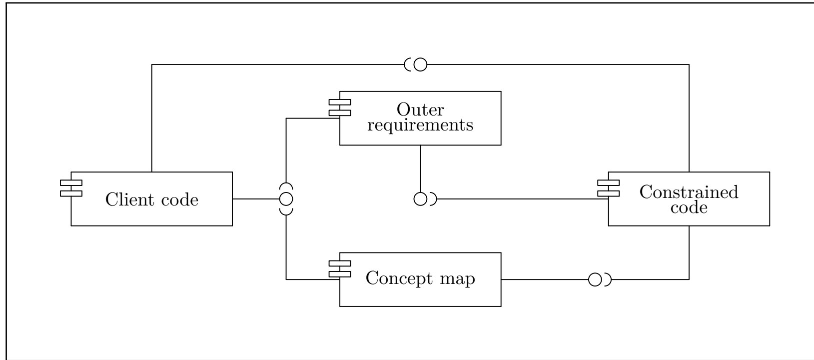
Instead of saying that every model of `Shape` also models `ShapeWithGet` under the translation, we are now saying that the API of the `Shape` concept satisfies the requirements of `ShapeWithGet` (under the translation). We can guarantee that this is the case, as long as the semantic requirements are consistent with respect to the given translation.

When typechecking a inner requirement, the compiler checks that the syntactic requirements of `C` are satisfied in the current scope, considering both the enclosing scope of the definition and the API of the requires clause, and then assumes that the semantic requirements of `C` are implied. In the example above, this means that if `ShapeWithGet` had stricter semantic constraints than `Shape`, the requires clause would be stating a false property. It is a duty of the programmer writing the inner requirement to ensure that this is not the case; the compiler is unable to detect this error, since it's an undecidable property. For syntax adaptation, when the semantic requirements are the same and the adapter is just renaming entities, this is trivially the case.

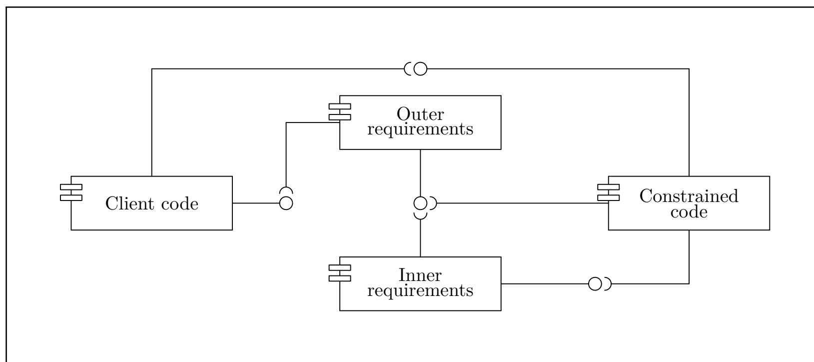
After the inner requirement passed typechecking, the following code is typechecked as if `C` was added to the concept requirements of the function.

The API of `C` is then available to the following code, and this is what allows the call to `do_work` in the example above to typecheck.

The following diagram models the first two approaches:



While the last two can be modeled as follows:



This feature should not be confused with the one with similar syntax in [GS-06], that introduces late-checked requirements and would break separate typechecking.

Using the terminology of the theory of institutions (see [GB-92]), inner requirements allow the definition of arbitrary signature morphisms. Note that in order to use morphisms that change the meaning of some entities we may have to wrap the operation twice. For example:

```
concept A<T> {
    // Returns 0.
    T f();
    // Returns 1.
    T g();
}
concept B<T> {
    T zero();
    T one();
}
```

```
concept C<T> {
    // Returns 1.
    T f();
    // Returns 0.
    T g();
}
concept A_to_B_Adapter<T> {
    A<T>;
    B<T>;
    T zero() {
        return f();
    }
    T one() {
        return g();
    }
}
concept B_to_C_Adapter<T> {
    B<T>;
    C<T>;
    T f() {
        return one();
    }
    T g() {
        return zero();
    }
}

// From a third-party library.
template <C T>
void process_C(T x);

template <B T>
void process_B(T x) {
    requires { B_to_C_Adapter<T>; }
    process_C(x);
}
template <A T>
void process_A(T x) {
    requires { A_to_B_Adapter<T>; }
    process_B(x);
}
```

10.1 Discussion

We think that the above feature is very useful for syntax adaptation, and it avoids the wrong semantic assumptions that would be done if concept maps were used instead.

10.2 Interaction with constexpr constraints

When using constexpr constraints together with inner requirements, we need a way to integrate syntactic adaptation with the constraints. Consider this example:

```
// Third party library
concept G<T> {
    class T {
        constexpr T();
    }
    constexpr bool g(T);
    g(T());
    ...
}

template <typename T>
requires { G<T> }
void impl();
```

```
// Client code

// The same as G, just using h instead of g.
concept H<T> {
    class T {
        constexpr T();
    }
    constexpr bool h(T);
    h(T());
    ...
}

concept H_to_G_adapter<T> {
    H<T>;
    G<T>;
    constexpr bool g(T x) {
        return h(x);
    }
}

template <typename T>
requires { H<T>; }
void public_function() {
    requires { H_to_G_adapter<T>; }
    impl();
}
```

The requirement `G<T>` of `impl` is satisfied by the inner requirement. However, without additional rules the syntactic requirements of the adapter would not be satisfied by `H<T>`, so `public_function` won't typecheck. The issue is that the adapter requires `h(T())`, that is satisfied by `H<T>`, but also `g(T())` that isn't.

We need to integrate the two features. A possible solution is to modify the check of constexpr constraints so that when the definition of a constexpr function definition is available in the requirements or in the enclosing scope, we will expand the definition before the syntactical comparison of constexprs. If the available definitions form a loop (this is allowed when at least one of them is an in-concept definition), the whole loop is ignored.[2]

In our example, `g(T())` is expanded to `h(T())`, the matching succeeds and `public_function` now passes typechecking.

[2] Trying to break such loops by removing an edge, say the last encountered definition, would produce a maintenance nightmare since swapping the order of two requirements in a concept would be enough to break the code using such concept.

11. Concept-based overloading

From the first concept papers (for example [Str-03]), an extended overloading based on concepts has been proposed. This means that a type or operation can be declared several times with different concept requirements and the right declaration will be chosen for each call site, depending on which requirements are satisfied. An interesting discussion on this topic can be found in [JWL-04].

11.1 Approaches

There are two main approaches for concept-based overloading: one using a Not concept combinator and one testing for weaker requirements.

Weaker requirements

In this approach, before doing the usual overload resolution, the requirements of the available entities are checked and the entities whose requirements don't hold are discarded. Afterwards, it may still be needed to disambiguate between different entities with the same signature by doing additional checks (for example, choosing the one with the weakest or strongest requirements) before the usual overload resolution.

In order to get the maximum performance, it's important to choose the entity with the strongest requirements. On the other hand, when we need to implicitly assume some requirements (using implicit concepts) if we don't choose the overload with the weakest requirements, we may make semantic assumptions that are not satisfied.

When using implicit concepts with optional warnings, we can use a mix of the above approaches: among the concepts whose syntactic requirements match, we prefer a requirement `C` over `D` in the following three cases:

- `C` implies `D` and we know that `C` is satisfied.
- `C` implies `D` and the syntactic requirements of `C` are strictly stronger than the ones of `D`.
- `D` implies `C`, but we don't know whether `D` is satisfied and the syntactic requirements of `D` aren't strictly stronger than the ones of `C`.

Knowing that a concept is satisfied means that there is a matching concept map[1] or that the concept has no semantic requirements. Note that even a concept with no axioms is considered as having semantic requirements, unless it's marked `auto`.

The check for stricter syntactic requirements allows us to prefer `BidirectionalIterator` to `InputIterator` (because the former also requires decrements), but not `ForwardIterator` to `InputIterator`, that have the same syntactic requirements.

This is done in the same spirit of [Str-09]'s explicit refinements, but the compiler automatically compares the syntactic requirements and the programmer doesn't have to annotate the refinements.

The solution above doesn't guarantee that, when there are multiple entities satisfying the requirements, one with maximum (or minimum) requirements exists. We talk about minimal and maximal from the point of view of guarantees, so "smaller" requirement means weaker.

If there is a set of entities with maximal (or, respectively, minimal) requirements, in general we can't choose the best one without further information from the programmer. This condition will result in an ambiguity error.

Using Not as a concept combinator

Another solution (adopted for example in [Str-03]) uses the Not combinator for concepts, so that we can make the compiler discard an implementation when a better one is available. This is called `enable_if`-style overloading in the C++ literature, because it can be emulated in C++11 using the `std::enable_if` class provided by the standard library. For example:

```
concept A<T> {
    ...
}
concept B<T> {
    ...
}

template <B T>
requires { !A<T>; }
void f(T); // Slow implementation

template <A T>
void f(T); // Fast implementation
```

[1] Note that a template concept map can allow the use of type traits and tags as if they were concept maps, we'll see an example later on in this chapter.

In this case the behavior of concept-based overloading is much simpler: if there is more than one viable overload then the compiler rejects the call as ambiguous, with no need to compare the respective requirements.

Discussion

Using Not a concept combinator in general has several problems, as we argued in chapter 5. We could introduce a limited version of Not just for the purpose of disambiguating concept-based overloads, but we would lose extensibility compared to the first approach. For example, if a slow implementation is provided by a third-party library and we wrote a faster one with stricter requirements, we wouldn't be able to disambiguate them because we would need to modify the `requires` clause of the slower `f`, and this is not possible since it's provided by a third party.

For this reason, we favor the first approach for concept-based overloading, that tests for weaker requirements and otherwise reports an ambiguity.

With implicit concepts, this approach produces a performance regression when tag-based overloading using `std::enable_if` is converted to concept-constrained code. For example, the overloads requiring `InputIterator` will always be preferred to the ones requiring `ForwardIterator` because the assumption is weaker and there are no further syntactic requirements.

To convert such code to concept-constrained overloading, the compiler must be informed that the tags carry semantic information, using templated concept maps. For example, to specify the iterator categories the STL can provide these concept maps[2]:

```
template <typename I>
requires {
    ... // The syntactic requirements of InputIterator

    typename I::iterator_category = std::input_iterator_tag;
}
InputIterator<I>;

template <typename I>
requires {
    ... // The syntactic requirements of ForwardIterator

    typename I::iterator_category = std::forward_iterator_tag;
}
ForwardIterator<I>;

...
```

[2] In fact we need two sets of concept maps, distinguishing the mutable iterator concepts from the non-mutable ones (see the definitions of these concepts in B.4).

Since the syntactic requirements and guarantees of each concept map are the same, the STL is not required to implement the iterator APIs for every `T` (which would be impossible) — each concept map is matched by its own requires clause. Similar concept maps can be found in [SGJ+-05]. The main difference in the above concept maps is that we need to require the syntactic requirements of the concepts, while in [SGJ+-05] they are not present. Since [SGJ+-05] doesn't describe how to check templated concept maps, it's not clear if this is due to a different way of checking them or is an error in the formalization.

With these concept maps, specifying a tag in an iterator type will also result in a concept map for the respective concept, and the concept-based overload resolution will then work as expected with implicit concepts.

Concept maps like the above should become the preferred way to convert `enable_if`-style overloading into concept-based overloading. We expect programmers able to write code using `enable_if` to have advanced knowledge of C++, enough to understand the translation into concept maps.

The transition allows the use of the overloaded function in concept-constrained templates. We don't break backward compatibility, so the code using `enable_if` doesn't have to be rewritten for any other reason (such as switching to a concept-enabled compiler and/or standard library, upgrading a library to a version using concepts, etc.).

11.2 Guideline for concept-based overloading

Given two concept-based overloads of the same function, say O_1 and O_2, with requirements R_1 and R_2, where R_2 is at least as strict as R_1 (i.e., R_2 implies R_1), then O_2 must have a precondition that implies the one of O_1 and a postcondition that is implied by the one of O_1.

In other words, adding a concept map that results in O_2 being selected instead of O_1 must be guaranteed not to break code.

For example, given the following two implementations of `f`:

```
template <InputIterator I>
void f(I first, I last) {
    ...
}
template <BidirectionalIterator I>
void f(I first, I last) {
    ...
}
```

Then the latter implementation must satisfy the precondition and postcondition of the former.

We disagree with [JWL-04] that criticizes the weaker precondition of the overloads of `std::advance` that have stricter requirements on the iterator type. We believe that stricter concept requirements mix well with weaker preconditions (providing stronger guarantees) and stronger postconditions.

The same reasoning applies to performance: overloads with stricter requirements that are expected to be faster may turn out to be actually slower. The performance benefit can only be guaranteed by specifying the complexity of both APIs in the concepts and using explicit concepts. With implicit concepts, there may be performance regressions due to (implicit) complexity assumptions that turn out not to hold.

11.3 Concept-based overloading for types

How does the guideline explained in the previous section apply to types? As for functions, we must guarantee that adding a concept map that allows the overload resolution to pick a more specialized implementation does not break the following code. Given two implementations of a class:

```
template <MoveConstructible T>
class vector {
    ...
}
template <Regular T>
class vector {
    ...
}
```

Any code that expects to use the first implementation and, due to a newly added concept map, switches to the second one should not break. Let's consider this use of the two implementations of `vector` above:

```
// Library 1
class X {
    ...
};
MoveConstructible<X>;

vector<X> f();

// Client code
// New concept map.
// Regular<X>;

vector<X> g() {
    return f();
}
```

When `f` and `g` are typechecked, `X` is only known to be `MoveConstructible`, so the first implementation of `vector` is chosen.

If the concept map is uncommented, then the return type of `g` changes to the second implementation above. We end up having two `vector<X>` types in the same program and, in general, to guarante that `g` keeps working we must require the two types to be the same. Therefore, we conclude that the language should *not* allow concept-based overloading for types at all. The constructors, destructor and methods of a class can still use concept-based overloading (just like global functions) but fields, inherited classes and virtual functions must be the same.[3]

In the C++ literature, concept-based overloading is traditionally used for both functions and types, see for example [GS-06].

11.4 Ambiguous calls

We may expect that following the above guideline for concept-based overloading of functions and forbidding concept-based overloading of types guarantees that adding a concept map (i.e. extending the API of a library) will never break client code. While they do guarantee that the client code doesn't break at run-time, they don't prevent the introduction of compilation errors, due to ambiguous calls.

This is not a problem introduced by concepts — in C++ calling an overloaded function doesn't just require the function to exist, but also that there is no ambiguity between the available overloads.

When using concept-based overloads, this problem is emphasized further. For example, consider the following situation:

```
// common.h
concept Foo<T> {
    ...
}
concept Bar<T> {
    ...
}
// Most general implementation.
template <Regular T>
void f() {
    ...
}
```

[3] Only concept-based overloading/specialization of types has this issue. The usual template specialization doesn't, because it's resolved based on the concrete types, not just a partial specification. So it can still be used for, e.g, `vector<bool>` even in a concept-enabled STL.

```
// library_A.h
#include <common.h>

// Optimized implementation for Foos.
template <Foo T>
void f() {
    ...
}
```

```
// library_B.h
#include <common.h>

// Optimized implementation for Bars.
template <Bar T>
void f() {
    ...
}
```

```
// library_C.h
#include <common.h>

class my_class {
    ...
};

Foo<my_class>;
```

```
// client.cpp
#include <library_A.h>
#include <library_B.h>
#include <library_C.h>

void do_stuff() {
    f<my_class>();
}
```

In this case, `do_stuff` calls the version of `f` specialized for `Foos` (since we know from the concept map in `library_C.h` that `my_class` is a `Foo`), and the typechecking succeeds.

However, if the library C is modified adding a concept map `Bar<my_class>`, then the call becomes ambiguous because both optimized implementations are available and there's no reason to choose one over the other. Note that the libraries A and B don't depend on each other, so neither of them can expect this issue.

Comparison with conventional overload resolution

Ambiguities in concept-based overload resolution suffer from the same integration issues as usual C++ overloading. Consider the following situation:

```
// common.h
class X {
    ...
};
```

```
// library_A.h
#include "common.h"

class Y {
    ...
    Y(const X&);
};

void f(const Y&);
```

```
// library_B.h
#include "common.h"

class Z {
    ...
    // Z(const X&);
};

void f(const Z&);
```

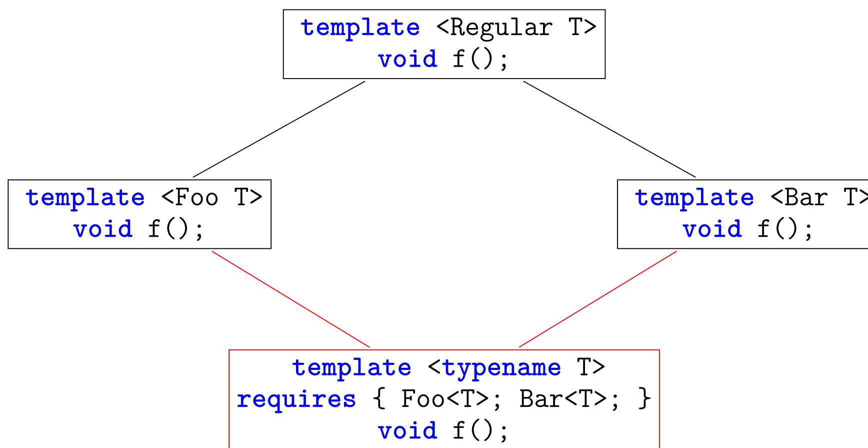
```
// client.h
#include "library_A.h"
#include "library_B.h"

void do_stuff(const X& x) {
    f(x);
}
```

The above code typechecks correctly. However, if we uncomment the constructor for `Z` that takes a `const X&`, then the call becomes ambiguous, since we can convert `x` either to a `Y` or to a `Z`.

As with the problem in concept-based overloading, since the library B does not depend on A, it doesn't know that adding the constructor will break client code, so this may seem a safe extension of the API of B.

11.5 Ambiguities as lack of a meet

If we consider the partial ordering relation "has stricter or equivalent requirements" (see [JGW+-06]), we can visualize the concept-based overloads of a function with a Hasse diagram. The example at the beginning of section 11.4 can be modeled as follows:



The red part is the *meet* of the two ambiguous overloads, defined as an overload with the conjunction of the requirements.[4] The ambiguity in the overload resolution is due to the lack of the implementation of such meet. [JGW+-06] refers to the meets as «tie-breaking overloads».

A function using concept-based overloading is guaranteed not to give rise to ambiguities if every pair of concept-based overloads has a meet. However, in some cases the meet can't be defined because the two requirements aren't composable. This is not a problem as long as there is no scope that satisfies both requirements.

We obtain the following completeness condition for the conjunction of requirements: if there is a scope that satisfies both requirements, then they must be composable. For example, a requirement for a type `foo` and a requirement for a function `foo` don't have to be composable, since no scope can satisfy both. On the other hand, a requirement for a function `int foo();` and one for a function `float foo();` *must* be composable, because we allow conversions in the return type during matching, so a scope can satisfy both. In this case, the calls to `foo` under such a constraint will be ambiguous (we can't overload on the return type), and the constrained

[4] More precisely, it's the meet of the two overloads in the partial order containing all possible function declarations, not just the ones in scope.

code will have to call a helper function requiring only one of them in order to pick the correct overload.

```
requires {
    int foo();
}
void do_work_int() {
    ...
    foo(); // Ok.
}

requires {
    int foo();
    float foo();
}
void do_work() {
    foo();          // Error, ambiguous.
    do_work_int(); // Ok.
}
```

Every library exposing multiple implementation of the same function using concept-based overloading should provide all the meets that don't result in contradictory requirements. Due to the completeness condition above, syntactically contradictory requirements are guaranteed not to be satisfied in any scope. Semantically contradictory requirements will result in an ambiguity. In this case the ambiguity is useful, since it can only happen due to contradictory concept maps and since there is no reasonable way to implement the meet, as in the example below.

```
concept StrictWeakOrdering<lt, T> {
    ...
}
concept TotalPreorder<leq, T> {
    ...
}
template <typename T>
requires { StrictWeakOrdering<less, T>; }
void sort_with_less(vector<T>& v) {
    ...
}
template <typename T>
requires { TotalPreorder<less, T>; }
void sort_with_less(vector<T>& v) {
    ...
}
template <typename T>
requires {
    StrictWeakOrdering<less, T>;
```

```
    TotalPreorder<less, T>;
}
void sort_with_less(vector<T>& v) {
    // There is no reasonable way to implement this function,
    // since no call can satisfy the requirements. When they
    // are not satisfied we can't provide a working
    // implementation anyway.
}
```

The client libraries can extend the set of overloads and provide the same guarantee, by implementing all non-contradictory meets.

Combining two independent libraries that provide overloads for the same general function, however, can still generate ambiguities because none of the two libraries has the duty to provide the meet of its overloads with the ones provided by the other library.

11.6 Discussion

We don't think that's possible to avoid the ambiguity problem, except by completely forbidding concept-based overloading, and we don't think we should drop such an important feature just for fear of ambiguities.

If every library follows the guideline for function overloads and provides all non-contradictory meets, ambiguities can only happen when there are contradictory concept maps or due to integration of concept-based overloads from different libraries.

Using `enable_if`-style overloading does *not* prevent the ambiguities by itself, the integration problem is less serious only because concept-based overloads in different libraries would be less frequent due to the extensibility issue.

When all the overloads are defined in the same library (as long as the meets are also defined) there is no risk of ambiguities anyway. The lack of meets can be compared with the presence of two available overloads with non-disjoint requirements in the `enable_if` approach.

Using concept-based modules (as discussed in chapter 9) makes the integration problem less serious: the user can still get ambiguity errors due to incompatible libraries in his own code, but can't get any such errors in the code of the modules he uses. Without modules, such errors can also happen in any included header (even of third party libraries), caused by a concept map that was included previously in the same translation unit and that the third party library doesn't know about.

11.7 Concept-based specialization

The name "concept-based overloading" is also used for a related, but different, language feature that we call "concept-based specialization", as in [SR-05] and [JGW+-06]. This feature is also related to the "dynamic overloading" in [JWL-04].

As the name suggests, concept-based overloading is resolved based on the concept requirements and scope (including concept maps) at the call site, while concept-based specialization happens when the generic function is instantiated on concrete types, based on the scope that triggers such instantiation.

Concept-based specialization is a formalization of template specialization, as concept-based overloading formalizes the overloading of template functions. Due to ADL, template function overloading (unlike plain overloading) can access some functions that are only in scope at the point of instantiation.

Advantages

At instantiation time all relevant concept maps are known. This additional information may allow to resolve concept-based overloads to more specialized implementations. For example, consider the following code[5]:

```
template <InputIterator I>
void advance(I& itr, I::difference_type n) {
    ... // O(n)
}
template <RandomAccessIterator I>
void advance(I& itr, I::difference_type n) {
    ... // O(1)
}

template <InputIterator I>
void f(I itr, I::difference_type n) {
    advance(itr, n);
    ...
}
template <RandomAccessIterator I>
void g(I itr) {
    ...
    f(itr, I::difference_type(100));
}
```

[5] The C++ standard also specifies a third version of `advance` that requires `BidirectionalIterator`. We ignore it here because it's not relevant to our discussion.

Using concept-based overloading, when `f` is typechecked the call to `advance` is resolved to the slow version, since `I` is only required to be an `InputIterator`. When `f` is called by `g`, even though `g` knows that `I` is a `RandomAccessIterator`, the slower version of `advance` will be executed within `f`.

On the other hand, using concept-based specialization means that the call to `advance` within `f` won't be fully resolved until the concrete types are known. The concrete type used as `I` must be a `RandomAccessIterator` (since `g` requires it) and the faster `advance` will always be called.

Simulating concept-based specialization

If we don't include concept-based specialization as a language feature, we can get a similar behavior by propagating information backwards through the instantiation chain, up to the caller's requires clause. [JWL-04] contains a similar example, but uses optional constraints instead of the Or operator. `optional C` is equivalent to `{} || {C}` using our syntax. We prefer not to add a new keyword for this purpose.

Assuming that `f` knows about the two specializations of `advance` and expects that choosing the right specialization would impact the performance of `f` itself, the above `f` can be rewritten as follows:

```
template <typename I>
requires { InputIterator<I> || RandomAccessIterator<I>; }
void f(I itr, I::difference_type n) {
    advance(itr, n);
    ... // O(sqrt(n)) work.
}
```

Note that the Or operator is the one based on syntactic sugar and concept-based overloading described in chapter 5. In fact, we could have used Xor here, since `RandomAccessIterator` implies `InputIterator`, but we prefer to use Or since it conveys the intuition that all meets are also defined.

In this case, the documentation of `f` can state that the complexity is $O(n)$ for `InputIterators` and, say, $O(\sqrt{n})$ for `RandomAccessIterators`.

The downside of this approach is that the programmer needs to explicitly specify suitable optional requirements to match the available specializations. In deep instantiation chains, this can be very verbose.

Also, a function that calls k functions using concept-based specialization, where the i-th function has n_i specializations (excluding the most general definition), the programmer has to specify "only" $\sum_i n_i$ constraints, but the compiler must typecheck the function $\prod_i n_i$ times, with each combination of constraints. This can slow down the compilation of such code.

The ideal would be to have concept-based specialization that:

- Doesn't force the programmer to explicitly specify optional requirements
- Avoids the combinatorial explosion of compilation time by typechecking the function once and generating code only for the combinations of specializations that are actually used.
- Maintains separate typechecking, even though the function is not typechecked for all combinations.
- Selects the strictest specialization whose requirements are satisfied.

Considering new specializations

Trying to mimic the behavior of ADL for unconstrained templates, we may decide to exploit the full information available at instantiation time: both properties of the concrete types (definitions and concept maps) and additional concept-based specializations. Maximizing the amount of information available means that the most efficient specializations will be used. However, this approach has some issues.

A minor issue is that concept-based specialization together with implicit concepts may result in *less* specialized overloads being selected. A simple solution to this problem is to collect the implicitly-generated concept maps during the instantiation of the constrained templates and add them to the instantiation scope, guaranteeing that the instantiation scope as at least as much information as any constrained template in the instantiation chain.

The following code shows a much more important issue:

```
template <Regular T>
void f() { ... }

template <Regular T>
requires { FooAble<T>; BarAble<T>; }
void g(T x) {
    f(x);
}

template <Regular T>
requires { FooAble<T>; }
void f() { ... }
template <Regular T>
requires { BarAble<T>; }
void f() { ... }
```

Here, even though at the point of definition of `g` the call to `f` doesn't seem to produce any issues (and doesn't even seem to be specialized), due

to the two later concept-based specializations any call to `g` will produce an ambiguity error inside `g`. Therefore, this approach breaks separate typechecking.

Disregarding new specializations

To keep separate typechecking, we can decide that the set of candidate specializations is determined when the call is typechecked, and the only information propagated from the instantiation scope is the properties of the concrete types (including matching concept maps). Note that this means decreased performance when the best specialization is not visible.

With this interpretation the last example compiles fine, since the last two overloads of `f` are not visible at the point of definition of `g`, so they are not considered.

When typechecking `g` we need to ensure that, no matter what concepts are satisfied by the concrete types, the call to `f` will have at least a candidate and won't be ambiguous. We can assume that the requirements of the current function are satisfied, otherwise the instantiation wouldn't be allowed.

We will call *candidate overloads* the set of concept-based overloads for the called function whose requirements are satisfied in the current scope (considering concept requirements and declarations and concept maps in the outer scope).

The first step of overloading resolution is to determine the strictest overload in the set of candidate overloads. If there is an ambiguity, it is flagged as an error. Otherwise, we consider the set of *candidate specializations*, containing the concept-based specializations of the chosen overload (note that this set contains none of the original candidate overloads).

Then, we check that every pair of candidate specializations whose requirements aren't (syntactically) contradictory has a meet in the same set.

After these checks we are guaranteed that no ambiguities can happen at instantiation time and we can use separate typechecking.

The most expensive part of this algorithm is the check for meets, that has a trivial $O(n^3)$ implementation (where n is the number of candidate specializations). [JGW+-06] describes an algorithm taking $O(2^k)$ time, where k is the number of involved concepts. While our algorithm is polynomial on n, it can still be exponential in the size of the source code, since a function definition requiring the Or of m concepts is expanded in $O(2^m)$ candidate specializations.

Interaction with modules

When concept-based specialization is used to resolve a function call in a module, we have two separate scopes available: the instantiation scope and the module scope. The declarations and concept maps for implementation details of the module may conflict with the ones in the instantiation scope; in general we can't just concatenate the two scopes. Also, even when this is possible it is semantically wrong: if a type `my_type` is defined both in the module (as implementation detail, without being exported) and in the instantiation scope, a concept map `Regular<my_type>` in one scope doesn't imply that both types are regular.

Extracting from the instantiation scope "only the requirements relevant to the template parameters" (in some way) is not a solution: the generic function may have been instantiated with `my_type`.

Let's try the dual approach: instead of propagating properties through the instantiation chain and then matching them to the requirements of the available specializations, we can propagate the specializations' requirement backwards through the instantiation chain and do the matching in the scope that triggered the instantiation. For example, given the following code:

```
template <InputIterator I>
void advance(I& itr, I::difference_type n) {
    ... // O(n)
}
template <RandomAccessIterator I>
void advance(I& itr, I::difference_type n) {
    ... // O(1)
}

template <InputIterator Itr>
void f(Itr itr, Itr::difference_type n) {
    advance(itr, n);
    ...
}
template <RandomAccessIterator I>
void g(I itr) {
    ...
    f(itr, I::difference_type(100));
}
```

The call to `advance` within `f` resolves to the first definition and the second one is also visible as a specialization. Template parameter deduction is run and `advance<Itr>` is determined to be the matching instantiation.

The most general definition is typechecked as usual, ensuring that its requirements are implied by the requirements of `f`. The requirements clause

of the specialization, after replacing the template parameters (in this case, `I`) with the types deduced for the instantiation (in this case, `Itr`) is stored inside the typechecked version of `f`. Inside the compiler, it will be stored as something like[6]:

```
template <InputIterator Itr>
requires { optional(RandomAccessIterator<Itr>); }
void f(Itr itr, Itr::difference_type n) {
    ((RandomAccessIterator<Itr>) ?
        advance__1<Itr>
    :
        advance__0<Itr>
    )
    (itr, n);
    ...
}
```

When `g` is typechecked, the implementation of `f` may not be visible (due to separate typechecking) so in general we don't know the optional requirements of `f`. The implementation of `g` will only advertise its own optional requirements (in this case, none).

At instantiation time, all implementations are available. Only then the optional requirements are collected from all the constrained templates that need to be instantiated and propagated according to the dependencies between the various templates and to the template parameters of the instantiations. In our example, at this stage `g` will inherit the optional requirements of `f`, with the replacement `I=Itr`.

After the propagation of optional requirements, the requirements of the outermost function are checked in the instantiation scope.

For example, the instantiation `g(v.begin())`, with `v` a `vector<int>` triggers the check of `RandomAccessIterator<vector<int>::iterator>` (in the calling scope). Then this information is propagated to the functions instantiated by `g` and will be used for instantiation.

In the next chapter we'll see a more low-level description of how this feature can be implemented in a compiler.

Interaction with inner requirements

Inner requirements allow the use of entities with requirements that are not satisfied by the function's requirements. In this case, we would have to "reverse" the syntactic adaptation in the inner requirement to map the

[6] We use `?:` and `optional` with concepts for presentation purposes only, we are not considering them as language features.

called function's optional requirements to optional requirements using the original syntax. Clearly, this is in general not possible.

Consider the example already used in chapter 10:

```
concept Shape<shape> {
    double area(const shape&);
}
concept ShapeWithGet<shape> {
    double getArea(const shape&);
}
// From a third-party library.
template <ShapeWithGet shape>
void do_work(const shape& s);

concept ShapeAdapter<shape> {
    Shape<shape>;
    ShapeWithGet<shape>;
    double getArea(const shape& s) {
        return area(s);
    }
}
template <Shape shape>
void f(shape s) {
    requires { ShapeAdapter<shape>; }
    do_work(s);
}
```

If `do_work` has an optional requirement, say `PolygonWithGet`, we would like to map it to the corresponding requirement for the original syntax, say `Polygon`. This can only be done manually by the programmer, since both reversing the syntactic adaptations and finding a corresponding concept (if any) aren't feasible.

For functions with inner requirements the programmer will have to manually specify the optional requirements using Or, if the performance of `f` would otherwise be affected.

The optional requirements of `do_work` (if any) will be checked against the information available in `f` (that is, `Shape` and `ShapeAdapter`). If they are satisfied, the relevant specializations will be used. In any case, a constrained function with an inner requirement, such as `f`, won't have optional requirements. These checks will happen at instantiation time, since the implementation of `do_work` may not be available until then.

12. Implementing concepts

In this chapter we explain a way to implement concept-constrained code by encoding it as unconstrained templates. This is not the only possible implementation, and probably an ad-hoc implementation would have a better performance. However, we think that explaining such an implementation is still useful, firstly because it shows that a relatively simple implementation is indeed possible (with no need of dynamic dispatch), and also to shed more light on the meaning of concept-constrained code, beyond type checking.

From a design viewpoint, this implementation embodies our intuition of concepts and helps to explain the design choices made in previous chapters. We use an extension of the dictionary-passing approach.

The choice of template parameters in the encoding is not a minor detail: since concept-based constraints are evaluated depending on the enclosing scope (and can therefore resolve differently in different parts of the same translation unit) it's important to identify what is matched by such constraints, so that when the same entities are matched in succeeding instantiations we can detect it and re-use the previous instantiation. This saves compilation time and reduces the size of the resulting executable.

Dynamic dispatch may lead to a simpler implementation, but comes with a run-time performance penalty that we want to avoid, so we don't consider that option.

12.1 Flattening concepts

We start by introducing a "flattening" operation on concept requirements, that will be used in the rest of the chapter. Basically, flattening a concept expands all concept definitions (except those in requires clauses) and drops axioms, default definitions and same-type constraints.

We explain the behavior of this operation informally, showing how to flatten `VectorTemplate<my_vector>`, with the following concept definitions[1]:

[1]These concepts are just for explanation purposes. They are simplified versions of some STL concepts.

```
concept EqualityComparable<T> {
    typename T;
    bool operator==(const T&, const T&);
    bool operator!=(const T& x, const T& y) {
        return !(x == y);
    }
    axiom (const T& x, const T& y, const T& z) {
        x == x;
        if (x == y)
            y == x;
        if (x == y && y == z)
            x == z;
    }
}
concept Vector<vector> {
    class vector {
        typename value_type;

        template <InputIterator I>
        requires { ConvertibleTo<I::value_type, value_type>;}
        vector(I, I);

        template <RandomAccessIterator I>
        requires { ConvertibleTo<I::value_type, value_type>;}
        vector(I, I);

        requires { EqualityComparable<value_type>; }
        EqualityComparable<vector>;
    }
}
concept VectorTemplate<vector> {
    template <CopyConstructible T>
    Vector<vector<T> >;
}
```

We start by taking the definition of `VectorTemplate` and replacing the arguments used in the instantiation.

```
template <CopyConstructible T>
Vector<my_vector<T> >;
```

Now we have a concept requirement and we expand it using the definition of `Vector` and the equivalence rules for concept APIs defined in chapter 6. We will once again use the `forall` notation, with the same meaning.

```
forall <CopyConstructible T>
class my_vector<T> {
    typename value_type;

    template <InputIterator I>
    requires { ConvertibleTo<I::value_type, value_type>; }
    my_vector(I, I);

    template <RandomAccessIterator I>
    requires { ConvertibleTo<I::value_type, value_type>; }
    my_vector(I, I);

    requires { EqualityComparable<value_type>; }
    EqualityComparable<my_vector<T> >;
}
```

We now rewrite the code removing the pseudo-class-definition syntax and distributing the outer `forall` and `requires` clause.

```
forall <CopyConstructible T>
typename my_vector<T>;

forall <CopyConstructible T>
typename my_vector<T>::value_type;

forall <CopyConstructible T, InputIterator I>
requires {
    ConvertibleTo<I::value_type, my_vector<T>::value_type>;
}
my_vector<T>::my_vector<I>(I, I);

forall <CopyConstructible T, RandomAccessIterator I>
requires {
    ConvertibleTo<I::value_type, my_vector<T>::value_type>;
}
my_vector<T>::my_vector<I>(I, I);

forall <CopyConstructible T>
requires {
    EqualityComparable<my_vector<T>::value_type>;
}
EqualityComparable<my_vector<T> >;
```

We can now flatten the requirement for `EqualityComparable`. Note that we drop the axiom and the definition of `!=` in terms of `==`.

```
forall <CopyConstructible T>
typename my_vector<T>;

forall <CopyConstructible T>
typename my_vector<T>::value_type;

forall <CopyConstructible T, InputIterator I>
requires {
    ConvertibleTo<I::value_type, my_vector<T>::value_type>;
}
my_vector<T>::my_vector<I>(I, I);

forall <CopyConstructible T, RandomAccessIterator I>
requires {
    ConvertibleTo<I::value_type, my_vector<T>::value_type>;
}
my_vector<T>::my_vector<I>(I, I);

forall <CopyConstructible T>
requires {
    EqualityComparable<my_vector<T>::value_type>;
}
typename my_vector<T>;

forall <CopyConstructible T>
requires {
    EqualityComparable<my_vector<T>::value_type>;
}
bool operator==<T>(const my_vector<T>&, const my_vector<T>&);

forall <CopyConstructible T>
requires {
    EqualityComparable<my_vector<T>::value_type>;
}
bool operator!=<T>(const my_vector<T>&, const my_vector<T>&);
```

At this point we have a list of simple requirements. We can remove duplicated requirements with the same requires clause (in this case, none) and, for types, also duplicate requirements with stronger requirements (in this case, the second requirement for `my_vector<T>`).

We keep the two requirements for the constructor because they may match different implementations — in this case, we expect the one requiring `RandomAccessIterator` to be more efficient.

```
forall <CopyConstructible T>
typename my_vector<T>;

forall <CopyConstructible T>
typename my_vector<T>::value_type;

forall <CopyConstructible T, InputIterator I>
requires {
    ConvertibleTo<I::value_type, my_vector<T>::value_type>;
}
my_vector<T>::my_vector<I>(I, I);

forall <CopyConstructible T, RandomAccessIterator I>
requires {
    ConvertibleTo<I::value_type, my_vector<T>::value_type>;
}
my_vector<T>::my_vector<I>(I, I);

forall <CopyConstructible T>
requires {
    EqualityComparable<my_vector<T>::value_type>;
}
bool operator==<T>(const my_vector<T>&, const my_vector<T>&);

forall <CopyConstructible T>
requires {
    EqualityComparable<my_vector<T>::value_type>;
}
bool operator!=<T>(const my_vector<T>&, const my_vector<T>&);
```

This is the final result of the flattening algorithm.

12.2 Converting concept requirements into template parameters

In this section we'll see how a requires clause can be converted to a list of template parameters.

Associated types are treated just like additional template parameters when encoding constrained code. They do differ when such constrained code is instantiated, though.

The steps are the following:

- Flatten the requires clause
- For each simple requirement in the list, recursively convert the concept requirements (if any) into template parameters and use them to declare a template template parameter using a unique name.

We need template template parameters, so we can only encode into types. Functions will be encoded as static methods of these types. Let's use `EqualityComparable<T>` as an example. After flattening, we get:

```
typename T;
bool operator==(const T&, const T&);
bool operator!=(const T&, const T&);
```

We have no concept requirements, so we can directly generate the corresponding template parameters:

```
<typename T1, // T
 typename T2, // contains operator== as static operator()
 typename T3> // contains operator!= as static operator()
```

In this way, we can start bottom-up and then convert all the requirements of any requires clause. Let's consider this requirement:

```
requires { EqualityComparable<my_vector<T>::value_type>; }
bool operator==(const my_vector<T>&, const my_vector<T>&);
```

Converting the requires clause yields the above `T1`, `T2`, and `T3`. This means that we can convert the whole requirement into:

```
<template<typename T1, typename T2, typename T3> class T4>
```

12.3 Transforming constrained templates into unconstrained ones

We can use this conversion to transform concept constrained code (after typechecking) into unconstrained templates. We will define functions as the `operator()` of a type, to match what we do in the concept requirements. For example, this code:

```
// We assume that the following are in scope:
template <typename T> requires { }
size_t my_vector<T>::size() const;
template <typename T> requires { }
const T& my_vector<T>::operator[](size_t) const;

template <EqualityComparable T>
bool operator==(const my_vector<T>& x,
                const my_vector<T>& y) {
    if (x.size() != y.size())
        return false;
    for (size_t i = 0; i < x.size(); i++)
        if (x[i] != y[i])
            return false;
    return true;
}
template <EqualityComparable T>
bool operator!=(const my_vector<T>& x,
                const my_vector<T>& y) {
    return !(x == y);
}
```

Can be converted to:

```
template <typename T>
struct __outer0; // size()
template <typename T>
struct __outer1; // operator[]
template <typename T1, typename T2, typename T3>
struct __outer2 {
    static bool operator()(const T1& x, const T1& y) {
        if (__outer0<T1>::operator()(x)
            != __outer0<T1>::operator()(y))
            return false;
        for (size_t i = 0; i < __outer0<T1>::operator()(x);
             i++)
            if (T3::operator()(
                        __outer1<T1>::operator()(x, i),
                        __outer1<T1>::operator()(y, i)))
                return false;
        return true;
    }
}
template <typename T1, typename T2, typename T3>
struct __outer3 {
    static bool operator()(const T1& x, const T1& y) {
        return !__outer2<T1, T2, T3>::operator(x, y);
    }
}
```

Note the difference between entities provided by the requires clause (that now use `T1`, `T2` and `T3`) and the ones that were resolved to outer functions (that now instantiate the encodings of such functions).

The matching between the template parameters of the encoding and the template parameters to use in the various instantiations is derived from the matching of concept requirements done by the typechecker. For example, the call to `operator==` succeeds because its requirements (`EqualityComparable<T>`) are satisfied in that scope, due to the same requirement of `operator!=`. This matching done by the typechecker is also used to pass the correct template parameters.

We can use this encoding for declarations of constrained functions and types, and also for entities introduced by a concept map (when there are any conversions involved or the requires clause is different). In this way, when we instantiate a constrained function the typechecker will match the requirements to the available concept maps, and we can pass the encodings of the entities in the concept map as template parameters of the encoding of the constrained function. For example, consider the following code:

```
template <T>
requires { }
class my_vector {
    ...  // Including size(), operator[]
}

// As above
template <EqualityComparable T>
bool operator==(const my_vector<T>& x,
                const my_vector<T>& y) {
    ...
}

template <EqualityComparable T>
EqualityComparable<my_vector<T> >;

// This is a builtin (implicit) concept map.
EqualityComparable<int>;

void do_work() {
    my_vector<int> x, y;
    x == y;
}
```

The following encodings are generated:

```
template <typename T>
struct __outer;  // my_vector<T>
template <typename T>
struct __outer0; // my_vector<T>::size()
template <typename T>
struct __outer1; // my_vector<T>::operator[]
... // Encoding of the rest of my_vector.

// As above, generated from the defn of operator==.
template <typename T1, typename T2, typename T3>
struct __outer2 { ... };
// As above, generated from the concept map for
// EqualityComparable using the in-concept defn.
template <typename T1, typename T2, typename T3>
struct __outer3 { ... };

// Encoding of EqualityComparable<int>
struct __outer4 {
    static bool operator()(const int& x, const int& y) {
        return x == y;
    }
}
struct __outer5 {
    static bool operator()(const int& x, const int& y) {
        return x != y;
    }
}

// Encoding of do_work()
struct __outer6 {
    static void operator() {
        __outer<int> x, y;
        __outer2<__outer<int>, __outer4, __outer5>
            ::operator()(x, y);
    }
}
```

Note how the equality between vectors is encoded in the composition of `__outer2` with `__outer<int>`, `__outer4` and `__outer5`, in the same way as the two concept maps together guarantee the existence of equality on vectors.

Avoiding duplicates

We should avoid generating duplicate encodings of functions, so that we can uniquely map each function signature (including the template and requires clauses) selected by overload resolution into its encoding. We can do the same for types, the only difference is that the instantiation is specified

explicitly by the programmer, rather than deduced by the typechecker, and concept-based overloading doesn't apply.

This removal of duplicates can only be done for concept maps in the global scope (or in a namespace scope). Concept maps in other scopes are not guaranteed to match the same entities as later concept maps, even if they are identical.

With the features that we picked, the only other scope where concept maps are allowed is inner requirements. A function definition containing an inner requirement, such as:

```
template <typename T1, ..., typename Tk>
requires { C }
void f() {
    requires { D }

    ...
}
```

Is encoded as if it was written as follows:

```
template <typename T1, ..., typename Tk>
requires { C }
D;

template <typename T1, ..., typename Tk>
requires { C }
void f() {
    ...
}
```

Except that the encoding of the concept map will be used only for this function.

Handling concept-based specialization

The implicit optional requirements deriving from the use of concept-based specialization can be encoded as boolean template parameters. The mapping of requirements to these boolean parameters needs to be retained, since they are not exposed in the prototypes and they are needed at instantiation time to choose the right specialization.

Supporting concept-based specialization makes the encoding of constrained templates within the compiler more complex, since they can only be propagated to the callers at instantiation time (since the implementation of the called functions may not be available during typechecking).

The code that we intuitively wrote in the previous chapter as:

```
template <InputIterator Itr>
requires { optional(RandomAccessIterator<Itr>); }
void f(Itr itr, Itr::difference_type n) {
    ((RandomAccessIterator<Itr>) ?
        advance__1<Itr>
    :
        advance__0<Itr>
    )
    (itr, n);
    ...
}
```

Will be initially encoded as:

```
template <// Encoding of InputIterator<Itr>
          typename T0, typename T1, ...,
          // Encoding of RandomAccessIterator<Itr>
          typename U0, typename U1, ...,
          // Encodes whether RandomAccessIterator<Itr> holds
          bool b0>
class __outer0 {
    static void operator()(T0 itr, T1 n) {
        (b0 ?
            advance__1<U0, U1, ...>::operator()
        :
            advance__0<T0, T1, ...>::operator()
        )
        (itr, n);
        ...
    }
}
```

We need to encode optional requirements too, since they often contain an extended API (as in this case).

However this is not enough, since we are only encoding the function's own optional requirements and not the ones of the called functions. A function `g` calling `f`, such as:

```
template <RandomAccessIterator I>
void g(I itr) {
    ...
    f(itr, I::difference_type(100));
}
```

Can only be encoded with respect to the prototype of `f` during type-checking, so the encoding won't take into account the additional template parameters of `f` due to optional requirements. The encoding will look like this:

```
template <// Encoding of RandomAccessIterator<Itr>
          typename T0, typename T1, ...>
class __outer1 {
    static void operator()(T0 itr) {
        ...
        __outer0<T0, T1, ...>::operator()(itr, T1(100));
    }
}
```

At instantiation time, after propagation of the optional requirements in the prototypes, we need to update the encodings starting with the leaves of the tree of instantiation. In our example we process `advance`, then `f` and finally `g`. The encoding of `g` is extended with the encoding of the optional constraints inherited from `f`, and becomes:

```
template <// Encoding of RandomAccessIterator<Itr>
          typename T0, typename T1, ...,
          // Encoding of RandomAccessIterator<Itr>
          typename U0, typename U1, ...,
          // Encodes whether RandomAccessIterator<Itr> holds
          bool b0>
class __outer2 {
    static void operator()(T0 itr) {
        ...
        __outer0<T0, T1, ..., U0, U1, ..., b0>
            ::operator()(itr, T1(100));
    }
}
```

The new name of the encoding (that we wrote as `__outer2`) must be uniquely identified by the prototype of `g` together with the list of optional constraints inherited from the instantiated functions.

In this example, the encodings contains repeated encodings of `RandomAccessIterator<Itr>` that a smart compiler can decide to detect and de-duplicate as an optimization. We don't do this in our examples because we want to show what happens in general.

The code that calls `g` will test the optional requirements and pass the results in the corresponding boolean parameters. In this case, `true` will always be passed since any typechecking call of `g` guarantees `RandomAccessIterator`.

When an optional requirements is false, the template parameters encoding its API can be instantiated with any type of the right kind, they won't be used anyway as long as the conditional `?:` operators involving the boolean parameters are evaluated at compile time, dropping the corresponding "branches" without actually generating code for such instantiations.

13. Additional features

This chapter describes several additional concept features, that don't involve an extensive analysis of the design choices, but can often be very useful as a complement to the features discussed in the other chapters.

13.1 Existential quantifiers on types

In some cases, we want to specify some properties of a return type. For example, suppose that we want to specify the minimal concept requirements for the following function:

```
template <typename T>
int f(const T& x) {
   g(x).getInt();
}
```

The corresponding concept may look like this:

```
concept C<T> {
     typename U = ???;
     U g(const T&);
     int U::getInt() const;
}
```

But it's unclear how we should initialize the type `U`. In this case, we want to expose some name in the provided signature (so that `g` and `getInt` can be called from `f`), but not in the requirements on the scope. This is quite similar to the use of complex expressions in use patterns, but a bit more powerful: since `U` has now a name (even if just local to the current concept) we can add requirements like `CopyConstructible<U>;` inside `C`.

Note that, even though we have to expose some type in the signature provided by the concept, we probably should expose a unique, internal name that the client code can't refer to, much like the type used by `auto` variables. Due to this similarity to `auto`, we could use a syntax like the following:

```
concept C<T> {
    typename U = auto;
    U g(const T&);
    int U::getInt() const;
}
```

Every `typename U = auto` is resolved, when the concept is instantiated on concrete types, based on the return type of the first operation returning `U` that appears in the concept. If there is no such operation, it's an error.

Note that in this case, the first operation will not have a conversion in the return type, while the following ones (if any) can. So the order of the constraints now matters.

Extended variant

This could be extended even further, by allowing arbitrary type expressions together with `auto`, not just return types. For example, we could allow this:

```
concept C<T> {
    auto typename U = decltype(X(5));
    int U::getInt() const;
}
```

and interpret the expression in the scope enclosing the concept map, rather than in the concept itself. So we don't define `X` in the concept, and it may be anything: a type (so `X(5)` is a call to the constructor), a function (so `X(5)` is a function call) or a value (so `X(5)` is a call to `operator()(int)`).

Refinements

A requirement for a function returning an existentially-qualified type can be refined by a stricter one, that uses an actual type. Given the following concepts:

```
concept WeakBinaryOperation<Op, T> {
    class Op {
        typename result_type = auto;
        result_type operator()(T, T);
        SomeConcept<result_type>;
    }
}

concept BinaryOperation<Op, T> {
    WeakBinaryOperation<Op, T>;
    class Op {
        T operator()(T, T);
    }
}
```

The API of `BinaryOperation` should contain only one function, because its requirement refines the one in `WeakBinaryOperation`. The compiler will remove the version returning the existentially-qualified type and replace the occurrences of such type in the API of the weaker concept with the actual type.

In the above example, this means that `BinaryOperation` now requires `SomeConcept<T>`.

Discussion

A more detailed analysis of existential types is needed, both on the semantics and in the choice of a good syntax. The goal of this chapter is to make this feature explicit — in the usage patterns syntax it was implicit, while in the pseudo-definitions approach it was completely missing.

13.2 Rvalue references and overload resolution

In [Gre-08], Gregor explains why concepts will produce a performance loss, unless specific measures are put in place. For example, consider the following code (a variant of the code in [Gre-08], modified to fit with the rest of this thesis):

```
concept String<S> {
    class S {
        S();
        S(const S&);
    };
    S operator+(const S&, const S&);
}
template <String S>
void f() {
    S x;
    S y = x + S() + S();
}
class string {
    string();
    string(const string&);
    string(string&&);
};
string operator+(const S&, const S&);
string operator+(S&&, const S&);
String<string>;
```

The `string` class is a `String`, but also provides a version of `operator+` that takes a rvalue reference and is more efficient than the one that takes a `const string&`.

However, the code that requires `String<S>` has only access to the less-efficient signature, and that overload is chosen before `S` is resolved to be `string`, so that the more efficient version is not used.

Discussion

[Gre-08] proposes a solution that delays part of the overload resolution to instantiation-time, so that the more efficient version can be chosen. However, this may result in a instantiation-time error, due to an ambiguity in the overload resolution or to the presence of a deleted constructor or method.

Instantiation-time errors are one of the main problems that concepts try to solve, so allowing them would make concept-checking weaker, breaking the separate typechecking property.

Instead, we think that we should modify the semantics of operation signatures in concepts so that a type `T` in a signature means that both `const T&` and `T&&` arguments must be allowed in that position. A `const T&` or `T&&` type in a signature, on the other hand, only requires parameters with that type leaving unspecified whether a parameter with the other type is allowed or not.

```
concept String<S> {
    class S {
        S();
        S(S);
    };
    void foo(S, S);
}

class string {
    string();
    string(const string&);
    // This would make the concept map fail.
    // string(string&&) = delete;
};

void foo(const S&, S&&) {...}
void foo(S&&, const S&) {...}

// Without this, the concept map would fail, since a call to
// foo with two S&& parameters would be ambiguous.
void foo(S&&, S&&) {...}

String<string>;
```

There is an apparent problem with this semantics: the first overload of `foo` alone satisfies the concept map, while together with the second it

doesn't (unless the third is also specified). This would lead to the problem that moving the concept map between the two declarations (and removing the third) will guarantee that `String<string>` is satisfied, while it actually isn't.

Consider the following code:

```
concept IFoo<S> {
    class S {
        ...
    };
    void foo(S, S);
}

class string {
    ...
};

void foo(const S&, S&&) {...}

String<string>;

void foo(S&&, const S&) {...} // Allowed?
```

It seems that we allow to break a concept map after it has been checked. However, this is prevented if we are using the feature of concept maps extending visibility, since after the concept map an overload of `foo` that takes a `S&&` and a `const S&` will be visible, so the second definition is flagged as an error (double definition).

Another way to think about this semantics is to think of the requirement for `foo` in the concept as syntactic sugar for 4 requirements (with all combinations of `const S&` and `S&&`), that will often map to the same implementation when the concept is instantiated with concrete types, but not necessarily. Using this interpretation, it's apparent that the implementation of this feature within a compiler is trivial (just syntactic sugar).

However, a requirement with n such parameters generates $O(2^n)$ precise requirements, so a compiler may want to represent them implicitly (and then handle this implicit representation in the typechecker) as an optimization, to decrease compilation time.

This feature is somewhat related to the rewriting of `T` as `const T&` in concepts proposed in [GS-06].

13.3 Passing overloaded functions to templates

In C++ a plain function can be passed to templates requiring a functor, since the function's name is implicitly convertible to a function pointer and that pointer can then be called inside the template, with the usual syntax.

```
std::string to_upper(const std::string& s);

void do_stuff() {
  vector<std::string> v;
  ...
  vector<std::string> v2;
  transform(v.begin(), v.end(), back_inserter(v2), to_upper);
}
```

However, when the function is overloaded this doesn't work anymore. The programmer must disambiguate the reference at the call site, it doesn't happen inside the template where the compiler would have enough information to do the disambiguation itself.

For example, if the `to_upper` function in the above code is provided by a third-party library, and that library switches to C++11 adding an overload for `to_upper` taking a rvalue reference, the `do_stuff` function will break.

We want to move the overload resolution from the caller to inside the template. This is already possible in C++11, using a functor class and perfect forwarding.

```
std::string to_upper(const std::string& s);
std::string to_upper(std::string&& s);

struct ToUpper {
    template <typename T>
    auto operator()(T&& x)
          -> decltype(to_upper(std::forward<T>(x))) {
        return to_upper(std::forward<T>(x));
    }
};

void do_stuff() {
  vector<std::string> v;
  ...
  vector<std::string> v2;
  transform(v.begin(), v.end(), back_inserter(v2), ToUpper());
}
```

If we expect libraries to add overloads, especially for rvalue references (and this is likely to happen), we should use this trick, or at least an explicit cast to the desired function signature. In order to take full advantage of

rvalue references, the overload resolution should be done as late as possible, so that we know which overload to choose and we get the best performance. Unfortunately, this approach is too verbose: most projects aren't going to write a functor class just for these reasons.

Since the ambiguity of the overload currently produces a compilation error, we could change the language in order to resolve the ambiguity by using a functor similar to the one above. Also, the compiler can add conversion operators to make the functor implicitly convertible to each of the function types that it can resolve to. Note that, to preserve backward compatibility, non-ambiguous overloads must still be resolved as a plain function.

We can extend this feature to operators, as long as the compiler considers the builtin operations as "overloads" of `operator+`, `operator-`, etc.

Consider the following code:

```
concept String<string> {
    typename string;
    string to_upper(string);
    ...
}

template <String string>
void do_stuff() {
  vector<string> v;
  ...
  vector<string> v2;
  transform(v.begin(), v.end(), back_inserter(v2), to_upper);
}
```

Here we wrote only one requirement for `to_upper`, but the compiler generates two split versions (one for const references and one for rvalue references) due to the feature in the previous section. This makes overload resolution fail unless we use the functor-based approach.

Discussion

This feature is useful by itself, and becomes even more important, almost compulsory, together with the one explained in the previous section — otherwise we would get unexpected ambiguities. For these reasons, we suggest its inclusion, and we'll use it in the following.

A similar idea was discussed in [SR-03], in turn inspired by the original template proposal paper.

13.4 Loops of function definitions

The problem

The in-concept default definitions for functions are useful to factor code common to various models of the concept. However, their use can produce loops of function definitions. Consider the following code:

```
// Library A
concept A<T> {
    void f(const T&);
    void g(const T& x) {
        f(x);
    }
}
```

```
// Library B
concept B<T> {
    void g(const T&);
    void f(const T& x) {
        g(x);
    }
}
```

```
// Client code using A, B.
concept C<T> {
    A<T>;
    B<T>;
}
...
C<int>;
```

Two independent libraries contain similar concepts, stating that `f` and `g` exist and that they are equivalent.

The concept `A` requires the programmer to define either just `f` or both `f` and `g`. On the other hand, the concept `B` requires the definition of either just `g` or both `f` and `g`.

For this reason, the two concepts are *not* equivalent. There are no conflicting semantic nor syntactic requirements, so they should still be compatible.

If the developer of the client code implements at least one of `f` and `g` before the concept map, then the relevant function definition is picked from `A` or `B`. However, if none of the two definitions is implemented both default definitions are used and, with no compiler errors or warnings, we end up having two mutually recursive functions.

If the definition in one of the two libraries (say, B) has just been introduced by the latest version, replacing an axiom stating the equivalence, the client code will silently break at run-time, even though there is no clear responsible of the problem. The two libraries may not know of each other, and are separately safe to use. On the other hand, the client *can* detect such an error by reviewing its own code together with the changes in B, but in large codebases this is likely to be caught only during the testing phase (if at all).

We think such situations should be detected by the compiler and result in a compilation error.

A solution

It's not possible to detect such problems before the concept map. In the example above, the concept `C` is not always problematic, but only when neither `f` or `g` is defined before the concept map.

Forbidding the concept `C` itself would limit composability of concepts — if the client code wants to allow the call of functions provided by both libraries on a type `T`, then the specification of the type `T` must contain the conjunction of the concepts `A` and `B`.

We can only detect this issue by having the compiler check for loops when a concept map is processed. We will determine the in-concept type and function definitions to use as follows:

- Collect all the definitions inside the concept used in the concept map.
- Drop the definitions of all entities that are already defined in the scope before the concept map (including definitions that have stricter requirements compared to the already existing ones).
- Typecheck the in-concept definitions, constructing a dependency graph for function definitions (see below).
- If there are any cycles, report an error (definition loop).
- For every prototype, if there is a definition coming from a concept C1 and one coming from a concept C2, with C2 a direct or indirect (and possibly templated) concept requirement of C1, drop the definition defined in C2.
- If there are still prototypes having multiple definitions, report an error (ambiguous definition).
- Otherwise, all remaining definitions are added to the current scope.

The dependency graph is a graph with two kinds of nodes: function prototypes (including their `template` and `requires` clauses) and function definitions. Every function definition has an edge towards each function it calls and a single incoming edge, from the respective function prototype.

An escape hatch

In some limited cases, mutual recursion between functions is not an error and doesn't give rise to unbounded recursion. For example, consider the following:

```
concept A<> {
    void f(int);
    // Expects a non-negative integer.
    void g(int);

    void f(int x) {
        if (x >= 0)
            g(x);
        else {
            // Handle negative ints directly.
            ...  // (no calls to g here)
        }
    }
}
concept B<> {
    void f(int);
    // Expects a non-negative integer.
    void g(int);

    void g(int x) {
        if (x < 0) {
            // Negative values will be handled by f instead.
            f(x);
            return;
        }
        ... // (no calls to f here)
    }
}
concept C<> {
    A<>;
    B<>;
}
C<>;
```

Even though using both definitions is actually safe, `f` and `g` depend on each other, so the looping detection explained earlier will give a false positive.

In practice, mutually-recursive functions need careful inspection to ensure termination, and are likely to be defined outside the concept instead of using in-concept definitions. However, we think that an escape hatch for these situations should be provided, providing a way for the programmer to state that mutual recursion is desired.

For this purpose we can introduce a builtin concept (say, `MutuallyRecursive`) or an attribute (say, `[[mutually_recursive]]`).

Relationship with IDE support for concepts

The looping detection allows the definition of concepts containing loops of definitions, as long as the programmer implements one of the involved functions before the concept map, thus breaking the loop.

We think that this capability should be used scarcely, when it's necessary for backward compatibility or integration of different concept-aware libraries. We do *not* encourage developers to write all conceivable definitions "just in case" because this, while it provides a slight benefit to the client code, limits the possible IDE support.

When there are no definition loops in the concept itself, an IDE can use a concept map to determine which functions and types remain to be implemented and generate the corresponding stubs. On the other hand, when there are loops in the in-concept definitions, the IDE will only know that at least one of the functions need to be implemented, and will have to either generate all the stubs or ask the developer for guidance.

13.5 Properties

Axioms provide a way to express universally-quantified properties of types and operations, but have no way to specify properties of specific values. For example, consider the `sort` function from the STL that takes a comparator:

```
template <typename RandomAccessIterator, typename Compare>
void sort(RandomAccessIterator first,
          RandomAccessIterator last,
          Compare comp);
```

We expect to concept-constrain such function as follows:

```
template <RandomAccessIterator Itr,
          StictWeakOrder<Itr::value_type> C>
void sort(Itr first, Itr last, C comp);
```

With a `StictWeakOrder` concept like:

```
concept StictWeakOrder<C, T> {
    typename T;
    class C {
        bool operator()(T, T) const;
    }
    // ???
}
```

We can't specify the semantic properties of the strict weak order in the concept, because such axioms would mean that all objects of type `C` satisfy the properties, while `sort` only requires them to be satisfied for the specific object used in the call. For instance, if we specify a function pointer as the functor, we only have to ensure that that function defines a strict weak ordering, rather than all functions with that type (this would make no sense). Still, that property is a part of the API of `sort`, and we would like to formalize it. [SS-12] introduces a C++ formalization of this notion of *property*, also used (but not as code) in [SM-09].

In the following, we will combine properties (using a syntax similar to [SS-12]) with a possible C++ syntax for preconditions, that are written informally in that paper. As [SS-12] notes, this feature can be introduced independently from concepts; we cover it in this chapter because it's needed to be able to fully specify semantic properties with concepts.

```
template <typename T, BinaryPredicate<T> R>
property reflexive(R r) {
    for all (T a)
        r(a, a);
}
template <typename T, BinaryPredicate<T> R>
property symmetric(R r) {
    for all (T a, T b)
        if (r(a, b))
            r(b, a);
}
template <typename T, BinaryPredicate<T> R>
property transitive(R r) {
    for all (T a, T b, T c)
        if (r(a, b) && r(b, c))
            r(a, c);
}
template <typename T, BinaryPredicate<T> R>
property equivalence_relation(R r) {
    reflexive<T, R>(r);
    symmetric<T, R>(r);
    transitive<T, R>(r);
}
```

```
template <typename T, BinaryPredicate<T> R>
property irreflexive(R r) {
    for all (T a)
        !r(a, a);
}
template <typename T, BinaryPredicate<T> R>
property asymmetric(R r) {
    for all (T a, T b)
        if (r(a, b))
            !r(b, a);
}
template <typename T, BinaryPredicate<T> R>
property strictWeakOrder(R r) {
    irreflexive<T, R>(r);
    asymmetric<T, R>(r);
    transitive<T, R>(r);
    auto eq = [&r](T x, T y) {
        return !r(x, y) && !r(y, x);
    };
    equivalence_relation<T, decltype(eq)>(eq);
}
```

The above code defines a property `strictWeakOrder` that can be used as the precondition of `sort`. The declaration with the precondition can be written as:

```
template <RandomAccessIterator Itr,
          BinaryPredicate<Itr::value_type> C>
precondition(iteratorRange(first, last)
             && strictWeakOrder<Itr::value_type, C>(comp))
void sort(Itr first, Itr last, C comp);
```

Note the use of the `iteratorRange` property to complete the semantic requirements.

13.6 Non-definable properties

Many preconditions that can't be defined in code can be defined with universally quantified statements as done in the previous section. However, some of them just can't be defined — for example an `dereferenceable()` predicate for iterators. Even though they can't be defined, these properties are still very useful in preconditions, so we propose to allow non-defined properties, as in the following:

```
template <Iterator Itr>
property dereferenceable(Itr itr);
```

A definition will never be required. This is not a problem, since properties aren't executed anyway.

Adding this feature allows to specify non-expressible properties, but at the same time may encourage the programmer to never define the properties, decreasing the quality of the API specification and preventing the use of proof or testing tools that, unlike the compiler, actually need the code implementing axioms and preconditions.

Anyway, we think that the pros outweigh the cons, since a programmer is free to not specify the precondition, and this is even worse than using non-defined properties.

13.7 Integrating axioms and properties

If both axioms and properties are introduced, we should at least consider having an uniform syntax.

For example, we can replace `for all` with `axiom`, as in the following:

```
template <typename T, BinaryPredicate<T> R>
property equivalence_relation(R r) {
    axiom reflexive(T a) {
        r(a, a);
    }
    axiom symmetric(T a, T b) {
        if (r(a, b))
            r(b, a);
    }
    axiom transitive(T a, T b, T c) {
        if (r(a, b) && r(b, c))
            r(a, c);
    }
}
```

We'll use this syntax for the STL concepts in appendix B.

Since axioms don't have to be executable, we allow the "call" of properties in axioms, as if they were functions returning `bool`. This allows axioms to express things like "itr is dereferenceable" even though such a function often can't actually be implemented for most iterator types.

13.8 Integrating preconditions in requires clauses

Some papers propose the specification of preconditions in axioms, inside `requires` clauses. However, we don't think that this should be done, since the compiler has to parse (1) template arguments and requirements, (2)

function arguments and (3) preconditions in this order. Merging steps (1) and (3) will result in a loop, with associated parsing issues.

For example, consider the following code:

```
template <typename F>
requires {
    axiom {
        x(y);
    }
    class F {
        typename argument_type;
        void operator()(T);
    }
}
void f(F::argument_type x, F y);
```

The compiler can't start typechecking the `requires` clause before the function arguments, since it would be unclear whether `x(y)` is a valid expression (since `x` and `y` are not yet defined, and the types are not known).

On the other hand, typechecking the arguments first means checking that `F::argument_type` is a type, which is only known after processing the `requires` clause.

14. Conclusion

We have analyzed and compared several concept features, both from previous papers and from new ideas. We hope that this thesis will give new insights on concepts for C++, giving rise to a constructive discussion of these features (and possibly some related ones as well) and eventually leading to a carefully designed concept syntax.

In this respect, we think that the feature of implicit concepts with optional warnings/errors presented in chapter 8 is a significant step toward an agreement between the two research groups working on C++ concepts, since it covers both use cases.

14.1 Usability of explicit concepts

We already argued that not all C++ projects will want to use explicit concepts, and we must allow such projects to use implicit concepts instead.

[SR-05] is concerned about concept maps, and argues why (at least in their concept design) «explicit assertion of concepts is unmanageable. The use of explicit asserts simply doesn't scale — even using techniques such as assertions of templated predicates»[1].

We believe that the use of explicit concepts becomes feasible when using the concept features proposed in this thesis, for projects that want a stricter typechecking (often at the price of conciseness).

In this section we try to address the concerns of [SR-05], showing how they can be solved using our concept design. Note that this does *not* by itself show that our design works "well enough" with explicit concepts — there may still be other issues that are not addressed, or even additional issues that have been introduced. Any feedback on this point is greatly appreciated.

Who has the duty to write concept maps

A concept map is the formalization of statement. For the purpose of deciding who has the duty of writing such a concept map, we will distinguish the "subject(s)" of the claim from the other parameters.

[1] [SR-05] refers to concept maps as "concept assertions".

Usually the subject is the first parameter, but in some cases there could be several (again, usually at the beginning of the parameter list) or even none if the concept has no parameters at all.

For example, the concept map `Vector<my_vector, int>` can be read as "`my_vector` is a `Vector` of `int`s", and the subject is `my_vector`. This is a subjective interpretation: the compiler treats all parameters in the same way.

In rare cases a concept may actually have two (or more) subjects: consider for example a concept describing a pair of types, respectively implementing a smart pointer and a smart reference type of `T`s. It's not possible to split this in a concept for the smart pointer and one for the smart reference without losing the operations involving both.

Also, in some cases there is no clear subject: `ConvertibleTo<T, U>` is in some cases used as a "property" of `T` and in others as a "property" of `U`. In this case, we will treat both as subjects.

With this notion of "subjects", we define a guideline for concept maps (in an explicit concept setting): *a concept map should be written in the first place where both the concept and the subjects are available.* Every entity exposed by a library should be specified with concept maps by the same library. This is the same idea behind concept-based modules.

For example, `Vector<std::vector<T>, T>` must be provided by the standard library because it defines both `Vector` and `std::vector<T>`. Since the type `T` is not known, a templated concept map must be used, requiring the relevant properties of the type `T`.

This policy addresses the problem for which [SR-05] considers the use of a specific feature called "Encapsulated assertions", the main difference is that we use template concept maps instead.

The use of implication as concept combinator allows us to express constraints such as "`std::vector<T>` is `EqualityComparable` as long as `T` is also `EqualityComparable`" directly in the `Vector` concept.

For template functions, typically the requirements are properties of the types used as arguments, so the user has the duty to write those concept maps for the types used for the function call.

Need to expose constraints used in the implementation

[SR-05] expresses the concern that using explicit concept would produce a leak of constraints used internally in the implementation (and that the user doesn't need to care about) into the public interface of the library. A systematic use of the above guideline, together with the *lack* of a Not operator, the use of constrained requirements within concepts, and of inner

requirements within functions (see chapter 10) prevents this from happening. Every definition exported to users of the library can use the same requires clause that would be used with implicit concepts, without advertising internal concept-based specialization (hidden by forbidding the use of Not) syntactic adaptations and internal concepts (both hidden by using inner requirements).

This example is called «A worst case example» in [SR-05]:

```
concept Small<T, n> {
    sizeof(T) <= n;
}
const int max = 200;

template <Small<max> T>
void f(const T&);

template <typename T>
requires { !Small<T, max>; }
void f(const T&);

template <typename T>
void foo(const T& t) {
    f(t); // Use f() as implementation detail
}
```

And it can be rewritten in our concept design as:

```
// This is marked auto because there are no semantic
// constraints (constexpr constraints don't matter since they
// are checked statically).
auto concept Small<T, n> {
    typename T;
    constexpr int n;
    sizeof(T) <= n;
}
const int max = 200;

template <Small<max> T>
void f(const T&);

template <typename T>
requires {}
void f(const T&);

template <typename T>
requires {}
void foo(const T& t) {
    f(t); // Use f() as implementation detail
}
```

Due to concept-based specialization, `Small` is an optional requirement of `f` and doesn't have to be a requirement of `foo`. Also, since there are no semantic requirements in the concept, the client code using `foo` will automatically use the optimized version when relevant, without being aware of it (the library exposing `foo` may want to advertise it in its public interface, but we stick to the scenario in [SR-05] where it doesn't want to).

Note that the decision to mark the above concept as `auto` is only based on the lack of semantic requirements and not on a subjective compromise of safety for conciseness — even though the most specialized overload is picked when relevant in this case, even without a specific concept map, the static checking of constexpr constraints guarantees that it is safe to do so.

Concept-based specialization

The use of concept-based specialization *without using Not* prevents the constraints on specializations from leaking to the callers even when the involved concepts do have semantic requirements.

Consider this example, again from [SR-05]:

```
template <C1 T> void helper(T x);
template <C2 T> void helper(T x);
template <C3 T> void helper(T x);

template <typename T>
void foo(T x) {
    helper(x);
}
```

Depending on the context, `foo` can be constrained in two ways. In case one of the concepts (say, `C2`) is implied by the other two, we can use it alone to constrain `foo`:

```
template <C2 T>
void foo(T x) {
    helper(x);
}
```

Otherwise, if none of `C1`, `C2` and `C3` is implied by the other two we need to expose all three requirements:

```
template <typename T>
requires { C1<T> | C2<T> | C3<T>; }
void foo(T x) {
    helper(x);
}
```

Here we are exposing `C1`, `C2` and `C3` in the prototype, but this is reasonable since no other requirements would work. If there was a requirement

`C` that made `foo` typecheck, calling C_i the requirement of the overload of `helper` chosen by concept-based overload resolution (disregarding concept-based specialization) we would have that:

- $C \Rightarrow C_i$, otherwise the i-th overload of `helper` wouldn't even be considered by overload resolution.
- $(\forall j \neq i)\ \ (C_j \Rightarrow C)$, otherwise we would have overconstrained `helper` by preventing the use of types that model C_j and not C
- $(\forall j \neq i)\ \ (C_j \Rightarrow C_i)$, by combining the previous two.

And this would contradict the initial hypothesis that no concept is implied by all the others.

We have used Xor above since it is guaranteed to work, but in the usual case that the conjunction of `C1`, `C2` and `C3` is a valid concept we can use Or (`||`) instead.

In both cases, this is equivalent to a set of concept based overloads and the same will happen to callers of `foo`, so the same discussion applies (i.e., Or and Xor aren't more viral than usual concept-based overloads).

14.2 Evaluation of the concept design

[SR-03] describes a list of design goals for C++ concepts. In this section we will evaluate our design choices with respect to these goals.

We use the same order as in [SR-03], that is intended as «rough priority order». We also cite relevant parts of the descriptions of each aim included in [SR-03].

A system as flexible as current templates

«[...] constraints should not require explicit declaration of argument types and be non-hierarchical. In particular, built-in types, such as `int` and `Shape*` should remain first-class template arguments, requiring no workarounds.»

We don't require exact declarations of the argument types. Conversions both in the parameters and in the return type are allowed during matching.

As in other C++ concept designs, concepts are not hierarchical (i.e., additional concept maps can be added to types provided by third-party libraries).

We propose no workarounds for builtin types, except that the concept-enabled standard library must now provide concept maps for builtin types in terms of the concepts in the standard library. Any proposal that uses explicit concepts needs such concept maps (even though we allow the use of both explicit and implicit concepts).

Inner types will be attached to builtin types using the feature of "Member types" introduced in [SR-03]. In section B.4 we show a concept map that adds such inner types to built-in pointers.

Enable better checking of template definitions

«The ideal is complete checking of a template definition in isolation from its actual arguments.»

Our design allows separate typechecking, that fully meets such ideal.

Enable better checking of template uses

«The ideal is complete checking of a template use in isolation from the template definition.»

Again, since our design allows separate typechecking, the ideal is met.

Better error messages

«[...] Three kinds of errors must ideally be caught and succinctly reported:

- the use of an unspecified operation in a template within a template definition (without access to the template arguments)
- the lack of a required operation by a template argument type (at the point of template use without access to the template definition)
- an invalid use of an operation or combination of operations within a template definition (this may require access to actual template arguments).

Note that it requires increasing amounts of information to detect those three categories of errors.»

Separate typechecking guarantees that all three kinds of errors are detected even before the concrete types are known.

Selection of template specialization based on attributes of template arguments

«It must be possible to define several templates (e.g. template functions or template classes) and to select the one to be used based on the actual template arguments. Furthermore, we must be able to specify preferences among related types and among separately developed types (class is the minimally constraining concept).»

This is exactly what concept-based overloading and specialization do. Preferences among types can be specified using concept maps, either manually or reusing existing type traits (see B.4).

Typical code performs equivalent to existing template code

«It is easy to improve checking of templates at the cost of flexibility and run-time performance — just define a template argument to be some kind of abstract class and you have perfect separation of concerns and conventional type checking. [...] The constraints/concept facility must sustain and enhance the compile-time evaluation and inlining that are the foundations of the performance of classical template code.»

Our proposed concept checking can be implemented in terms of conventional typechecking using synthesized templates (see chapter 12) and inheriting their compile-time evaluation and inlining capabilities.

Allowing conversions in the parameters and return types potentially decreases performance due to the introduction of wrapper functions. By inlining such conversions in the caller we can recover the performance loss. Note that such inlining should not affect type checking (including overload resolution and visibility issues) — it should just be an optimization.

Simple to implement in current compilers

«This should hold true for compile-time, link-time, and code-generation concerns. Templates have become terribly difficult to implement well. A constraint/concept facility should not make that problem significantly worse. Furthermore, it should ease the checking of most uses.»

Chapter 12, on the implementation of concepts hints that implementing our proposed concept features on top of an existing C++ compiler should not require extensive modifications.

However, such chapter only covers basic concept checking and code generation. An actual implementation of these features (for example, in ConceptClang) is definitely needed to prove that this is the case and also to measure how concept checking affects compilation time and the performance of the resulting binary (if it's affected at all).

Compatibility with current syntax and semantics

«Ideally, every existing piece of template code will still work as before. Furthermore, it should do so when mixed with code using concepts. The syntax used to express concepts should fit smoothly with the existing template syntax. We cannot require existing types to be systematically rewritten to be useful as arguments to templates using concepts.»

We allow the use of unconstrained code from constrained code and vice-versa, and this requires no modifications to existing unconstrained code. As required by separate typechecking, any unconstrained code used in a constrained scope will be fully typechecked with its requires clause.

We believe that the concept syntax we use is similar to common C++ code and thus easy to grasp for existing C++ developers. This is a subjective claim, though, so we welcome feedback and constructive criticism.

Separate compilation of template and template use

«This obvious ideal requires use of some sort of `vtbl` (container of pointers to functions) to implement the interface from a template to a template argument.»

Our concept design does not meet this requirement. In fact, due to C++'s expressive power, in general it's impossible to fully compile a constrained template (or even a template instantiation) down to machine code in separation from the instantiation context, parametrized on the called functions.

For example, consider:

```
template <int n>
class C {
    static void foo() {
        ...
    }
}

template <>
class C<0> {
    static void foo() {
        ...
    }
}

... // Other specializations of C.

template <typename T>
void f() {
    C<sizeof(T)>::foo();
}
```

The behavior of `f` depends on the size of `T`, that can only be determined when `f` is instantiated on a concrete `T`.

Even if it was possible, by extending the `vtable` with compile-time values, we wouldn't be able to inline the calls to `foo()` and this would break the performance requirements on constrained templates discussed earlier.

Our proposed features are incompatible with this `vtable` approach anyway, because the use of implications as concept combinators means that we need to pass template template parameters (as we saw in chapter 12) to the synthesized unconstrained template.

The encoding of constrained concepts as unconstrained ones that, when instantiated, need no further overload resolution and are guaranteed not to produce compilation errors can be can be thought as a form of separate pre-compilation.

Simple/terse expression of constrains

«The most general statement of the constraints on template arguments is that the template specialization compiles. That's what type checking does for otherwise unconstrained template arguments. To be useful and comprehensible in the absence of the template body, a constraint must succinctly specify a logical property.»

This is another subjective condition. We think that our concept syntax is concise and easy to use, and any feedback on this would be appreciated.

Express constraints in terms of other constraints

«Often, a template requires arguments that must meet a logical combination of concepts. Thus, a programmer must be able to name constraints and express new constraints as combinations of existing ones. This could include "negative constraints", such as "not a reference".»

We allow the use of `And` in concepts, and of `Or` and `Xor` in concept constraints. We discard `Not` due to the issues detailed in sections 5.9 (for `Not` as a general connective) and 11.1 (regarding the use of `Not` for concept-based overloading).

Constraints of combinations of template arguments

«Often, several template arguments are used in combination. In such cases, the ability to combine operations on arguments can be more important than the exact attributes of individual arguments. It should be possible to express requirements on the relationships among template arguments.»

We allow concept constraints on combination of types, as most concept papers for C++ (except the very first ones) propose to do.

Express semantics/invariants of concept models

«For example, a user should be able to state that his/her array class has the property that its elements occupy contiguous storage, or that an increment followed by a decrement of his/her bidirectional iterator is a no-op.»

The property on contiguous storage can be added as a concept in the standard library with informal requirements, together with concept maps for builtins. The second property can be formalized using an axiom.

The extensions shouldn't hinder other language improvements

«In particular, constraints/concepts should not preclude other suggested improvements in support of generic programming, such as template aliases, templated name spaces, and namespace template arguments.»

We do propose the integration of concepts with modules (chapter 9) but this is only to gain additional advantages. Such integration is not needed to make our concept features compatible with modules.

Any improvements in the C++ generic programming can be matched with corresponding improvement to the specification logic, to keep them consistent. For example, namespace template arguments will be compatible as long as we also allow them in concept constraints.

14.3 Limitations of the concept design

Our design choices allow the specification of *almost* all current C++ templates (transforming them in unconstrained templates), but not all. Consider the following code:

```
template <bool b>
requires {}
class X {};

concept C<> {
    void g(X<false>);
    void g(X<true>);

    template <bool b>
    void h(X<b>);
}

template <typename T, bool b>
requires { C<>; }
void f() {
    X<0>();                            // (1) Ok.
    X<sizeof(vector<int>)>();          // (2) Ok.
    g(X<0>());                         // (3) Ok.
    g(X<sizeof(vector<int>)>());       // (4) Ok.

    vector<T>();                       // (5) Ok, of course.

    static const bool k = b;           // (6) Ok.
    static const int l = sizeof(T);    // (7) Ok.

    X<b>();                            // (8) Ok?
    X<sizeof(T)>();                    // (9) Ok?
```

```
    h(X<b>());                          // (10) Ok?
    h(X<sizeof(T)>());                  // (11) Ok?

    g(X<b>());                          // (12) Error.
    g(X<sizeof(T)>());                  // (13) Error.
}
```

In a constrained scope (such as the body of `f` above) we have to distinguish two kinds of compile-time expressions: the expressions whose value can be calculated during typechecking and the ones that can only be calculated at instantiation time. We'll call the second kind of expressions "unknown [compile-time] expressions".

Non-type template arguments (such as `b` above) and the size of unknown types (for example, `sizeof(T)` above) are unknown expressions. Also, any compile-time expression containing an unknown expression is itself an unknown expression.

Unknown expressions can still be used as constant expressions, but using them as template parameters to types can prevent overload resolution. Consider the examples 12 and 13 above. The typechecker doesn't have enough information to choose the right overload of `g`, even though the two overloads aren't ambiguous. Also, it's clear that the two overloads cover all possible values of `bool`, but there is no overload of `g` that can cover all possible values.

We may want to allow the use of types instantiated with unknown expressions when overload resolution has enough information to succeed anyway, as in the examples 8, 9, 10 and 11 above.[2] However, this would be a very intrusive change to overload resolution, and may not be desirable.

```
template <typename T>
void f_impl(const T& x, true_type) {
    ... // T is a POD
}
template <typename T>
void f_impl(const T& x, false_type) {
    ... // T is a non-POD
}
template <typename T>
void f(const T& x) {
    f_impl(x, integral_constant<bool, is_pod<T>::value>());
}
```

[2] In this case, we would also have to consider expressions obtained through such types as unknown expressions.

This is a more real-world example: the above function `f` can *not* be concept-constrained (unless we modify the body) and also can't be called by a concept-constrained function, since `is_pod<T>::value` is an unknown compile-time expression and a constrained concept wouldn't be able to pick the right template specialization.

If we want to call `f` from constrained templates, we need to refactor it using concept constraints instead of boolean template parameters, for example:

```
template <typename T>
requires {}
void f(const T& x) {
    ... // T is a non-POD
}

template <POD T>
void f(const T& x) {
    ... // T is a POD
}
```

We are using a `POD` concept. All built-in boolean type traits should be matched by corresponding concepts with empty APIs and predicates that require a concrete type with the relevant property. Such concepts can be defined in the standard library in terms of the respective type traits, using constexpr constraints.

14.4 Comparison with other concept designs

These tables summarize the main design decisions for concepts, compared to four papers: [SGJ+-05] and [SR-05] to represent the proposals of the Indiana and Texas groups, respectively, before the joint proposal in [GS-06]. We don't include [SS-12] in the comparison because that paper is more focused on formalizing STL concepts rather than on completely describing the needed language features.

In the tables, ✓ means that a feature is supported, ✗ means that it was considered in the paper and rejected and - means that it is not adopted in the paper, but wasn't discussed either.

For quick reference, we include the relevant chapter or section together with each feature.

Core decisions	[SGJ+-05]	[SR-05]	[GS-06]	This thesis
Explicit vs implicit concepts (8)	explicit	implicit	both, auto concepts are implicit	both, depends on compiler flags
Constraint syntax (2.1)	pseudo signatures	use patterns, no functio-nalization	pseudo signatures	pseudo signatures
Separate typechecking (14.2)	✓	?[1]	✗, «nearly separate»	✓

Concept composition	[SGJ+-05]	[SR-05]	[GS-06]	This thesis
And (5)	✓, with disjont union	✓, with union	✓, with union	✓, with union
Or (5)	✗	✓	✓	✓, limited
Not (5)	✗	✓	✓	✗
Implication (5)	–[2]	–	–[2]	✓
Forall + implication (6) (template requirements)	✓, only single constraints	–	✓, only single constraints	✓

Concept maps	[SGJ+-05]	[SR-05]	[GS-06]	This thesis
Templated (6)	✓	✓	✓	✓
With body (3.1)	✓	✓	✓	✗
As qualifiers (3.1)	✓	–	✓	✗
Affecting visibility (3.1)	–	–	–	✓
Can be overloaded (3.1)	✓	✓	✓	✗

Other	[SGJ+-05]	[SR-05]	[GS-06]	This thesis
Concept-based overloading (11)	✓	✓	✓	✓, except for types
Concept-based specialization (11.7)	–	✓, meets not checked	✓, meets not checked	✓, except for types
Same-type constraints (7)	✓	✗	✓	✓, two versions

[1] The paper tries to achieve it and claims that it holds, but several of the suggested features seem to break it (e.g., concept-based specialization without checking for meets and use patterns without functionalization).

[2] Requires clauses are proposed for both templated and non-templated function definitions, so implication for single constraints is consistent with the paper.

14.5 Future work

We hope that the ideas in this thesis will help in the development of C++ concepts, but this work is far from being complete. In this section we summarize what in our opinion is most needed.

Feedback and discussion

C++ concepts have generated a lot of discussion in the involved research groups, and we expect that this thesis will be no exception. Many design decision involve tradeoffs in terms of conciseness, safety, performance, maintainability and simplicity, just to name a few.

Such decisions are inherently subjective, since different points of view would suggest different solutions. Therefore, further discussion will help to pinpoint the main outstanding issues and to address them, hopefully result in a generally accepted design.

Some features are still at an early stage of development and may undergo significant changes. Pre/post-conditions, concept-based modules and inner requirements fall into this category.

Theoretical analysis

Is it possible to extend the F^G language defined in [SL-05] with the additional concept features described in this thesis (especially, templated concept requirements, inner requirements and optional requirements)?

We believe that it is possible, but this topic is out of the scope of this thesis. Such an extension will give more insights on those features and will help to ensure that the resulting type system has the expected properties.

Our encoding of constrained code as templates suggests that F is not expressive enough, lacking types dependent on types and/or values.

Implementation

Implementing the discussed concept features in a compiler (most likely ConceptClang or ConceptGCC) will help finding subtle issues. Also, this would allow to measure of the performance of both concept-checking itself and of the resulting binary. Performance is considered very important in C++, and poor performance has been one of the reasons why concepts were not included in C++11.

Formalization of the standard library

In appendix B we formalize the concepts that are informally defined in the STL. To complement a compiler implementation, the rest of the C++ standard library should also be made concept-aware, and possibly also encapsulated in a concept-based module.

A. Other features

This appendix contains a collection of minor concept features and of discarded ones, with brief explanations.

A.1 Uniform member syntax

Some papers, for example [SR-03], propose to allow calling a function and a method with the same syntax, in concept-constrained code.

There are three possible ways to do so:

1. Consider the function call syntax and the member call syntax equivalent
2. Allow to use the member call syntax to call functions, but not the other way around
3. Allow to use the function call syntax to call members, but not the other way around

The 3rd way is the one proposed in [SR-03].

Discussion

Adding this feature will make concept-constrained code “special” compared to other code, making the language less uniform and more complex to understand. On the other hand, if we extend this behavior to non-concept-constrained code, with the same philosophy of concept maps extending visibility, then we lose backward compatibility, because we would be defining a method (or a function) in terms of a function (or, respectively, a method) in the concept map, while such a method (or, respectively, function) may be defined later, producing a double-definition error.

For these reasons, we don’t think a uniform member syntax should be added to the language. However, such a feature may be desirable in some projects and it can be emulated using in-concept default operation definitions, even though all operations have to be written twice inside the concept, once as a function and once as a method, and one (or both) of them must be defined in terms of the other. This library-based approach can emulate each of the 3 variants above.

A.2 Late check

[GS-06] and [GSW-08] propose a `late_check` keyword, that prevents concept checking of the following expression (in [GS-06]) or block (in [GSW-08]). Such non-checked code is checked *later*, when the template is instantiated with concrete types, as in unconstrained templates.

This feature is called a *hack* in [Str-09], and we agree with this definition. We think that it should not be included in the standard, as it would obviously break separate typechecking.

The lack of `late_check` forces code that can't be constrained, as the example in section 14.3, to be implemented as an unconstrained template.

A.3 Substitution

A possible generalization of a concept's parameter list is to allow arbitrary substitutions when a concept is instantiated. This allows the user of a concept to generalize on some names that weren't expected to be generalized on when the concept was defined. For example:

```
concept Vector<vector, T> {
    ...
    size_t vector::size() const;
}
template <Regular T>
class weird_vector {
    ...
    size_t Size() const;
};
template <typename T>
Vector<weird_vector<T>, T>[weird_vector<T>::size
                            => weird_vector<T>::Size];
```

This is equivalent to adding another parameter to the concept:

```
concept Vector<vector, T, size_function> {
    ...
    size_t size_function(const vector&);
}
template <Regular T>
class weird_vector {
    ...
    size_t Size() const;
};
template <Regular T>
Vector<weird_vector<T>, T, weird_vector<T>::Size>;
```

Note that this is very different from a non-trivial mapping in a concept map: `Vector<weird_vector, T, weird_vector::size>` is a different instantiation than `Vector<weird_vector, T, weird_vector::Size>`, so if a constrained template requires the first one, the above concept map will *not* match the requirement.

In a similar way, arbitrary substitutions would be applied to expressions. For example, after the previous code we could write:

```
template <typename V, typename T>
requires { Vector<V, T>; }
void f() {
    ...
}

void g() {
    f<weird_vector<int>, int>()
        [weird_vector::size => weird_vector::Size];
    // Checks for:
    // Vector<weird_vector<int>, int>
    //     [weird_vector::size => weird_vector::Size]
    // And finds the above concept map, ok.
}
```

Discussion

We think that allowing arbitrary substitutions is not intuitive for users (because users may expect it to work as a non-trivial mapping in a concept map) and not that useful in practice.

Moreover, this feature may rename types and operations that belong to the enclosing scope (for example, builtin entities and entities defined in headers `#include`d before the definition), breaking the semantic properties expected by the implementation.

We prefer using the inner requirements feature of chapter 10, that can replace this one in most cases while also avoiding the renaming of entities from the enclosing scope.

A.4 Methods obtained as replacements from functions

Since a method is basically a function that takes a hidden `this` parameter, we think that we should allow the following code:

```
concept StrictWeakOrder<T, less> {
    bool less(const T&, const T&);
    ...
}
concept IFoo<Foo> {
    class Foo {
        static bool static_less(const Foo&, const Foo&);
        bool method_less(const Foo&) const;
    }
    StrictWeakOrder<Foo::static_less, Foo>;
    StrictWeakOrder<Foo::method_less, Foo>;
}
```

Where the two `StrictWeakOrder` requirements in `IFoo` match respectively the static method `static_less` and the method `method_less`, as if they were global functions. This allows to reuse the same concepts defined for global functions also for methods and static methods.

A.5 Implicitly-generated constraints for argument types

Often, the `requires` clause of an operation contains properties to ensure the existence of the types used as parameters or return type.

For example, consider the following code:

```
template <Moveable T>
class vector {
    ...
};
template <Nice T>
class weird {
    ...
};
...
template <typename T>
requires {
    Moveable<T>;
    Nice<T>;
}
std::vector<T> f(const std::weird<T>& y) {
    ...
}
```

The `Moveable<T>` and `Nice<T>` requirements can be deduced from the signature of `f`, so we can let the user omit them and have the compiler add them afterwards.

Of course, even if all constraints can be omitted, we still have to specify `requires {}`, otherwise we would get an unconstrained template, and the implicit concept requirements would not be added.

A generalization of this feature has already been proposed in [SGJ+-05], and such a generalization may be more useful than the instance explained here. See section 4.1 of that paper for details.

We think that implicitly adding concept requirements decreases readability of the code. For this reason, we discard this feature.

A.6 Delete syntax

This feature for concepts corresponds to the deleted operation declarations introduced in C++11.

```
concept IFoo<Foo> {
    class Foo {
        static void f(double);
        static void f(int) = delete;
    };
}

template <IFoo Foo>
void g() {
    Foo::f(1.0); // Ok.
    Foo::f(1);   // Error: f(int) deleted.
}

class my_foo {
    static void f(double);
    // Not necessary, just to show that it doesn't break the
    // following concept map.
    static void f(int);
};
IFoo<my_foo>; // Ok.
```

The function `g` that uses `IFoo` can see both overloads of `f`, but an error occurs when the deleted overload is selected by overload resolution (as already happens for deleted operations in C++11).

On the other hand, deleted functions are not included in the requirements of a concept, so the concept map doesn't look for an `f(int)` in scope.

As in C++11, it's an error to specify the same declaration both as deleted and as non-deleted.

A.7 Variadic concepts

[SS-11] proposes variadic concepts, in a similar fashion to variadic templates. This language feature allows to re-use the same concept for a varying number of arguments, as in the following:

```
concept Callable<f, Params...> {
    typename... Params;
    typename Result = auto;
    Result f(Params...);
}
```

We used a slightly different syntax above, compared to [SS-11], to be consistent with the syntax used in the rest of this thesis.

This feature is not to be confused with the ability to require variadic templates in concepts, which extends the expressive power of the language by allowing an “infinite” number of assertions in a concept map, so that it’s possible to constrain variadic templates.

As with variadic templates, this feature allows to save code. There are no apparent problems with it, since for each instantiation of a concept a non-variadic version of the concept can be generated, just by counting parameters and performing trivial code transformations near the ... in the concept.

A.8 Concept definitions in namespaces

This feature allows libraries to define their concepts in a namespace. For example, the standard library can define the `std::Hash` concept as follows:

```
namespace std {
    concept Hash<H, Key> {
        typename Key;
        CopyConstructible<H>;
        Destructible<H>;
        class H {
            size_t operator()(Key);
        }
    }
}
```

Note that, for example, `std::Hashable<my_hash, int>` requires the types `my_hash` and `int`, not `std::my_hash` and `std::int`. The namespace clause affects the name of the concept and is used to resolve names that aren’t declared in the concept. In the example above, `size_t` resolves to `std::size_t`, assuming that it was defined before this concept.

A.9 Concept maps in namespaces

Similarly to concept definitions in namespaces, we allow concept maps in namespaces. The only effect is that the concept and the concept parameters are also looked up in that scope (*before* the substitution of parameters).

```
concept C<T> {
    void f(T);
}
namespace std {
    class my_type {
        ...
    };

    C<my_type>;
}
```

Is equivalent to:

```
concept C<T> {
    void f(T);
}
namespace std {
    class my_type {
        ...
    };
}
C<std::my_type>;
```

Note that this requires a function `f` in the *global* namespace, not in namespace `std`, because only the parameters of the concept are affected.

Also, the above concept maps will *not* match a requirement for `C<my_type>` — it matches `C<std::my_type>` only.[1]

A.10 Other constraints

This section talks about other atomic constraints that can be useful to specify in a concept.

Namespaces

This feature allows to specify a namespace name in a concept. This works basically the same as for type specifications, and we can use a syntax similar to the class pseudo-definition syntax.

[1] However, if a requirement for `C<my_type>` is used within a `namespace std` block and `std::my_type` is in scope, then `my_type` will be resolved to `std::my_type` before the requirement is checked, so the above concept map will be a match.

```
concept IFoo<Foo> {
    class Foo {
        ...
    };
    namespace std {
        void swap(Foo&, Foo&);
    }
}
```

Friend declarations

We don't think that friend declarations should be allowed in a concept, since they break the specification-based polymorphism mindset on which concepts are based on. For example, a constrained template can accept any type as long as it has certain properties, while the use of `friend` declarations would force to specify the exact type that was specified as `friend`.

This doesn't mean that a concept-aware class can't use `friend`: it only means that in order to access such friend declarations, the class has to depend on a specific type and not just on a concept requirement.

Friends are typically used as an implementation detail, and the API of a library using `friend` can still be specified using concepts anyway.

Attributes

There seem to be no problems in allowing attributes within concepts, as long as we decide whether concept maps check for the existence of the attribute or just add the ones in the concept to the ones specified by the user.

This choice should probably be done per-attribute.

Enums

`enum` definitions should also present no problems, with the usual declaration and definition syntax.

Global variables and public fields

Allowing to specify variables and fields within concepts should also present no problems. Such specifications are especially important when using concept-based modules, because such variables and fields may be part of the API of a library (for example `std::cout` and `std::pair<T,U>::first`).

For `static const` variables and fields, we allow to specify a default value, with a syntax similar to the one that we used for types.

These are some examples:

```
concept Pair<pair, T, U> {
    class pair {
        ...
        T first;
        U second;
    };
}
```

```
// iostream.cpp
import istream;
import ostream;

export {
    namespace std {
        istream cin;
        ostream cout;
    }
}

namespace std {
    extern istream cin;
    extern ostream cout;
}
```

```
enum Calendar {
    GREGORIAN, PROLEPTIC_GREGORIAN
};
concept IDate<Date> {
    class Date {
        ...
        // If not defined, the calendar is
        // PROLEPTIC_GREGORIAN.
        default static const Calendar calendar
            = PROLEPTIC_GREGORIAN;
    };
}
concept ProlepticDate<Date> {
    IDate<Date>;
    class Date {
        // The calendar must be PROLEPTIC_GREGORIAN.
        static const Calendar calendar = PROLEPTIC_GREGORIAN;
    };
}
```

Specialization

We propose to add a built-in `Specialization` concept that checks that a type is a specialization of a template. For example, the following two concept maps are satisfied by the C++ standard library:

```
Specialization<std::vector<int>, std::vector>;
Specialization<std::true_type, std::integral_constant>;
```

Due to the generality of this concept, the only property that we can expose in the API is that the first argument is a type. So, as far as the API is concerned, this concept is equivalent to:

```
auto concept Specialization<T, X> {
    typename T;
}
```

Templated axioms

When a concept specifies templated entities, often the axioms also need to be templated and have their own requirements.

In fact, it's already possible to express such requirements by using a templated concept requirement for a concept that contains the axiom.

This also allows the use of equality in axioms even if the enclosing concept doesn't require it, and we think this solution should be preferred to the introduction of a builtin, non-evaluable `eq` predicate that checks the equality of values.

By requiring `EqualityComparable` we allow testing tools to execute such axioms for the types that implement it. This would not be possible with the builtin `eq`, unless its equivalence with any user-defined `operator==` is specified in the standard.

B. STL concepts

In this appendix we formalize the STL concepts using the concept features chosen in the previous chapters, to show how the various features work together.

Additional specifications can and should be included in the concepts, as comments (for example, complexity specifications and pre/post-conditions). We don't include such comments because they would decrease conciseness and thus readability — we just want to show how requirements can be formalized.

All the following concepts and properties are assumed to be defined in the `std` namespace.

B.1 Basic operations

Callable

```
auto concept Callable<F, Args...> {
    typename... Args;
    typename Result = auto;
    class F {
        Result operator()(Args...);
    }
}
```

UnaryPredicate

```
auto concept UnaryPredicate<Op, T> {
    class Op {
        bool operator()(const T&);
    }
}
```

BinaryPredicate

```
auto concept BinaryPredicate<Op, T> {
    class Op {
        bool operator()(const T&, const T&);
    }
}
```

UnaryOperation

```
auto concept UnaryOperation<Op, T> {
    class Op {
        T operator()(T);
    }
}
```

BinaryOperation

```
auto concept BinaryOperation<Op, T> {
    class Op {
        T operator()(T, T);
    }
}
```

DefaultConstructible

```
concept DefaultConstructible<T> {
    class T {
        T();
    }
}
```

Destructible

```
auto concept Destructible<T> {
    class T {
        ~T();
    }
}
```

Swappable

```
concept Swappable<T, U> {
    typename T;
    typename U;
    void swap(T, U);
    void swap(U, T);
}
```

Here we side-step the issue of visibility of `std::swap` by requiring a generic `swap` function. The client code (unless it's also constrained code requiring `Swappable`) will have to ensure that a matching `swap` function is in scope, either due to a definition or to the inclusion of `<utility>` together with a `using std::swap;` or `using namespace std;` declaration in scope.

This is consistent with the note in the standard saying that `<utility>` may or may not be included by code requiring `Swappable`.

MoveConstructible

```
concept MoveConstructible<T> {
    class T {
        T(T&& x);

        requires {
            DefaultConstructible<T>;
            Swappable<T&, T&>;
        }
        T(T&& x) : T() {
            swap(*this, x);
        }
    }
}
```

We specify that, if `T` is also `DefaultConstructible` and `Swappable`, then the move assignment has the specified semantics.

In this way, we avoid creating a concept requiring all three conditions. With that approach, the programmer would have to notice when all three conditions are met and use the combined concept in these occasions.

Both this and the alternative formalization have a subtle issue: since `std::swap` is defined in terms of move construction and assignment, in scopes where `std::swap` is visible through a using declaration we would risk to use both definitions and get an infinite recursion, if it wasn't for the looping detection explained in section 13.4.

CopyConstructible

```
concept CopyConstructible<T> {
    MoveConstructible<T>;
    class T {
        T(const T& x);
    }
}
```

ConvertibleTo

```
auto concept ConvertibleTo<T, U> {
    class T {
        operator U();
    }
}
```

AssignableTo

```
auto concept AssignableTo<T, U> {
    U& operator=(T);
}
```

MoveAssignable

```
concept MoveAssignable<T> {
    class T {
        void operator=(T&& x);

        requires { Swappable<T&, T&>; }
        void operator=(T&& x) {
            swap(*this, x);
        }
    }
}
```

CopyAssignable

```
concept CopyAssignable<T> {
    requires MoveAssignable<T>;
    class T {
        T& operator=(T& x);

        requires { CopyConstructible<T>; }
        T& operator=(T& x) {
            if (this != &x)
                *this = T(x);
            return x;
        }
    }
}
```

ValueSwappable

```
auto concept ValueSwappable<iterator> {
    Iterator<iterator>;
    Swappable<iterator::value_type>;
}
```

Incrementable

```
concept Incrementable<T> {
    CopyConstructible<T>;
    T& operator++(T&);
    T operator++(T& x, int) {
        T tmp = x;
        ++tmp;
        return move(tmp);
    }
}
```

Decrementable

```
concept Decrementable<T> {
    CopyConstructible<T>;
    T& operator--(T&);
    T operator--(T& x, int) {
        T tmp = x;
        --tmp;
        return move(tmp);
    }
}
```

SemiRegular

```
auto concept SemiRegular<T> {
    DefaultConstructible<T>;
    CopyConstructible<T>;
    CopyAssignable<T>;
    Destructible<T>;
    Swappable<T&, T&>;
}
```

Regular

```
auto concept Regular<T> {
    SemiRegular<T>;
    Comparable<T>;
}
```

By moving the in-concept function definitions inside the fine-grained concepts, `SemiRegular` and `Regular` can be marked as `auto` concepts, so that they become just abbreviations for their contents.

B.2 Ordering relations

Equivalence

```
template <typename T, BinaryPredicate<T> Eq>
property equivalence(Eq eq) {
    axiom reflexivity(T x) {
        eq(x, x);
    }
    axiom symmetry(T x, T y) {
        if (eq(x, y))
            eq(y, x);
    }
    axiom transitivity(T x, T y, T z) {
        if (eq(x, y) && eq(y, z))
            eq(x, z);
    }
}

concept Equivalence<T, eq> {
    typename T;
    bool eq(T, T);
    axiom {
        equivalence<T, decltype(eq)>(eq);
    }
}
```

The axioms are defined in a separate property, but can be moved inside the concept. We prefer this implementation since we reuse the same axioms for both functors (using the property) and plain functions (using the concept).

Due to the requirement splitting for `eq`, in the axiom the `eq` expression is a functor exposing the four available overloads (see section 13.3).

EqualityComparable

```
concept EqualityComparable<T> {
    Equivalence<T, operator== >;
    bool operator!=(const T& x, const T& y) {
        return !(x == y);
    }
}
```

The requirement for `operator==` can match a (possibly templated) global function, a function in the namespace of `T` (due to Koenig lookup) or a method in the class `T`.

Note the definition of `!=` in terms of `==`. Such definition will be used if the user doesn't provide one, and otherwise becomes an axiom.

StrictWeakOrder

```
template <typename T, BinaryPredicate<T> Less>
property strict_weak_order(Less less) {
    axiom irreflexivity(T x) {
        !less(x, x);
    }
    axiom antisymmetry(T x, T y) {
        if (less(x, y))
            !less(y, x);
    }
    axiom transitivity(T x, T y, T z) {
        if (less(x, y) && less(y, z))
            less(x, z);
    }
    axiom uncomparable_equivalence {
        auto eq = [](const T& x, const T& y) {
            return !less(x, y) && !less(y, x);
        };
        equivalence<T, decltype(eq)>(eq);
    }
}

concept StrictWeakOrder<T, less> {
    bool less(const T&, const T&);
    axiom {
        strict_weak_order<T, less>;
    }
}
```

LessThanComparable

```
concept LessThanComparable<T> {
    StrictWeakOrder<T, operator< >;
    bool operator>(const T& a, const T& b) {
        return b < a;
    }
    bool operator>=(const T& a, const T& b) {
        return !(a < b);
    }
    bool operator<=(const T& a, const T& b) {
        return !(a > b);
    }
}
```

The above code should be auto-explanatory. The only thing worth noting is the reuse of the property `equivalence` in `strict_weak_order` (and thus indirectly in `LessThanComparable`). We can't reuse `Equivalence` here, since we are not exposing the equivalence, but the property can be reused.

TotalOrder

```
concept TotalOrder<T, less> {
    EqualityComparable<T>;
    StrictWeakOrder<T, less>;
    axiom(T x, T y) {
        (x != y) == (less(x, y) || less(y, x));
    }
}
```

Comparable

```
auto concept Comparable<T> {
    LessThanComparable<T>;
    TotalOrder<T, operator< >;
}
```

B.3 Algebraic concepts

Functionalization

This is just a helper concept to help simplify some later concepts.

```
concept Functionalization<T, U, op, op_equal> {
    CopyConstructible<T>;
    CopyAssignable<T>;
    T op(T, U);
    T& op_equal(T&, U);

    T op(const T& x, const U& y) {
        T tmp = x;
        op_equal(tmp, y);
        return move(tmp);
    }
    T op(const T& x, U&& y) {
        T tmp = x;
        op_equal(tmp, move(y));
        return move(tmp);
    }
    T op(T&& x, const U& y) {
        op_equal(x, y);
        return move(x);
    }
    T op(T&& x, U&& y) {
        op_equal(x, move(y));
        return move(x);
    }
}
```

Commutative

```
template <EqualityComparable T, BinaryOperation<T> Op>
property commutative(Op op) {
    axiom(T x, T y) {
        op(x, y) == op(y, x);
    }
}
concept Commutative<T, op> {
    EqualityComparable<T>;
    T op(T, T);
    axiom { commutative<T, decltype(op)>(op); }
}
```

Associative

```
template <EqualityComparable T, BinaryOperation<T> Op>
property associative(Op op) {
    axiom(T x, T y, T z) {
        op(x, op(y, z)) == op(op(x, y), z);
    }
}
concept Associative<T, op> {
    EqualityComparable<T>;
    T op(T, T);
    axiom { associative<T, decltype(op)>(op); }
}
```

Monoid

For this and later concepts modeling mathematical notions, we use high parametrization in the concepts, since there is no standard on the names, also due to the high reusability of these concepts.

This is in contrast to what we do in most other concepts, where there is a standard naming for the primitives and we favor conciseness over reuse. Note that code constrained with such concepts can still be reused using the feature of inner requirements together with adapter concepts (chapter 10).

```
concept Monoid<T, zero, plus> {
    SemiRegular<T>;
    T zero();
    T plus(T, T);
    axiom(T x) {
        plus(x, zero()) == x;
        plus(zero(), x) == x;
    }
}
```

Group

```
concept Group<T, zero, plus, minus> {
    Monoid<T, zero, plus>;
    axiom(T x) {
        plus(x, minus(x)) == zero();
        plus(minus(x), x) == zero();
    }
}
```

CommutativeGroup

```
auto concept CommutativeGroup<T, zero, plus, minus> {
    Group<T, zero, plus, minus>;
    Commutative<T, plus>;
}
```

Ring

```
concept Ring<T, zero, plus, minus, times> {
    CommutativeGroup<T, zero, plus, minus>;
    Associative<T, times>;
    axiom (T x, T y, T z) {
        times(x, plus(y, z))
            == plus(times(x, y), times(x, z));
        times(plus(x, y), z)
            == plus(times(x, z), times(y, z));
    }
}
```

CommutativeRing

```
auto concept CommutativeRing<T, zero, plus, minus, times> {
    Ring<T, zero, plus, minus, times>;
    Commutative<T, times>;
}
```

IntegralDomain

```
concept IntegralDomain<T, zero, one, plus, minus, times> {
    CommutativeRing<T, zero, plus, minus, times>;
    Monoid<T, one, times>
    axiom (T x, T y) {
        if (x != zero() && y != zero())
            times(x, y) != zero();
    }
}
```

OrderedIntegralDomain

```
concept OrderedIntegralDomain<T, zero, one,
                              plus, minus, times, less> {
    IntegralDomain<T, zero, one, plus, minus, times>;
    TotalOrder<T, less>;
    axiom(T x, T y, T z) {
        less(plus(x, z), plus(y, z)) == less(x, y);
    }
    axiom(T x, T y) {
        auto non_negative = [](const T& x) {
            return less(zero(), x) || x == zero();
        }
        if (non_negative(x) && non_negative(y))
            non_negative(times(x, y));
    }
}
```

Field

```
concept Field<T, zero, one,
              plus, minus, times, inverse> {
    IntegralDomain<T, zero, one,
                   plus, minus, times>;
    T inverse(T);
    axiom(T x) {
        if (x != zero())
            times(x, inverse(x)) == one();
    }
}
```

Arithmetic

This concept specifies the requirements and API of the builtin integer and floating-point types, but it is *not* restricted to builtins, and can also be used for user-defined types with similar behavior.

It's also worth mentioning that there are almost no precise semantic guarantees. This allows limited-range integers to model the concept (even though they can overflow) and even floating-point numbers, whose operations fail to satisfy even the simplest algebraic properties.

The conversion requirements are written in terms of the three biggest arithmetic built-in types to allow the conversion to and from these builtins. This doesn't mean that the implementation has to provide separate conversions: one of them is enough to match all three requirements.

```
auto concept Arithmetic<T> {
    Regular<T>;
    class T {
        explicit T(long long);
        explicit T(unsigned long long);
        explicit T(long double);
        explicit operator long long();
        explicit operator unsigned long long();
        explicit operator long double();
    }

    T operator+(T);
    T operator-(T);
    T operator+(T, T);
    T operator-(T, T);
    T operator*(T, T);
    T operator/(T, T);
    T& operator+=(T&, T);
    T& operator-=(T&, T);
    T& operator*=(T&, T);
    T& operator/=(T&, T);

    Incrementable<T>;
    Decrementable<T>;

    // The semantics of the above operations should
    // "roughly" satisfy the StrictArithmetic requirements.

    default T& operator-=(T& x, const T& y) {
        x += -y;
        return x;
    }
    default T& operator-=(T& x, T&& y) {
        x += -move(y);
        return x;
    }

    T operator+(const T& x) {
        return x;
    }
    T operator+(T&& x) {
        return move(x);
    }
    T& operator++(T& x) {
        x += T(1);
    }
    T& operator--(T& x) {
        x -= T(1);
```

```
    }
    Functionalization<T, T, operator+, operator+= >;
    Functionalization<T, T, operator-, operator-= >;
    Functionalization<T, T, operator*, operator*= >;
    Functionalization<T, T, operator/, operator/= >;
}
```

StrictArithmetic

The `StrictArithmetic` concept is mostly provided to describe the properties that "approximately" hold for models of `Arithmetic`. However, there are some models, for example infinite-precision integers and rationals.

The specification of `operator/` is deliberately weaker than the one in `Field`, that is not even approximately true for integers.

```
concept StrictArithmetic<T> {
    Arithmetic<T>;

    T& operator-=(T& x, const T& y) {
        x += -y;
        return x;
    }
    T& operator-=(T& x, T&& y) {
        x += -move(y);
        return x;
    }
    T zero() {
        return T(0);
    }
    T one() {
        return T(1);
    }
    OrderedIntegralDomain<T, zero, one,
                          operator+, operator-, operator*,
                          operator< >;

    axiom(long long x) {
        long long(T(x)) == x;
    }
    axiom(unsigned long long x) {
        unsigned long long(T(x)) == x;
    }
    axiom(T x, T y) {
        if (y != zero())
            x*y/y == x;
    }
}
```

Integer

```
concept Integer<T> {
    Arithmetic<T>;

    T operator~(T);
    T operator&(T, T);
    T operator|(T, T);
    T operator^(T, T);
    T operator%(T, T);
    T operator<<(T, size_t);
    T operator>>(T, size_t);
    T& operator&=(T&, T);
    T& operator|=(T&, T);
    T& operator^=(T&, T);
    T& operator%=(T&, T);
    T& operator<<=(T&, size_t);
    T& operator>>=(T&, size_t);

    T& operator-=(T& x, const T& y) {
        return x += -y;
    }
    T& operator-=(T& x, T&& y) {
        return x += -move(y);
    }

    Associative<T, operator& >;
    Associative<T, operator| >;
    Associative<T, operator^ >;
    Commutative<T, operator& >;
    Commutative<T, operator| >;
    Commutative<T, operator^ >;
    Functionalization<T, T, operator&, operator&= >;
    Functionalization<T, T, operator|, operator|= >;
    Functionalization<T, T, operator^, operator^= >;
    Functionalization<T, T, operator%, operator%= >;
    Functionalization<T, size_t, operator<<, operator<<= >;
    Functionalization<T, size_t, operator>>, operator>>= >;

    axiom (T x, T y) {
        (a/b)*b + a%b == a;
    }

    // The following axiom (intentionally) doesn't
    // fully specify the behavior of ~.
    axiom(T x, T y) {
        if (x >= 0 && y >= 0)
            (x & y) == ~(~x | ~y);
    }
```

```
    axiom (T x, T y, T z, size_t k) {
        if (x >= T(0) && y >= T(0) && z >= T(0)) {
            // Define << and >>.
            (x << 0) == x;
            (x >> 0) == x;
            if (k > 0) {
                (x << k) == (x << (k - 1))*T(2);
                (x >> k) == (x >> (k - 1))/T(2);
            }
            // Define & and | for powers of 2.
            (x & T(1)) == (x % T(2));
            (x & (T(1) << k)) == ((x >> k) % T(2)) << k;
            if ((x & (T(1) << k)) != T(0))
                (x | (T(1) << k)) == x;
            else
                (x | (T(1) << k)) == (x + (T(1) << k));
            // Define & and | for any number.
            (x & T(0)) == T(0);
            (x | T(0)) == x;
            (x & (y | z)) == (x & y) | (x & z);
            // Define ^.
            (x ^ y) >= T(0);
            if ((x & (1 << k)) == (y & (1 << k)))
                ((x ^ y) & (1 << k)) == T(0);
            else
                ((x ^ y) & (1 << k)) == (T(1) << k);
        }
    }
}
```

UnsignedInteger

```
concept UnsignedInteger<T> {
    Integer<T>;
    axiom(T x) {
        x >= T(0);
    }
}
```

StrictInteger

```
auto concept Integer<T> {
    Integer<T>;
    StrictArithmetic<T>;
}
```

B.4 Iterators

NullablePointer

```
concept NullablePointer<P> {
    SemiRegular<P>;
    EqualityComparable<P>;
    class P {
        P(std::nullptr_t) : P() { }
        operator bool() {
            return p != nullptr;
        }
        P& operator=(std::nullptr_t) {
            *this = P();
        }
    }
    bool operator==(P x, std::nullptr_t) {
        return x == P();
    }
    bool operator!=(P x, std::nullptr_t) {
        return x != P();
    }
    bool operator==(std::nullptr_t, P x) {
        return x == P();
    }
    bool operator!=(std::nullptr_t, P x) {
        return x != P();
    }
    axiom(P p, std::nullptr_t np) {
        P(np) == nullptr;
        (p = np, p == nullptr) <=> (p = np, true);
    }
}
```

Iterator

```
concept Iterator<iterator> {
    CopyConstructible<iterator>;
    CopyAssignable<iterator>;
    Destructible<iterator>;
    Swappable<iterator&, iterator&>;
    // The previous requirements are the same as SemiRegular,
    // except DefaultConstructible.

    class iterator {
        typename difference_type;
        typename value_type;
        default typename pointer = value_type*;
        default typename reference = value_type&;
        typename iterator_category;
        Integer<difference_type>;

        reference operator*();
        default pointer operator->() {
            return &*(*this);
        }
        iterator& operator++();
    }
}
```

The standard doesn't explicitly require `difference_type` to be an `Integer` type, but code using `Iterator` (including the definition of `RandomAccessIterator`) expects the usual integer operations to exist.

PointerReference

This concept specifies a pair of types that respectively act as pointers and references to `T`. It is satisfied by builtin pointers and also by user-defined smart pointers. Some smart pointers are associated to plain references (so `reference` is `T&`), while others use a smart-reference type that can be specified as parameter.

```
concept PointerReference<pointer, reference, T> {
    NullablePointer<pointer>;
    class pointer {
        T* operator->() const;
        reference operator*() const;
    }
    class reference {
        pointer operator&() const;
        operator T();
    }
}
```

InputIterator

```
concept InputIterator<iterator> {
    Iterator<iterator>;
    EqualityComparable<iterator>;
    class iterator {
        CopyConstructible<value_type>;
        PointerReference<pointer, reference, value_type>;

        typename postinc_result = auto;
        postinc_result operator++(int);
        class postinc_result {
            value_type operator*();
        }

        axiom (const iterator& itr) {
            ((void) *itr, *itr ) <=> (*itr);
        }
        axiom (const iterator& x, const iterator& y) {
            if (x == y)
                *x <=> *y;
        }
        axiom (iterator itr) {
            *itr++ <=> { T tmp = *itr;
                         ++itr;
                         return tmp; };
        }
    }
}
```

MutablePointerReference

```
concept MutablePointerReference<pointer, reference, T> {
    PointerReference<pointer, reference, T>;
    class reference {
        void operator=(T) const;
    }
}
```

OutputIterator

This concept has two differences compared to the formalization in [SGJ+-05].

Firstly, [SGJ+-05] parametrizes this concept on `value_type`, because an `OutputIterator` can be used with any type that is `CopyAssignable` to the actual type. We prefer to let the user of `OutputIterator` specify such constraint if needed. This case is so uncommon that [SGJ+-05] defines a specialized version of `OutputIterator` that is roughly equivalent to our formalization of `OutputIterator` itself.

```
concept OutputIterator<iterator> {
    Iterator<iterator>;
    class iterator {
        MutablePointerReference<pointer, reference,
                                value_type>;

        typename postinc_result = auto;
        postinc_result operator++(int);
        class postinc_result {
            reference operator*();
            operator const iterator&();
        }

        axiom (iterator itr) {
            &itr = &++itr;
            *itr++ <=> { iterator tmp = *itr;
                         ++itr;
                         return tmp; };
        }
    }
}
```

ForwardIterator

```
concept ForwardIterator<iterator> {
    InputIterator<iterator>;
    DefaultConstructible<iterator>;
    Incrementable<iterator>;

    axiom (iterator x, iterator y) {
        if (x == y)
            ++x == ++y;
    }
}
```

MutableForwardIterator

As in [SGJ+-05], we define mutable versions of the iterator concepts. However, we only require `value_type` to be assignable to `reference` (through `OutputIterator`), so `reference` can be a smart reference type. This makes `std::vector<bool>::iterator` model our concept if a dummy declaration of `operator->()` is provided, for example returning `NULL` (this is not a problem since `bool` has no fields).

```
auto concept MutableForwardIterator<iterator> {
    ForwardIterator<iterator>;
    OutputIterator<iterator>;
}
```

BidirectionalIterator

```
concept BidirectionalIterator<iterator> {
    ForwardIterator<iterator>;
    Decrementable<iterator>;

    axiom (iterator x, iterator y) {
        --(++x) == x;
        if (--x == --y)
            x == y;
        &--x == &x;
    }
}
```

MutableBidirectionalIterator

```
auto concept MutableBidirectionalIterator<iterator> {
    BidirectionalIterator<iterator>;
    OutputIterator<iterator>;
}
```

RandomAccessIterator

```
concept RandomAccessIterator<iterator> {
    BidirectionalIterator<iterator>;
    class iterator {
        Functionalization<iterator, difference_type,
                          operator+, operator+= >
        Functionalization<iterator, difference_type,
                          operator-, operator-= >
        static iterator operator+(difference_type n,
                                  const iterator& itr) {
            return itr + n;
        }
        static iterator operator+(difference_type n,
                                  iterator&& itr) {
            return move(itr) + n;
        }
        iterator& operator-=(difference_type n) {
            return itr += -n;
        }
        reference operator[](difference_type n) const {
            return *(x + n);
        }
        reference operator[](difference_type n) && {
            return *(move(x) + n);
        }
        difference_type operator-(const iterator& y) const;
        bool operator<(const iterator& y) const {
            return y - *this > 0;
        }

        axiom(iterator x, difference_type n) {
            (x + n - x) == n;
        }
        // This is just an axiom, not a function definition
        // because, while it satisfies the operational
        // requirements, it doesn't satisfy the complexity
        // guarantees.
        axiom (iterator itr, difference_type n) {
            ( itr += n ) <=> { difference_type m = n;
                               if (n >= 0)
                                   while (n--) ++itr;
                               else
                                   while (n++) --itr;
                               return itr;
                             }
        }
    }
}
```

MutableRandomAccessIterator

```
auto concept MutableRandomAccessIterator<iterator> {
    RandomAccessIterator<iterator>;
    OutputIterator<iterator>;
}
```

StrictPointerReference

This concept specifies builtin pointers and references, without ruling out user-defined classes implementing smart pointers and smart references. Note that we do *not* require `P = T*` and `P = T&`. This is a striking example of reuse — most of the specification comes from `RandomAccessIterator`.

Together with the following concept map it attaches the iterator inner types to builtin pointers, but the concept can be useful by itself as a concept requirement for generic code requiring pointer-like and reference-like types, when `PointerReference` is not enough.

```
concept StrictPointerReference<P, R, T> {
    typename T;
    class P {
        RandomAccessIterator<P>;
        typename difference_type = ptrdiff_t;
        typename value_type = T;
        typename pointer = P;
        typename reference = R;
        typename iterator_category
            = random_access_iterator_tag;
    }
    class R {
        P operator&();
    }
}

template <typename T>
requires {}
StrictPointerReference<T*, T&, T>;

template <CopyAssignable T>
MutableRandomAccessIterator<T*, T&, T>;
```

The second concept map requires `CopyAssignable` because the expression `*p = x` is only allowed under this condition.

B.5 Containers

The container-related concepts in the standard include special requirements for specific types — for example, the note in `AssociativeContainer` specifying that `value_compare` is the same as `key_compare` for `set` and `multiset`, while it's an ordering on pairs for `map` and `multimap`. Changing the predicate part of a concept for certain models is not an issue, but changing the API part is a problem, because code requiring `AssociativeContainer` must work with all models and won't be able to make any of the two assumptions, and this may prevent it from typechecking.

Another issue in the STL container-related concepts is that some requirements are weakened in derived concepts: for example, even though `AssociativeContainer` requires `Container` and `Container` requires a default constructor in all cases, an `AssociativeContainer` is only required to have a default constructor if the respective `key_compare` has one. Therefore, while these concepts are still meaningful for a human reader, this contradiction will lead to compilation failures when formalized concepts requirements are checked by a compiler.

A possible way out is to add a requires clause to the concept requirement, so that for instance `AssociativeContainer` will only require `Container` when the comparator satisfies certain requirements. This will limit the applicability of concept-constrained code requiring `Container`, because when not all such operations are needed the comparator type will be over-constrained.

The opposite approach is to split the container concepts in more fine-grained concepts, so that all notes about special cases disappear. This makes the specification of both the STL containers and the requirements of template functions using them more verbose, since we have to deal with the increased number of concepts. The feature that allows to express any implication of concept requirements in a concept (see section 5.8) will help reducing the number of concepts, but several additional concepts are needed anyway.

In the following container-related concepts, we have tried to split out optional and special requirement, while still keeping the main concepts used in the standard and their relationships. However, further reformulation of the container-related requirements is needed, using either one of the above approaches or a combination of both; we leave this to future work.

Container

```
concept Container<container> {
    class container {
        typename value_type;
        default typename reference = value_type&;
        default typename const_reference = const value_type&;
        typename iterator;
        typename const_iterator;
        typename difference_type;
        typename size_type;

        Destructible<value_type>;
        ForwardIterator<iterator>;
        ForwardIterator<const_iterator>;
        typename iterator::value_type
            = value_type;
        typename const_iterator::value_type
            = const value_type;
        Integer<difference_type>;
        typename iterator::difference_type
            = difference_type;
        typename const_iterator::difference_type
            = difference_type;
        UnsignedInteger<size_type>;

        default difference_type = ptrdiff_t;
        default size_type = size_t;

        DefaultConstructible<container>;
        MoveConstructible<container>;
        MoveAssignable<container>;
        Destructible<container>;
        Swappable<container>;

        requires { CopyInsertable<value_type, container>; }
        CopyConstructible<container>;

        // The requires clause is not explicitly specified in
        // the C++ standard, but it's needed by most
        // containers (e.g., std::vector).
        requires { CopyInsertable<value_type, container>; }
        CopyAssignable<container>;

        requires { EqualityComparable<value_type>; }
        EqualityComparable<container>;

        void swap(container& x) {
            swap(*this, x);
```

```
        }

        iterator begin();
        iterator end();
        const_iterator begin() const;
        const_iterator end() const;
        const_iterator cbegin() const {
            return begin();
        }
        const_iterator cend() const {
            return end();
        }

        size_type size() const;
        static size_type max_size();
        bool empty() const {
            return begin() == end();
        }

        axiom(container x) {
            // This can't be used as a definition of
            // size() because it doesn't satisfy the
            // complexity requirements.
            x.size() == distance(x.begin(), x.end());
            x.size() <= container::max_size();
        }

        requires { EqualityComparable<value_type>; }
        axiom(container x, container y) {
            if (x == y) {
                x.size() == y.size();
                equal(x.begin(), x.end(), y.begin());
            }
        }

        axiom(difference_type n) {
            if (n >= 0)
                difference_type(size_type(n)) == n;
        }
    }
}
```

ReversibleContainer

```
concept ReversibleContainer<container> {
    class container {
        Container<container>;
        BidirectionalIterator<iterator>;
        BidirectionalIterator<const_iterator>;

        default typename reverse_iterator
            = std::reverse_iterator<iterator>;
        default typename const_reverse_iterator
            = std::reverse_iterator<const_iterator>;
        default reverse_iterator rbegin() {
            return reverse_iterator(end());
        }
        default reverse_iterator rend() {
            return reverse_iterator(begin());
        }
        default const_reverse_iterator rbegin() const {
            return reverse_iterator(end());
        }
        default const_reverse_iterator rend() const {
            return reverse_iterator(begin());
        }
        const_reverse_iterator crbegin() const {
            return rbegin();
        }
        const_reverse_iterator crend() const {
            return rend();
        }
    }
}
```

EmplaceConstructible

```
auto concept EmplaceConstructible<T, container, Args...> {
    typename T;
    class container {
        default typename allocator_type = allocator<T>;
    }
    void allocator_traits<container::allocator_type>
        ::construct(container::allocator_type&, T*, Args...);
}
```

MoveInsertable

```
auto concept MoveInsertable<T, container> {
    EmplaceConstructible<T, container, T&&>;
}
```

CopyInsertable

```
auto concept CopyInsertable<T, container> {
    EmplaceConstructible<T, container, const T&>;
}
```

Allocator

This is the formalization of the `Allocator` requirements described in the section 17.6.3.5 of the C++11 standard. The concept is quite complicated, but it's due to the complexity of the requirements themselves. The size of the formal specification below and of the informal specification in the standard is comparable.

This concept is a good "stress test" to check that the concept features and their interaction are expressive enough, also beyond the simple and sometimes artificial examples used in the previous chapters.

Due to the length of the concept definition, we will make remarks as comments inside the code, at the relevant position, rather than at the end as usual.

```
concept Allocator<allocator, T> {
    class allocator {
        typename value_type = T;
        default typename pointer = T*;
        default typename const_pointer
            = pointer_traits<pointer>::rebind<const T>;
        default typename void_pointer
            = pointer_traits<pointer>::rebind<void>;
        default typename const_void_pointer
            = pointer_traits<pointer>::rebind<const void>;
        default typename difference_type
            = pointer_traits<pointer>::difference_type;
        default typename size_type
            = make_unsigned<difference_type>::type;

        template <typename U>
        class rebind {
            typename other;
            // The following two constraints are weaker than
            // it may seem: we are inside an unconstrained
```

```
        // template, so without additional information we
        // don't know if the instantiation of these types
        // succeeds for a given type U, and we can't
        // check them.
        typename other::void_pointer
            = void_pointer;
        typename other::const_void_pointer
            = const_void_pointer;
    }

    class const_pointer {
        const_pointer(pointer);
        explicit const_pointer(const_void_pointer);
        const T& operator*() const;
        const T* operator->() const {
            return &*(*this);
        }
    }
    class pointer {
        explicit pointer(void_pointer);
        T& operator*() const;
        T* operator->() const {
            return &*(*this);
        }
    }
    class const_void_pointer {
        const_void_pointer(pointer);
        const_void_pointer(const_pointer);
        const_void_pointer(void_pointer);
    }
    class void_pointer {
        void_pointer(pointer);
    }
    RandomAccessIterator<const_pointer, const value_type>;
    RandomAccessIterator<pointer, value_type>;
    NullablePointer<const_void_pointer>;
    NullablePointer<void_pointer>;
    axiom(pointer p, const_pointer cp) {
        &*const_pointer(p) == &*p;
        pointer(void_pointer(p)) == p;
        const_pointer(const_void_pointer(cp)) == cp;
    }

    pointer allocate(size_type);
    default pointer allocate(size_type n, pointer p) {
        return allocate(n);
    }
    void deallocate(pointer, size_type);
```

```
default size_type max_size() {
    return numeric_limits<size_type>::max();
}

EqualityComparable<allocator>;

allocator(allocator&);
allocator(allocator&&);

template <typename U>
allocator(rebind<U>::other&);
template <typename U>
allocator(rebind<U>::other&&);

template <typename C, typename... Args>
void construct(C* c, Args&& args) {
    ::new ((void*)c) C(forward<Args>(args)...);
}
template <typename C>
void destroy(C* c) {
    c->~C();
}

default allocator
select_on_container_copy_construction() {
    return *this;
}

default typename
propagate_on_container_copy_assignment = false_type;
default typename
propagate_on_container_move_assignment = false_type;
default typename
propagate_on_container_swap = false_type;
class propagate_on_container_copy_assignment {
    static const bool value;
}
class propagate_on_container_move_assignment {
    static const bool value;
}
class propagate_on_container_swap {
    static const bool value;
}
Derived<
    propagate_on_container_copy_assignment,
    integral_constant<bool,
        propagate_on_container_copy_assignment
            ::value>>;
Derived<
```

```
                propagate_on_container_move_assignment,
                integral_constant<bool,
                    propagate_on_container_move_assignment
                        ::value>>;
            Derived<
                propagate_on_container_swap,
                integral_constant<bool,
                    propagate_on_container_swap::value>>;
            // The above three Derived constraints mean that the
            // three types are either derived from true_type or
            // false_type.
            // We can't express such constraints directly because
            // we forbid the use of OR for general concept
            // composition.
        }

        template <typename U>
        bool operator==(const allocator& a,
                        const allocator::rebind<U>::other& b) {
            return a == allocator(b);
        }
        template <typename U>
        bool operator!=(const allocator& a,
                        const allocator::rebind<U>::other& b) {
            return !(a == b);
        }
    }
```

To match the specification in the standard, the above concept contains some *unconstrained* templates, and any template requiring `Allocator` needs additional constraints in order to use them, to ensure that the instantiation succeeds.

We have written a two-argument concept "allocator is an Allocator for T" instead of a one-argument "allocator is an Allocator" because we would have had to write all requirements inside an unconstrained template, and such requirements are not enough for any meaningful use of the allocator, since we don't know whether each type or function instantiates correctly (see section 6.8).

The above concept can be reused for concept maps like "my_allocator is an Allocator for all types that satisfy concept C", but the parts that are unconstrained templates above should be rewritten as concept-constrained, to express that the desired types and operations instantiate correctly in this case.

AllocatorAwareContainer

```
concept AllocatorAwareContainer<container> {
    class container {
        typename value_type;
        typename allocator_type;
        typename allocator_type::value_type = value_type;
        allocator_type get_allocator();

        requires { DefaultConstructible<allocator_type>; }
        DefaultConstructible<container>;

        container(allocator_type);

        requires { MoveConstructible<allocator_type>; }
        MoveConstructible<container>;

        requires {
            MoveInsertable<value_type, allocator_type>;
        }
        container(container&&, allocator_type);

        requires {
            MoveInsertable<value_type, container>;
            MoveAssignable<value_type>;
        }
        MoveAssignable<container>;

        requires {
            CopyInsertable<value_type, container>;
            CopyAssignable<value_type>;
        }
        CopyAssignable<container>;

        Swappable<container>;
        void swap(container& x) {
            swap(*this, x);
        }

        requires { DefaultConstructible<allocator_type>; }
        axiom {
            container().get_allocator() == allocator_type();
        }
        axiom(allocator_type a) {
            container(a).get_allocator() == a;
            container(a).empty();
        }
    }
}
```

SequenceContainer

```
concept SequenceContainer<container> {
    class container {
        Container<container>;
        default typename allocator_type
            = allocator<value_type>;

        requires {
            CopyInsertable<container, value_type>;
        }
        container(size_type, const value_type&);

        template <ForwardIterator I>
        requires {
            EmplaceConstructible<value_type, container,
                                 I::value_type>;
            ConvertibleTo<I::value_type, value_type>;
        }
        container(I, I);

        template <InputIterator I>
        requires {
            EmplaceConstructible<value_type, container,
                                 I::value_type>;
            MoveInsertable<value_type, container>;
            ConvertibleTo<I::value_type, value_type>;
        }
        container(I, I);

        // This has the conjunction of the requirements of
        // the two versions above, and prevents ambiguous
        // concept-based overloads when they are both
        // satisfied.
        template <ForwardIterator I>
        requires {
            EmplaceConstructible<value_type, container,
                                 I::value_type>;
            MoveInsertable<value_type, container>;
            ConvertibleTo<I::value_type, value_type>;
        }
        container(I, I);

        container(initializer_list<value_type> x)
            : container(x.begin(), x.end()) {
        }

        template <typename... Args>
```

```
requires {
    EmplaceConstructible<value_type, container,
                         Args...>;
    MoveInsertable<value_type, container>;
    MoveAssignable<value_type, container>;
}
emplace(const_iterator, Args&&...);

requires {
    MoveInsertable<value_type, container>;
    MoveAssignable<value_type>;
}
iterator insert(const_iterator, value_type&&);

requires {
    CopyInsertable<value_type, container>;
    CopyAssignable<value_type>;
}
iterator insert(const_iterator, value_type);

requires {
    CopyInsertable<value_type, container>;
    CopyAssignable<value_type>;
}
iterator insert(const_iterator, size_type,
                value_type);

template <ForwardIterator I>
requires {
    EmplaceConstructible<value_type, container,
                         I::value_type>;
    ConvertibleTo<I::value_type, value_type>;
}
iterator insert(const_iterator, I, I);

template <InputIterator I>
requires {
    EmplaceConstructible<value_type, container,
                         I::value_type>;
    MoveInsertable<value_type, container>;
    MoveAssignable<value_type>;
    ConvertibleTo<I::value_type, value_type>;
}
iterator insert(const_iterator, I, I);

// This has the conjunction of the requirements of
// the two versions above, and prevents ambiguous
// concept-based overloads when they are both
// satisfied.
```

```
template <ForwardIterator I>
requires {
    EmplaceConstructible<value_type, container,
                         I::value_type>;
    MoveInsertable<value_type, container>;
    MoveAssignable<value_type>;
    ConvertibleTo<I::value_type, value_type>;
}
iterator insert(const_iterator, I, I);

iterator insert(const_iterator itr,
                initializer_list<value_type> x) {
    return insert(itr, x.begin(), x.end()) {
}

requires { MoveAssignable<value_type>; }
erase(const_iterator, const_iterator);

void clear();

template <ForwardIterator I>
requires {
    EmplaceConstructible<value_type, container,
                         I::value_type>;
    AssignableTo<I::value_type, value_type>;
    ConvertibleTo<I::value_type, value_type>;
}
void assign(I, I);

template <InputIterator I>
requires {
    EmplaceConstructible<value_type, container,
                         I::value_type>;
    AssignableTo<I::value_type, value_type>;
    MoveInsertable<value_type, container>;
    ConvertibleTo<I::value_type, value_type>;
}
void assign(I, I);

// This has the conjunction of the requirements of
// the two versions above, and prevents ambiguous
// concept-based overloads when they are both
// satisfied.
template <ForwardIterator I>
requires {
    EmplaceConstructible<value_type, container,
                         I::value_type>;
    AssignableTo<I::value_type, value_type>;
    MoveInsertable<value_type, container>;
```

```
            ConvertibleTo<I::value_type, value_type>;
        }
        void assign(I, I);

        requires {
            CopyInsertable<value_type, container>;
            CopyAssignable<value_type>;
        }
        void assign(initializer_list<value_type> x) {
            assign(x.begin(), x.end());
        }

        requires {
            CopyInsertable<value_type, container>;
            CopyAssignable<value_type>;
        }
        void assign(size_type, value_type);
    }
}
```

Several requirements are weaker than the ones described in the standard, because (as noted in the standard) `vector` and `deque` don't satisfy them.

The `StrictSequenceContainer` concept below is provided for classes satisfying the stricter requirements.

StrictSequenceContainer

```
concept StrictSequenceContainer<container> {
    SequenceContainer<container>;
    class container {
        template <InputIterator I>
        requires {
            EmplaceConstructible<value_type, container,
                                 I::value_type>;
            ConvertibleTo<I::value_type, value_type>;
        }
        container(I, I);

        template <typename... Args>
        requires {
            EmplaceConstructible<value_type, container,
                                 Args...>;
        }
        emplace(const_iterator, Args&&...);

        requires {
            MoveInsertable<value_type, container>;
        }
```

```
        iterator insert(const_iterator, T&&);

        requires {
            CopyInsertable<value_type, container>;
        }
        iterator insert(const_iterator, T);

        template <InputIterator I>
        requires {
            EmplaceConstructible<value_type, container,
                                 I::value_type>;
            ConvertibleTo<I::value_type, value_type>;
        }
        iterator insert(const_iterator, I, I);

        erase(const_iterator, const_iterator);

        template <InputIterator I>
        requires {
            EmplaceConstructible<value_type, container,
                                 I::value_type>;
            AssignableTo<I::value_type, value_type>;
            ConvertibleTo<I::value_type, value_type>;
        }
        void assign(I, I);
    }
}
```

FrontAccessSequenceContainer

This and the next few concepts don't have a name in the standard, but are described as optional sequence container requirements.

```
concept FrontAccessSequenceContainer<container> {
    SequenceContainer<container>;
    class container {
        reference front() {
            return *begin();
        }
        const_reference front() const {
            return *begin();
        }
    }
}
```

BackAccessSequenceContainer

```
concept BackAccessSequenceContainer<container> {
    SequenceContainer<container>;
    ReversibleContainer<container>;
    class container {
        reference back() {
            iterator itr = end();
            --itr;
            return *itr;
        }
        const_reference back() const {
            const_iterator itr = end();
            --itr;
            return *itr;
        }
    }
}
```

RandomAccessSequenceContainer

```
concept RandomAccessSequenceContainer<container> {
    FrontAccessSequenceContainer<container>;
    BackAccessSequenceContainer<container>;
    class container {
        RandomAccessIterator<const_iterator>;
        RandomAccessIterator<iterator>;
        reference operator[](size_type n) {
            return *(begin() + n);
        }
        const_reference operator[](size_type n) const {
            return *(begin() + n);
        }
        reference at(size_type n) {
            if (n >= size())
                throw out_of_range();
            return *(begin() + n);
        }
        const_reference at(size_type n) const {
            if (n >= size())
                throw out_of_range();
            return *(begin() + n);
        }
    }
}
```

FrontInsertionSequenceContainer

```
concept FrontInsertionSequenceContainer<container> {
    FrontAccessSequenceContainer<container>;
    class container {
        template <typename... Args>
        requires {
            EmplaceConstructible<value_type, container,
                                 Args...>;
        }
        void emplace_front(Args&&);

        requires { MoveInsertable<value_type, container>; }
        void push_front(value_type&&);

        requires { CopyInsertable<value_type, container>; }
        void push_front(value_type);

        void pop_front();
    }
}
```

BackInsertionSequenceContainer

```
concept BackInsertionSequenceContainer<container> {
    BackAccessSequenceContainer<container>;
    class container {
        template <typename... Args>
        requires {
            EmplaceConstructible<value_type, container,
                                 Args...>;
        }
        void emplace_back(Args&&);

        requires { MoveInsertable<value_type, container>; }
        void push_back(value_type&&);

        requires { CopyInsertable<value_type, container>; }
        void push_back(value_type);

        void pop_back();
    }
}
```

AssociativeContainer

```
concept AssociativeContainer<container> {
    Container<container>;
    class container {
        typename key_type;
        default typename key_compare = less<key_type>;
        typename value_compare;
        BinaryPredicate<key_compare, key_type>;
        BinaryPredicate<value_compare, value_type>;

        requires { CopyConstructible<key_compare>; }
        container(key_compare);

        requires { DefaultConstructible<key_compare>; }
        DefaultConstructible<container>;

        template <InputIterator I>
        requires {
            ConvertibleTo<I::value_type, value_type>;
            CopyConstructible<key_compare>;
            EmplaceConstructible<value_type, container,
                                 I::value_type>;
        }
        container(I, I, key_compare);

        template <InputIterator I>
        requires {
            ConvertibleTo<I::value_type, value_type>;
            DefaultConstructible<key_compare>;
            EmplaceConstructible<value_type, container,
                                 I::value_type>;
        }
        container(I, I);

        requires {
            DefaultConstructible<key_compare>;
            CopyInsertable<value_type, container>;
        }
        container(initializer_list<value_type>);

        requires {
            CopyInsertable<value_type, container>;
            CopyAssignable<value_type>;
        }
        container& operator=(initializer_list<value_type>);

        key_compare key_comp() const;
        value_compare value_comp() const;
```

```
template <typename... Args>
requires {
    EmplaceConstructible<value_type, container,
                         Args...>;
}
iterator emplace_hint(const_iterator,
                      Args&&... args);

requires {
    MoveInsertable<value_type, container>;
}
iterator insert(const_iterator, value_type&&);

requires {
    CopyInsertable<value_type, container>;
}
iterator insert(const_iterator, value_type);

template <InputIterator I>
requires {
    ConvertibleTo<I::value_type, value_type>;
    EmplaceConstructible<value_type, container,
                         I::value_type>;
}
void insert(I, I);

requires {
    CopyInsertable<value_type, container>;
}
void insert(initializer_list<value_type>);

size_type erase(key_type);
iterator erase(const_iterator);
iterator erase(const_iterator, const_iterator);

void clear() {
    erase(begin(), end());
}

const_iterator find(key_type) const;
iterator find(key_type);

size_type count(key_type) const;

const_iterator lower_bound(key_type) const;
const_iterator upper_bound(key_type) const;
iterator lower_bound(key_type);
iterator upper_bound(key_type);
```

```
        pair<const_iterator, const_iterator>
        equal_range(const key_type& k) const {
            return make_pair(a.lower_bound(k),
                             a.upper_bound(k),);
        }

        pair<iterator, iterator>
        equal_range(const key_type& k) {
            return make_pair(a.lower_bound(k),
                             a.upper_bound(k),);
        }
    }
}
```

Note that we don't require `key_compare` to be an ordering relation, because that's a requirement on the *value* of that type used to construct the container. Other values of the same type may not satisfy the requirement.

UniqueAssociativeContainer

This concept formalizes the associative container requirements for containers with unique keys.

```
concept UniqueAssociativeContainer<container> {
    AssociativeContainer<container>;
    class container {
        template <typename... Args>
        requires {
            EmplaceConstructible<value_type, container,
                                 Args...>;
        }
        pair<iterator, bool> emplace(Args&&... args);

        requires { MoveInsertable<value_type, container>; }
        pair<iterator, bool> insert(value_type&&);

        requires { CopyInsertable<value_type, container>; }
        pair<iterator, bool> insert(value_type);
    }
}
```

MultipleAssociativeContainer

This concept formalizes the associative container requirements for containers with multiple keys.

```
concept MultipleAssociativeContainer<container> {
    AssociativeContainer<container>;
    class container {
        template <typename... Args>
        requires {
            EmplaceConstructible<value_type, container,
                                 Args...>;
        }
        iterator emplace(Args&&... args);

        requires { MoveInsertable<value_type, container>; }
        iterator insert(value_type&&);

        requires { CopyInsertable<value_type, container>; }
        iterator insert(value_type);
    }
}
```

MappingContainer

This concept formalizes the associative container requirements for mapping containers (i.e., `map` and `multimap`).

```
concept MappingContainer<container> {
    AssociativeContainer<container>;
    class container {
        typename mapped_type;
        Destructible<mapped_type>;
    }
}
```

UnorderedAssociativeContainer

```
concept UnorderedAssociativeContainer<container> {
    Container<container>;
    AllocatorAwareContainer<container>;
    class container {
        typename key_type;
        Destructible<key_type>;

        class hasher {
            size_t operator()(key_type);
        }
```

```
typename key_equal;
BinaryPredicate<key_equal, key_type>;

class local_iterator {
    Iterator<local_iterator>;
    typename iterator_category
        = iterator::iterator_category;
    typename value_type
        = iterator::value_type;
    typename pointer
        = iterator::pointer;
    typename reference
        = iterator::reference;
}
class const_local_iterator {
    Iterator<const_local_iterator>;
    typename iterator_category
        = const_iterator::iterator_category;
    typename value_type
        = const_iterator::value_type;
    typename pointer
        = const_iterator::pointer;
    typename reference
        = const_iterator::reference;
}

requires {
    CopyConstructible<hasher>;
    CopyConstructible<key_equal>;
}
container(size_type, hasher, key_equal);

requires {
    CopyConstructible<hasher>;
    DefaultConstructible<key_equal>;
}
container(size_type, hasher);

requires {
    DefaultConstructible<hasher>;
    DefaultConstructible<key_equal>;
}
container(size_type);

requires {
    DefaultConstructible<hasher>;
    DefaultConstructible<key_equal>;
}
DefaultConstructible<container>;
```

```
template <InputIterator I>
requires {
    ConvertibleTo<I::value_type, value_type>;
    EmplaceConstructible<value_type, container,
                         I::value_type>;
    CopyConstructible<hasher>;
    CopyConstructible<key_equal>;
}
container(I, I, size_type, hasher, key_equal);

template <InputIterator I>
requires {
    ConvertibleTo<I::value_type, value_type>;
    EmplaceConstructible<value_type, container,
                         I::value_type>;
    CopyConstructible<hasher>;
    DefaultConstructible<key_equal>;
}
container(I, I, size_type, hasher);

template <InputIterator I>
requires {
    ConvertibleTo<I::value_type, value_type>;
    EmplaceConstructible<value_type, container,
                         I::value_type>;
    DefaultConstructible<hasher>;
    DefaultConstructible<key_equal>;
}
container(I, I, size_type);

template <InputIterator I>
requires {
    ConvertibleTo<I::value_type, value_type>;
    EmplaceConstructible<value_type, container,
                         I::value_type>;
    DefaultConstructible<hasher>;
    DefaultConstructible<key_equal>;
}
container(I, I);

requires {
    CopyConstructible<value_type, container>;
    DefaultConstructible<hasher>;
    DefaultConstructible<key_equal>;
}
container(initializer_list<value_type>);

requires {
```

```
        CopyConstructible<value_type>;
        CopyConstructible<hasher>;
        CopyConstructible<key_equal>;
    }
    CopyConstructible<container>;

    requires {
        CopyInsertable<value_type, container>;
        CopyAssignable<value_type>;
    }
    void operator=(initializer_list<value_type> il);

    requires {
        CopyInsertable<value_type, container>;
        CopyAssignable<value_type>;
        CopyConstructible<value_type, container>;
        DefaultConstructible<hasher>;
        DefaultConstructible<key_equal>;
        MoveAssignable<container>;
    }
    void operator=(initializer_list<value_type> il) {
        *this = container(il);
    }

    hasher hash_function() const;
    key_equal key_eq() const;

    template <typename... Args>
    requires {
        EmplaceConstructible<value_type, container,
                             Args...>;
    }
    iterator emplace_hint(const_iterator, Args&&...);

    requires { MoveInsertable<value_type, container>; }
    iterator insert(const_iterator, value_type&&);

    requires { CopyInsertable<value_type, container>; }
    iterator insert(const_iterator, value_type);

    template <InputIterator I>
    requires {
        ConvertibleTo<I::value_type, value_type>;
        EmplaceConstructible<value_type, container,
                             I::value_type>;
    }
    void insert(I first, I last);

    template <InputIterator I>
```

```
        requires { CopyInsertable<value_type, container>; }
        void insert(initializer_list<value_type>);

        size_type erase(key_type);
        iterator erase(const_iterator);
        iterator erase(const_iterator, const_iterator);
        void clear();

        const_iterator find(key_type) const;
        iterator find(key_type);
        size_type count(key_type) const;

        pair<const_iterator, const_iterator>
        equal_range(key_type) const;

        pair<iterator, iterator>
        equal_range(key_type);

        size_type bucket_count() const;
        size_type max_bucket_count() const;
        size_type bucket(key_type) const;
        size_type bucket_size(size_type) const;

        const_local_iterator begin(size_type) const;
        const_local_iterator end(size_type) const;
        local_iterator begin(size_type);
        local_iterator end(size_type);

        const_local_iterator cbegin(size_type n) const {
            return begin(n);
        }
        const_local_iterator cend(size_type n) const {
            return end(n);
        }

        float load_factor() const;
        float max_load_factor() const;
        void max_load_factor(float);

        void rehash(size_type);

        void reserve(size_type n) {
            rehash(ceil(n / max_load_factor()));
        }
    }
}
```

UniqueUnorderedAssociativeContainer

```
concept UniqueUnorderedAssociativeContainer<container> {
    UnorderedAssociativeContainer<container>;
    class container {
        template <typename... Args>
        requires {
            EmplaceConstructible<value_type, container,
                                 Args...>;
        }
        pair<iterator, bool> emplace(Args&&...);

        requires { MoveInsertable<value_type, container>; }
        pair<iterator, bool> insert(value_type&&);

        requires { CopyInsertable<value_type, container>; }
        pair<iterator, bool> insert(value_type);
    }
}
```

MultipleUnorderedAssociativeContainer

```
concept MultipleUnorderedAssociativeContainer<container> {
    UnorderedAssociativeContainer<container>;
    class container {
        template <typename... Args>
        requires {
            EmplaceConstructible<value_type, container,
                                 Args...>;
        }
        iterator emplace(Args&&...);

        requires { MoveInsertable<value_type, container>; }
        iterator insert(value_type&&);

        requires { CopyInsertable<value_type, container>; }
        iterator insert(value_type);
    }
}
```

UnorderedMappingContainer

This concept formalizes the unordered associative container requirements for mapping containers (i.e., `unordered_map` and `unordered_multimap`).

```
concept UnorderedMappingContainer<container> {
    UnorderedAssociativeContainer<container>;
    class container {
        typename mapped_type;
        Destructible<mapped_type>;

        requires {
            CopyAssignable<key_type>;
            CopyAssignable<mapped_type>;
            CopyAssignable<hasher>;
            CopyAssignable<key_equal>;
            CopyInsertable<value_type, container>;
        }
        CopyAssignable<container>;
    }
}
```

UnorderedSetContainer

This concept formalizes the unordered associative container requirements for set containers (i.e., `unordered_set` and `unordered_multiset`).

```
concept UnorderedSetContainer<container> {
    UnorderedAssociativeContainer<container>;
    class container {
        requires {
            CopyAssignable<key_type>;
            CopyAssignable<hasher>;
            CopyAssignable<key_equal>;
            CopyInsertable<value_type, container>;
        }
        CopyAssignable<container>;
    }
}
```

B.6 Misc concepts

BitmaskType

```
concept BitmaskType<Bitmask, elements...> {
    class Bitmask {
        default typename int_type
            = underlying_type<Bitmask>::type;
        explicit Bitmask(int_type);
        explicit operator int_type(Bitmask);
        Integer<int_type>; // NOTE: not in the standard.
    };
    CopyAssignable<Bitmask>; // NOTE: not in the standard.

    constexpr Bitmask operator&(Bitmask x, Bitmask y) {
        return Bitmask(Bitmask::int_type(x)
                     & Bitmask::int_type(y));
    }
    constexpr Bitmask operator|(Bitmask x, Bitmask y) {
        return Bitmask(Bitmask::int_type(x)
                     | Bitmask::int_type(y));
    }
    constexpr Bitmask operator^(Bitmask x, Bitmask y) {
        return Bitmask(Bitmask::int_type(x)
                     ^ Bitmask::int_type(y));
    }
    constexpr Bitmask operator~(Bitmask x) {
        return Bitmask(~Bitmask::int_type(x));
    }
    Bitmask& operator&=(Bitmask& x, Bitmask y) {
        x = x & y;
        return x;
    }
    Bitmask& operator|=(Bitmask& x, Bitmask y) {
        x = x | y;
        return x;
    }
    Bitmask& operator^=(Bitmask& x, Bitmask y) {
        x = x ^ y;
        return x;
    }

    // The & of distinct elements must be zero.
    constexpr Bitmask... elements;
    axiom() {
        (Bitmask::int_type(elements) != 0)...;
    }
}
```

Hash

```
concept Hash<H, Key> {
    typename Key;
    CopyConstructible<H>;
    Destructible<H>;
    class H {
        size_t operator()(Key&);
        size_t operator()(Key);
    }
}
```

UnaryTypeTrait

```
concept UnaryTypeTrait<trait, value_type> {
    template <typename T>
    requires { }
    class trait {
        DefaultConstructible<trait>;
        CopyConstructible<trait>;
        value_type value;
    }
}
```

BinaryTypeTrait

```
concept BinaryTypeTrait<trait, value_type> {
    template <typename T, typename U>
    requires { }
    class trait {
        DefaultConstructible<trait>;
        CopyConstructible<trait>;
        value_type value;
    }
}
```

TransformationTrait

```
concept TransformationTrait<trait> {
    template <typename T>
    requires { }
    class trait {
        typename type;
    }
}
```

Ratio

```
concept Ratio<R> {
    class R {
        static constexpr intmax_t num;
        static constexpr intmax_t den;
        typename type;
    }
    Specialization<R, ratio>;
}
```

Clock

```
concept Clock<clock> {
    class clock {
        typename rep;
        typename period;
        typename duration = chrono::duration<rep, period>;
        class time_point {
            typename duration = clock::duration;
        }
        static const bool is_steady;
        static time_point now();
        Integer<rep>;
        Ratio<period>;
        Specialization<time_point, chrono::time_point>;
    }
}
```

TrivialClock

```
concept TrivialClock<clock> {
    Clock<clock>;
    class clock {
        Regular<rep>;
        Regular<duration>;
        Regular<time_point>;
        static time_point now() noexcept;
        // Not expressible:
        // TrivialClock<time_point::clock>;
    }
}
```

The fact that, for every type `T`, `TrivialClock<T>` requires `TrivialClock<T::time_point::clock>` can't be expressed with the concept features we analyzed. In this case we would need a fixed point operator, but it's not a widely useful feature, so we don't consider its inclusion.

CharTraits

```
concept CharTraits<traits> {
    class traits {
        typename char_type;
        typename int_type;
        typename off_type;
        typename pos_type;
        typename state_type;
        DefaultConstructible<state_type>;
        CopyConstructible<state_type>;
        CopyAssignable<state_type>;
        // These requirements are not explicitly
        // specified in the standard.
        SemiRegular<int_type>;
        SemiRegular<char_type>;

        static bool
        lt(const char_type&, const char_type&);
        StrictWeakOrder<lt, char_type>;

        static bool
        eq(const char_type& a, const char_type& b) {
            return !lt(a, b) && !lt(b, a);
        }
        Equivalence<eq, char_type>;

        // This is marked default because only the sign of
        // the result is specified. The implementation
        // may return different values.
        default static int
        compare(const char_type* p, const char_type* q,
                size_t n) {
            for (size_t i = 0; i < n; ++i) {
                if (lt(p[i], q[i]))
                    return -1;
                if (lt(q[i], p[i]))
                    return 1;
            }
            return 0;
        }

        static size_t length(const char_type* p) {
            size_t i = 0;
            while (!eq(p[i], char_type()))
                ++i;
            return i;
        }
```

```
const char_type*
find(const char_type* p, size_t n, char_type c) {
    for (size_t i = 0; i < n; ++i)
        if (eq(p[i], c))
            return p + i;
    return 0;
}

static void assign(char_type&, char_type);

static char_type*
assign(char_type* s, size_t n, char_type c) {
    for (size_t i = 0; i < n; ++i)
        assign(s[i], c);
    return s;
}

static char_type*
move(char_type* s, const char_type* p, size_t n) {
    if (s + n <= p || p + n <= s || s <= p) {
        for (size_t i = 0; i < n; ++i)
            assign(s[i], p[i]);
    } else {
        size_t i = n;
        while (i > 0) {
            --i;
            assign(s[i], p[i]);
        }
    }
    return s;
}

static char_type*
copy(char_type* s, const char_type* p, size_t n) {
    for (size_t i = 0; i < n; ++i)
        assign(s[i], p[i]);
    return s;
}

static bool eq_int_type(int_type, int_type);
static int_type to_int_type(char_type);
static int_type eof();

axiom (char_type c, char_type d) {
    eq(c, d) == eq_int_type(to_int_type(c),
                            to_int_type(d));
}
axiom (int_type k) {
    if (k != eof()) {
```

```
                !eq_int_type(k, eof());
                !eq_int_type(eof(), k);
                eq_int_type(eof(), eof());
            }
        }

        int_type not_eof(int_type e);

        axiom (int_type k) {
            if (eq_int_type(k, eof())
                !eq_int_type(not_eof(k), eof());
            else
                not_eof(k) == k;
        }

        char_type to_char_type(int_type);

        axiom (char_type c) {
            eq(to_char_type(to_int_type(c)), c);
        }
    }
}
```

SeedSequence

```
concept SeedSequence<S> {
    class S {
        typename result_type;
        UnsignedInteger<result_type>;
        DefaultConstructible<S>;

        template <InputIterator I>
        requires { UnsignedInteger<I::value_type>; }
        S(I ib, I ie);

        S(initializer_list<result_type> il)
            : S(il.begin(), il.end() {
        }

        template <RandomAccessIterator I>
        requires { UnsignedInteger<I::value_type>; }
        void generate(I rb, I re);

        size_t size() const;

        template <OutputIterator I>
        void param(I ob);
    }
}
```

UniformRandomNumberGenerator

```
concept UniformRandomNumberGenerator<generator> {
    class generator {
        typename result_type;
        UnsignedInteger<result_type>;

        static result_type min();
        static result_type max();
        result_type operator()();

        axiom (generator g) {
            min() < max();
            g() >= min();
            g() <= max();
        }
    }
}
```

RandomNumberEngine

```
concept RandomNumberEngine<engine> {
    UniformRandomNumberGenerator<engine>;
    class engine {
        DefaultConstructible<engine>;
        CopyConstructible<engine>;
        CopyAssignable<engine>;
        EqualityComparable<engine>;

        engine(result_type);

        template <SeedSequence S>
        engine(S&);

        void seed();
        void seed(result_type);
        template <SeedSequence S>
        void seed(S&);

        void discard(unsigned long long n) {
            for (unsigned long long i = 0; i < n; ++i)
                (*this)();
        }

        template <CharTraits traits>
        static basic_ostream<traits::char_type, traits>&
        operator<<(basic_ostream<traits::char_type, traits>&,
                   engine);
```

```
        template <CharTraits traits>
        static basic_istream<traits::char_type, traits>&
        operator>>(basic_istream<traits::char_type, traits>&,
                   engine);
    }
}
```

RandomNumberEngineAdaptor

```
concept RandomNumberEngineAdaptor<engine, base_engine> {
    RandomNumberEngine<engine>;
    class engine {
        const base_engine& base() const;

        static bool operator==(const engine& x,
                               const engine& y) {
            return x.base() == y.base();
        }
        // The other requirements are about the
        // implementation of 'engine', and don't
        // extend its API.
    }
}
```

RandomNumberDistribution

```
concept RandomNumberDistribution<distribution> {
    class distribution {
        typename result_type;
        Arithmetic<result_type>;

        class param_type {
            typename distribution_type = distribution;

            CopyConstructible<param_type>;
            CopyAssignable<param_type>;
            EqualityComparable<param_type>;
        }

        DefaultConstructible<distribution>;
        CopyConstructible<distribution>;
        CopyAssignable<distribution>;
        EqualityComparable<distribution>;

        distribution(param_type);

        void reset();
```

```
        param_type param() const;
        void param(param_type);

        axiom (param_type p) {
            distribution(p).param() == p;
        }
        axiom (distribution d, param_type p) {
            (d.param(p), d.param()) == p;
        }

        template <UniformRandomNumberGenerator generator>
        result_type operator()(generator&);
        template <UniformRandomNumberGenerator generator>
        result_type operator()(generator&, param_type);

        result_type min() const;
        result_type max() const;

        template <CharTraits traits>
        static basic_ostream<traits::char_type, traits>&
        operator<<(basic_ostream<traits::char_type, traits>&,
                   engine);

        template <CharTraits traits>
        static basic_istream<traits::char_type, traits>&
        operator>>(basic_istream<traits::char_type, traits>&,
                   engine);
    }
}
```

BasicLockable

```
concept BasicLockable<L> {
    class L {
        void lock();
        void unlock() noexcept;
    }
}
```

Lockable

```
concept Lockable<L> {
    BasicLockable<L>;
    class L {
        bool try_lock();
    }
}
```

TimedLockable

```
concept TimedLockable<L> {
    Lockable<L>;
    class L {
        template <typename rel_time>
        requires {
            Specialization<rel_time, chrono::duration>;
        }
        bool try_lock_for(rel_time);

        template <typename abs_time>
        requires {
            Specialization<abs_time, chrono::duration>;
        }
        bool try_lock_until(abs_time);
    }
}
```

We used the `Specialization` requirement, but any conforming implementation needs stricter requirements in order to use the parameter. Such requirements should probably be replaced by specifications of the API of `chrono::duration` and `chrono::time_point`, respectively.

Mutex

```
concept Mutex<mutex> {
    Lockable<mutex>;
    class mutex {
        DefaultConstructible<mutex>;
        Destructible<mutex>;

        mutex(const mutex&) = delete;
        mutex(mutex&&) = delete;
    }
}
```

TimedMutex

```
concept TimedMutex<mutex> {
    TimedLockable<mutex>;
}
```

Bibliography

[BDH-09] A. H. Bagge, V. David and M. Haveraaen.
"The axioms strike back: testing with concepts and axioms in C++11".
In *Proceedings of the eighth international conference on Generative programming and component engineering*, pages 15-24. ACM New York, NY, USA, 2009. doi:10.1145/1621607.1621612

[BDH-11] A. H. Bagge, V. David and M. Haveraaen.
"Testing with axioms in C++11".
In *Journal of Object Technology*, Volume 10, pages 10:1-32, 2011. doi:10.5381/jot.2011.10.1.a10

[Catsfoot website] Catsfoot's website.
`http://catsfoot.sourceforge.net/`

[ConceptClang-09] The website of the 2009 release of ConceptClang.
`http://www.generic-programming.org/software/ConceptClang/`

[ConceptClang-11] The website of the 2011 release of ConceptClang.
`https://www.crest.iu.edu/projects/conceptcpp/`

[GB-92] J. A. Goguen and R. M. Burstall.
"Institutions: abstract model theory for specification and programming".
In *J. ACM*, Volume 39, Issue 1, pages 95-146. ACM New York, NY, USA, 1992. doi:10.1145/147508.147524

[GJL+-07] R. Garcia, J. Jarvi, A. Lumsdaine, J. Siek and J. Willcock.
"An extended comparative study of language support for generic programming".
In *Journal of Functional Programming*, Volume 7, Issue 2, pages 145-205. Cambridge University Press New York, NY, USA, 2007. doi:10.1017/S0956796806006198

[Gre-08] D. Gregor.
"Type soundness and optimization in the concepts proposal".
ISO/IEC JTC1/SC22/WG21 N2576, 2008.
`http://open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2576.pdf`

[GS-06] D. Gregor and B. Stroustrup.
"Concepts (Revision 1)".

ISO/IEC JTC1/SC22/WG21 N2081, 2006.
http://open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2081.pdf

[GSW-08] D. Gregor, B. Stroustrup, J. Widman and J. Siek.
"Proposed wording for concepts (revision 6)".
ISO/IEC JTC1/SC22/WG21 N2676, 2008.
http://open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2676.pdf

[JGW+-06] J. Järvi, D. Gregor, J. Willcock, A. Lumsdaine and J. Siek.
"Algorithm specialization in generic programming: challenges of constrained generics in C++".
In *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 272-282. ACM New York, NY, USA, 2006. doi:10.1145/1133981.1134014

[JWL-04] J. Järvi, J. Willcock and A. Lumsdaine.
"Algorithm specialization and concept-constrained genericity".
In *Concepts: a Linguistic Foundation of Generic Programming*, 2004.
http://osl.iu.edu/publications/prints/2004/jarvi04:algorithm_specialization.pdf

[RSM-09] G. D. Reis, B. Stroustrup and A. Meredith.
"Axioms: Semantics Aspects of C++ Concepts".
ISO/IEC JTC1/SC22/WG21 N2887, 2009.
http://open-std.org/jtc1/sc22/wg21/docs/papers/2009/n2887.pdf

[SGJ+-05] J. Siek, D. Gregor, J. Järvi, R. Garcia, J. Willcock and A. Lumsdaine.
"Concepts for C++0x".
ISO/IEC JTC1/SC22/WG21 N1758, 2005.
http://open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1758.pdf

[Sie-12] J. G. Siek. Edited by J. Gibbons.
"The C++0x "Concepts" Effort".
In *Proceedings of the 2010 international spring school conference on Generic and Indexed Programming*, pages 175-216. Springer Berlin Heidelberg, Berlin, Germany, 2012. doi:10.1007/978-3-642-32202-0_4

[SL-00] J. Siek and A. Lumsdaine.
"Concept checking: binding parametric polimorphism in C++".
In *First Workshop on C++ Template Programming*, 2000.
http://osl.iu.edu/~jsiek/concept_checking.pdf

[SL-05] J. Siek and A. Lumsdaine.
"Essential language support for generic programming".
In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 73-84. ACM New York, NY, USA, 2005. doi:10.1145/1065010.1065021

[SM-09] A. A. Stepanov and P. McJones.
"Elements of Programming".
Addison-Wesley Professional, 2009.
http://books.google.it/books?id=tjOlkl7ecVQC

[SR-03] B. Stroustrup and G. D. Reis.
"Concepts — Design choices for template argument checking".
ISO/IEC JTC1/SC22/WG21 N1522, 2003.
http://open-std.org/jtc1/sc22/wg21/docs/papers/2003/n1522.pdf

[SR-03b] B. Stroustrup and G. D. Reis.
"Concepts — syntax and composition".
ISO/IEC JTC1/SC22/WG21 N1536, 2003.
http://open-std.org/jtc1/sc22/wg21/docs/papers/2003/n1536.pdf

[SR-05] B. Stroustrup and G. D. Reis.
"A concept design (rev. 1)".
ISO/IEC JTC1/SC22/WG21 N1782, 2005.
http://open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1782.pdf

[SS-11] B. Stroustrup and A. Sutton.
"Design of concept libraries for C++".
In *Proceedings of the 4th international conference on Software Language Engineering*, pages 97-118. Springer Berlin Heidelberg, Berlin, Germany, 2011.
doi:10.1007/978-3-642-28830-2_6

[SS-12] B. Stroustrup and A. Sutton.
"A Concept Design for the STL".
ISO/IEC JTC1/SC22/WG21 N3351, 2012.
http://open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3351.pdf

[Str-03] B. Stroustrup.
"Concept checking — A more abstract complement to type checking".
ISO/IEC JTC1/SC22/WG21 N1510, 2003.
http://open-std.org/jtc1/sc22/wg21/docs/papers/2003/n1510.pdf

[Str-09] B. Stroustrup.
"Simplifying the use of concepts".
ISO/IEC JTC1/SC22/WG21 N2906, 2009.
http://open-std.org/jtc1/sc22/wg21/docs/papers/2009/n2906.pdf

[Van-06] D. Vandevoorde.
"Modules in C++ (revision 4)".
ISO/IEC JTC1/SC22/WG21 N2073, 2006.
http://open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2073.pdf