A Uniform Terminology for C++ Concepts

Larisse Voufo, Andrew Lumsdaine
Indiana University
{lvoufo, lums}@crest.iu.edu

Abstract

The purpose of this note is to specify a uniform terminology for the design and implementation of concepts for C++, based on influential literature on generic programming with concepts. The terminology results from an ongoing work in specifying an infrastructure for implementing C++ concepts, called ConceptClang. We solicit feedback and hope that the terminology will facilitate conversations about C++ concepts as efforts in language support for the feature in question evolve.

In this specification, we define each term with an explanation of the rationale behind it, especially when it may conflict with existing terminology, and relate it to ongoing design and implementation efforts. The process highlights key components of the concepts feature and their diverse relationships with either structural or named conformances for matching types to concepts.

1. Introduction

While the *concepts* feature has been part of C++ library developments since at least the development of the Standard Template Library (STL) [2, 17, 24], its support as a language feature has been under development and anticipated over the last decade [3, 11, 14, 26, 28]. Various design and implementation philosophies for concepts, as either library or language features, have been and continue to be explored. For example, in terms of implementation, there are: a library-based implementation of concepts (BCCL) [21, 22] (and its related extensions [5, 8]), a context-free ConceptC++ to pure C++ transformation tool [7], a GCC-based prototype implementation of concepts called ConceptGCC [12, 13], a `G` to C++ translation [23], an ongoing prototype implementation dubbed Concepts-Light [27], and an ongoing design-independent implementation called ConceptClang [29, 30].

In terms of designs, we mainly have the C++ standard documents N2914[3] and N3351[26], respectively dubbed *pre-Frankfurt* and *Palo Alto* designs. There are several proposed extensions of these designs (e.g. [14, 28]) that can be found in archived C++ standard committee papers [6] and resources linked from the ConceptClang website [29].

While the above listing is not exhaustive, its vastness already indicates the complexity of the task of designing concepts for C++, as well as the potential confusion in terminology between differing design philosophies. In anticipation of these problems, in this note, we propose a uniform terminology for C++ concepts that we hope will facilitate ongoing and future discussions about the design and implementation of the feature. The terminology combines ideas from influential literature on generic programming with concepts [2, 25], related efforts in formalization [31], and the alternative philosophies listed above.

For each term we introduce, we explain the rationale behind it and relate it to ongoing implementation efforts. For clarity and conciseness, our arguments are primarily based on the "*Elements of Programming*" (EOP) book [25] and the three compiler implementations ConceptGCC, ConceptClang, and Concepts-Light; since we find the book and compilers to fairly represent all alternative philosophies of primary interest. To explain, we first note that most approaches at designing concepts for C++ stem from the design of the STL, which shares foundational roots with EOP. Afterall, the origin of concepts can be traced back to the algebraic specification language called Tecton [15]. Thus, we consider alternative design philosophies relatable to ideas from EOP. As a result, those ideas form the primary basis for the rationale behind our terminology.

Further, we also note that all but two alternative implementations (listed above) are exclusively based on the pre-Frankfurt design: Concepts-Light and ConceptClang. While Concepts-Light is exclusively based on the Palo Alto design and primarily focused on expressing template requirements in a concise and natural manner, ConceptClang is both independent from and extensible to both alternatives. In fact, ConceptClang supports design-specific extensions of its infrastructure for both designs. Another primary difference between the implementation philosophies of ConceptClang from that of other compilers is that ConceptClang treats components of concepts as first class entities, rather than syntactic sugar versions of existing entities. For instance, while ConceptGCC, like all other implementations that are exclusively based on the pre-Frankfurt design, go from the perspective of treating concepts as class templates, Concepts-Light goes from the perspective of treating concepts as boolean constant expressions. We hope to clarify the above distinctions as we introduce the terms in our proposed terminology in the next sections of this note.

The rest of this note is structured as follows. In the next section, we list the essential components of any design or implementation for concepts and briefly relate them to the different notions of conformances for matching types to concepts. In Sects. 2 thru 5, we describe each essential component and the description introduces terms that we relate to alternative philosophies as briefly and concisely as possible. Each of these sections ends with a listing of the terms it introduces, as a reminder of what the reader is expected to understand upon reading the section. A lack of familiarity for any term listed is likely and indication of our omitting important information. So, we are open to any comment, suggestion or question.[1] Sect. 6 concludes the note with a discussion on a key component of the concepts feature, namely *concept model archetypes* (cf. Sect. 4).

From now on, we assume the reader's familiarity with the alternative design and implementation philosophies listed above.

1.1 The Components

Based on the elements of generic programming with concepts [4, 10, 30, 31], we can abstract the following main components of design or implementation:

- concept definitions, with
    - refinements (specification), and
    - requirements (specification),
- concept models, with
    - refinements satisfaction, and
    - requirements satisfaction,
- constrained template definitions, with
    - constraints specification, and
    - type-checking of the body with
        - entity reference building, and
- constrained template uses, with
    - constraints satisfaction, and
    - instantiation of the body with
        - entity reference rebuilding.

[1] Emails are provided in the title section.

In Sects. 2 thru 5, we will define these components in the above order, and the process will drive our terminology specification.

Meanwhile, we note that different conformances for matching types to concepts, whether structural or nominal, can be applied at the points of satisfying either constraints, for each constrained template use, or refinements and requirements, for each concept model. We explain the different conformances in Sect. 3.

2. Concept Definitions

A *concept definition* specifies a grouping of requirements that generic components impose on their types. In C++, generic components consist of template declarations. A concept definition thus specifies constraints that a template declaration can impose on its type parameters. For example, in the pre-Frankfurt design, one can write

```
concept MyConcept<typename P> :
      CopyConstructible<P> {
  void foo(P);
}
```

to mean that declarations that the template definition uses must match the type signature `void foo(P);` as well as the type signatures in the `CopyConstructible` concept.

We refer to the `CopyConstructible` concept as a *refinement* of the `MyConcept` concept, and the declaration `void foo(P);` is a *requirement* of the `MyConcept` concept. (We could be overloading the word "refinement" in a weird way, but we would like to think of "refinement" as "the state of being refined".[2])

In the Palo Alto design, one can rewrite the above as

```
concept MyConcept<typename P> =
    CopyConstructible<P> &&
  requires (P a) {
    foo(a);
  };
```

but, this time, to mean that uses of the name `foo` must match the pattern `foo(a)`, for every value `a` of type `P`.

EOP defines a concept as

> "a description of requirements on one or more types stated in terms of the existence and properties of procedures, type attributes, and type functions defined on the types".

For a specified concept, both the pre-Frankfurt and Palo Alto designs may describe the same set of requirements, but only state them differently. For instance, while the pre-Frankfurt design states the requirements in terms of pseudo-signatures, with notions of associated functions, types, and requirements, the Palo Alto design states the requirements in terms of use-patterns extended with optional annotations for type sameness and convertibility. In fact, the Palo Alto design explicitly states that one can effectively translate the notions of associated types and requirements in the pre-Frankfurt design into a notion of *type functions* (along side appropriate use-patterns) in the Palo Alto design.

Therefore, for our purposes of uniformly communicating about concepts, we propose to adopt the terminology from the EoP book and refer to any declaration or expression inside a concept definition, i.e., a pseudo-signature, or use-pattern, or anything else, as *requirement* or *requirement specification*. Refinements are also requirements, but since they tend to require a different kind of reasoning, we will treat them separately from other kinds of requirements (cf. Sect. 5).

In the same train of thought, let us call representations of concept definitions *concept definition declarations*. concept definition declarations contain representations of requirements and refinements. Depending on the design or implementation, requirements can be represented as declarations or expressions. However, refinements can be represented uniformly as *concept model archetypes* to be defined later in Sect. 4.

[2] So, if `MyConcept` refines `CopyConstructible`, then `CopyConstructible` is refined by `MyConcept`, and thus is a refinement of `MyConcept`.

2.1 Implementation

ConceptClang: ConceptClang represents concept definitions as declarations which can hold other declarations. Thus, requirements are represented as declarations, and refinements are represented as concept model archetypes.

ConceptGCC: In ConceptGCC, concept definition declarations are simply class template declarations. This approach stemmed from BCCL and has inspired work on automatically generating concept model archetypes [32] and a context-free ConceptC++ to C++ transformation [7]. That said, requirements are represented as declarations, and refinements as inherited classes.

Concepts-Light: Concepts-Light represents concept definition declarations as constant boolean expressions. In general, such expressions are simply template declarations such as a `constexpr bool` function templates, class templates convertible to `bool`, or aliases to types that are convertible to `bool`. That said, requirements are represented as constant boolean expressions and refinements as constraints to the expression templates.

2.2 Terminology Recap

Concept definition. Concept definition declaration. Requirement (specification). Refinement (specification).

3. Concept Models

According to EOP, a type that satisfies the requirements of a concept is called *a model of* the said concept. In general, and particularly when using concepts for generic programming, just knowing that a type satisfies the requirements of a concept is not enough. One must also know how the type satisfies the said requirements. Thus, it is customary to think of a model of a concept in terms of both the type that it is, and how it models the said concept. In fact, a previous work on formalizing concepts based on its origin as algebraic specifications [31] defines the model of a concept as a "structure" matching the said concept, both structurally and semantically. Consequently, we call *concept model* a statement of both

- *what* type models which concept and
- *how* the type satisfies the requirements of the said concept.

In other words, a concept model is also a statement of the modeling relationship between a type and a con-

```
1 concept_map MyConcept<int> {
2   void foo(int) { ... }
3 }
```

Figure 1: An explicitly specified concept model, in the pre-Frankfurt design.

```
1 concept_map MyConcept<int> {
2   CopyConstructible<int>; //<-- link to
3     // the refinement's model.
4
5   void foo(int) { ... }
6 }
```

(a) For the pre-Frankfurt design.

```
1 concept_map MyConcept<int> {
2   CopyConstructible<int>; //<-- link to
3     // the refinement's model.
4
5   ... foo(...int...) { ... } //<--
6     // All viable candidates for call
7     // foo(a), where a is of type int.
8 }
```

(b) For the Palo Alto design.

Figure 2: An internal representation of a concept model

cept, which expresses the *what* and the *how* of the relationship.

We refer to *how* a type models a concept as *requirement satisfaction* and note that this term is currently in use with a conflicting meaning. Indeed, its current meaning is assigned to a different term in our terminology, i.e., *constraints satisfaction*, that we will introduce in Sect. 5. We find this change in terminology necessary for maintaining consistency with respect to all components of concepts as herein defined.

On the different notions of conformance: A concept model can be provided either explicitly by a user – a.k.a. via *explicit modeling*, or implicitly by a compiler – a.k.a. via *implicit modeling*. When provided by a user, we say that types are matched to concepts *nominally*—or via *named conformance*. Otherwise, we say that they are matched *structurally*—or via *structural conformance*. For example, the pre-Frankfurt design allows explicit modeling via the `concept_map` construct, as in Fig. 1. The pre-Frankfurt design also allows implicit modeling under some special cases, e.g. the use of `auto` concepts. On the other hand, the Palo Alto design allows only implicit modeling.

Under named conformance, a compiler has no knowledge of the modeling relationships between types and concepts, except for those indicated by each explicit, e.g. `concept_map`, declaration it encounters. For each such explicit declaration, it generates an internal representation similarly to that in Fig. 2a that it uses when needed. Under structural conformance, the compiler knows about as many modeling relationships as it can generate based on existing types and concepts. Essentially, it generates an internal representation when needed, either as in Fig. 2a (or equivalent) for the pre-Frankfurt design, or as in Fig. 2b (or equivalent) for the Palo Alto design. In essence, while the pre-Frankfurt design brings all matching implementations that it finds within the surrounding scope into the internal representation, the Palo Alto design brings in all viable candidates for the call `foo(a)`, with `a` of type `int`.

The notion of conformance resurfaces when generating internal representations, since requirements can be satisfied either explicitly by a user, or implicitly by a compiler. We say that

> *the use of named conformance for matching types to concepts does not preclude the use of structural conformance for specifying how types match concepts. In contrast, the use of structural conformance for matching types to concepts implies the use of structural conformance for specifying how types match concepts.*

For example, with the explicit modeling in Fig. 1, the user explicitly stated how each requirement of the concept is satisfied, by providing an implementation for the specified `void foo(P);` in Line 2. However, the pre-Frankfurt design allows one to implicitly satisfy each requirement, by simply not providing an implementation, e.g. omitting Line 2, and thus leaving it up to the compiler to derive the appropriate implementation, e.g. to cover Line 5 in Fig. 2a. With implicit modeling, one has no choice but to leave all requirements satisfaction up to the compiler.

To sum up, while the pre-Frankfurt design supports both structural and named conformances for both levels of matching types to concepts, i.e., the *what* and the *how*, the Palo Alto design supports only structural conformance at both levels. Of the two levels, the *what* makes a particular difference in the checking of constrained template uses, as we will see in Sect. 5. □

Concept model declarations, templates and archetypes: Analogously to concept definitions, let us call representations of concept models *concept model dec-*

larations. Further, since concept models can match either concrete types, as in Figs. 1, 2a, and 2b, or generic types, via template forms of concept model declarations, let us respectively emphasize the distinction in the matched types with the notions of *concrete concept model* and *concept model template*. Likewise, let us call their distinctive representations *concrete concept model declarations* and *concept model template declarations*.

The distinction in matched types helps highlight more areas of differences between alternative designs. For example, we notice that concept model templates are only a concern when explicit modeling is supported. Therefore, while the Palo Alto design only needs to generate concrete concept models, the pre-Frankfurt design generates concept model template declarations every time a template form of the `concept_map` construct is used.

Let us make another distinction between concept models, only this time, the emphasis is on the *how* level. At times, as we will see in Sect. 4, one may lack complete information about the matched type and it may be necessary to generate a temporary concept model to use as placeholder for when one has complete information about the said type. Let us call such a temporary concept model *concept model archetype*, and emphasize that concrete concept models cannot be concept model archetypes, nor concept model templates. Also, concept model templates cannot be concept model archetypes either. This is a direct consequence of the way in which each kind of concept model is used. Sect. 4 will provide further details about concept model archetypes.

By now, it should be clear that

> *while modeling mechanisms may differ, by whether and how structural or named conformance is supported, the notion of concept model is still essential to the design and implementation of concepts.*

In fact, the notion of model is inherent to concepts-based reasoning in that, one cannot (should not) speak of a concept without a notion of an existing model for that concept. Indeed, the Palo Alto design is developed based on this rationale, among others.

To close our description of concept models, note that the notion of *concept model* replaces the ongoing notion of *concept map*, which can easily be confused with the use of the `concept_map` construct. Thus, the new notion effectively decouples the statement of a modeling relationship from the modeling mechanism in place.

3.1 Implementation

ConceptClang: Similarly to concept definitions, concept models are represented as declarations holding other declarations. The only difference is that the declarations concept model declarations hold are represent statements of requirements satisfaction.

For the pre-Frankfurt design, the satisfaction of a requirement is essentially a three-stage name lookup process[3], as described in the N2914 document[3]. For the Palo Alto design, the requirement satisfaction follows a different path. Essentially, ConceptClang internally produces the program snippet illustrated in Fig. 2b via appropriate type substitution and checking that the expression `foo(a)` is valid. In other words, during the type-substitution, it gathers all valid function candidates for the call `foo(a)` and adds the candidates to the concept model declaration.

For both designs, refinements are satisfied as if they were constraints on constrained template definitions (cf. Sect. 4). In other words, the process of satisfying refinements essentially reuses the procedure for satisfying the constraints on a constrained template, at its point of use [30].

ConceptGCC: As with concept definitions, ConceptGCC keeps things simple and expresses concept models as class template specializations of the class templates that represent the concepts that they model. Requirements are still represented as declarations, and refinements are specializations of the inherited classes that represent concept refinements.

Concepts-Light: Similarly to ConceptGCC, Concepts-Light implicitly expresses concept models as constant boolean expressions or expression template specializations, with appropriately updated refinements in the requires clauses. Requirements are still represented as constant boolean expressions, and refinements are the updated refinements.

[3] While the N2914 document itself does not use the terminology of "three-stage name lookup", we find it fitting since name lookup can be repeated three times.

3.2 Terminology Recap

Concept model. Concrete concept model. Concept model template. Concept model declaration. Concrete concept model declaration. Concept model template declaration. Requirement satisfaction. Refinement satisfaction. Explicit modeling. Implicit modeling. Named conformance. Structural conformance.

4. Constrained Template Definitions

When it comes to alternative design and implementation philosophies, this is where things start to get the most interesting.

A *constrained template definition* is a template definition with constraints specified on its template parameters. For example, in either the pre-Frankfurt or Palo Alto design, one may define a generic function as

```
template<MyConcept T1, typename T2>
void gen_func(T1 x) {
  foo(x);
}
```

which is equivalent to

```
template<typename T1, typename T2>
requires MyConcept<T1>
void gen_func(T1 x) {
  foo(x);
}
```

This template is constrained since the type `T1` is expected to satisfy the requirements of the `MyConcept` concept, when the template is used.

We refer to the constraints on the template parameters as *constraints specification* to parallel the notion of *requirement specification* in concept definitions.

A constraints specification is a list of constraints which serve two purposes: they act as predicates and they provide a scope for name resolution. Conceptually, these constraints serve as placeholders for when concrete concept models are available, and indicate which names are valid to use in the body of the constrained template.

Names that are valid to use are either associated to the concepts expressed by the constraints, or correspond to other constrained template definitions. To preserve certain optimizations (without hindering separate type-checking), ConceptC++ also allows uses of names that do not depend on any constrained template parameter, e.g. non-dependent call expressions.

To follow the terminology from the drafts of the C++ standard, and for generality, we refer to uses of names, e.g. `foo(x)`, as *entity references*. Further, we refer to the implementation process of checking their validity, followed by building representations of them when valid, as *entity reference building*. (In programming languages, entity reference building is typically referred to as *name binding* [1, 9, 16, 18–20].)

Entity reference building involves name lookup, which depends on the scope provided by the constraints specification. We refer to the said scope as *restricted scope* and note that the pre-Frankfurt design currently refers to the same scope as *requirement scope*. However, we find that adopting the pre-Frankfurt term would add confusion with respect to the terms of *requirement (specification)* and *requirement satisfaction* in Sects. 2 and 3. We also do not find a term that specifically refers to constraints, e.g. *constraints scope*, applicable enough because restricted scopes are also used in concept definitions and models to check statements of satisfaction (e.g. implementations) of requirements.

On the notion of archetype: A restricted scope keeps track of each specified constraint, and name lookup in a restricted scope searches through all the constraints that the scope keeps track of. How the constraints are represented or kept track of may be implementation-dependent. But, due to syntactic and semantic similarities between constraints and concept models, one may think of a constraint as a special kind of concept model that lacks concrete type information—information that is available at the point of use of a template. Therefore, we treat constraints as temporary concept models, with minimal statements of requirement satisfaction, that serve as placeholders for concrete concept models; And call representations of these special concept models *concept model archetypes*. For instance, one may represent the constraint `MyConcept<T1>` as in Fig. 3. It is importaant to note that no concrete statement of satisfaction for the requirements is provided. Instead, type-substituted versions of the requirements are provided.

The notion of *concept model archetype* (CMA) we herein introduce is simply an application of that of *archetype* as used in ConceptGCC and BCCL. ConceptGCC generates archetypes to temporarily represent two different kinds of entities: types or concept models. We clarify the distinction between the two kinds with the notions of *type archetype* and CMA. It is important to note that CMAs can not be considered concrete since concrete concept models are fully satisfied, e.g. imple-

```
concept_map MyConcept<T1> {
  CopyConstructible<T1>; //<-- link to
  // the refinement's model archetype.

  void foo(T1);
}
```

(a) For the pre-Frankfurt design

```
concept_map MyConcept<T1> {
  CopyConstructible<T1>; //<-- link to
  // the refinement's model archetype.

  requires (T1 a) {
    foo(a);
  };
}
```

(b) For the Palo Alto design

Figure 3: A concept model archetype

mented. CMAs end up playing a key role in relating all components of concepts. We document this role in the conclusion (Sect. 6). □

Entity reference building involves a bit more than name lookup in the Palo Alto design. Essentially, entity references must be checked against the use-patterns in CMAs constructed based on the constraints specification. For example, one must check that the function call `foo(x)`, in the definition of the constrained template `gen_func` above, indeed matches the use-pattern `foo(a)` where `a` has type `T1`. We refer to this stage of entity reference rebuilding as *expression validation*, since expressions are validated against use-patterns.

Expression validation can be executed at any point during entity reference building, based on the implementation. Of all the implementations herein surveyed, only ConceptClang currently implements it.

4.1 Implementation

Perhaps arguably, this section highlights the broad adaptability and flexibility provided by the ConceptClang implementation model, especially when it comes to the treatment of CMAs.

ConceptClang: ConceptClang respectively extends the representations of templates and template parameter scopes with constraints specifications. Likewise, name lookup is extended accordingly, searching for names in CMAs representing specified constraints. CMAs are constructed as concept model declarations, except that instead of *satisfying* each requirement as in Sect. 3, representations of requirements are *type-substituted* over instead. The result for both the pre-Frankfurt and Palo Alto design are illustrated in Fig. 3.

The behavior of name lookup in restricted scope slightly differs in both designs. For the pre-Frankfurt design, it is straightforward since substituted representations of requirements are visible named declarations. For the Palo Alto design, it helps to note that ConceptClang introduces dummy declarations to represent each name used in each use-pattern. For example, for the use-pattern `foo(a)` in the `MyConcept` concept definition, ConceptClang introduces a declaration like `void foo(P);` in the representation of definition and links the use-pattern to that declaration. The same declaration is type-substituted into the constructed `MyConcept<T1>` CMA and accordingly linked from the substituted use-pattern. Thus, the type-substituted declaration is also named and visible by name lookup.

When building the entity reference `foo(x)`, name lookup finds the substituted declaration `void foo(T1);`, dummy or not, in the `MyConcept<T1>` CMA; and treats the template parameter `T1` as a concrete, non-dependent, type when checking the entity reference.

Essentially, this constitutes the end of the checking of a template definition for the pre-Frankfurt design, but not for the Palo Alto design. For the Palo Alto design, ConceptClang proceeds with expression validation. Expression validation turns out to be somewhat of a complex mechanism that is best triggered at the end of parsing a top level expression, and requires that some entity references be marked for validation upon their checking. The details on this fall outside the scope of this note.

ConceptClang introduces type archetypes into concept definition declarations to represent member declarations associated with concepts like `CopyConstructible`. For example, say the concept parameter for the definition of `CopyConstructible` is named `P`. To represent the copy constructor for `P`, a type archetype is introduced into the declaration of `CopyConstructible` that contains a declaration of the constructor. Like dummy declarations for the Palo Alto design, these type archetypes are named, type-substituted into CMAs and visible by name lookup. The resulting effect is similar to what we get with BCCL's archetypes.

ConceptGCC: ConceptGCC constructs CMAs as class template specializations (like concrete concept models) and extends the representations of template parameter scopes similarly to ConceptClang, but with a slight difference: It explicitly generates type archetypes upon parsing each template parameter and subsequently adds them to the template parameter scope. For example, in the `gen_func` example above, ConceptGCC generates three archetypes: two type archetypes for both `T1` and `T2`, and a CMA for `MyConcept<T1>`.

The process can be adapted into ConceptClang as generating CMAs for specified constraints and dummy CMAs. For example, in our running example with `gen_func`, ConceptClang would generate two CMAs: one for `MyConcept<T1>`, containing the type archetype for `T1`, and a dummy one containing an empty type archetype for `T2`.

Clearly, this process is different from that currently performed by ConceptClang, since it constructs CMAs only for specified constraints. The addition or omission of the dummy CMAs may constitute the difference between whether partially constrained template definitions are completely checked or not. In other words, the `gen_func` example is fully checked since `T2` is not used in the body, but what if `T2` were used in the body? Should uses of `T2` be checked as if `T2` is constrained or not?

Using the ConceptClang implementation model, decisions on topics like this can be separated from the implementation details and, perhaps arguably, expressed and adapted more clearly. Beyond CMAs, consider potential extensions that allow names associated to concepts to be qualified not just by type names, but also namespaces and other scope qualifiers. When necessary, such qualifiers can simply be processed in a manner similar to type archetypes. Thus, the ConceptClang implementation model offers more room for adaptability, which is usually helpful in maintaining both backward and future compatibility of C++ libraries, as they transition from unconstrained templates to constrained templates.

Concepts-Light: Concepts-Light is primarily concerned with expressing constraints concisely and naturally. Thus, it does not check the body of constrained templates and simply expresses the requirements as a conjunction of constant boolean expressions (like refinements on concept declarations). Here, CMAs are simply constant boolean expression template specializations that, conceptually, provide no scope for name resolution.

Extending Concepts-Light with this functionality is conceivably a future extension and a next step towards a complete implementation for concepts. Either way, all alternatives can be implemented as an extension of ConceptClang.

4.2 Terminology Recap

Constrained template. Constrained template definition. Constraint. Constraints specification. Archetype. Type archetype. Concept model archetype. Restricted scope. Entity reference. Entity reference building. Expression validation.

5. Constrained Template Uses

This is another area in which alternative design and implementation philosophies make interesting differences.

In any case, the use of a constrained template, e.g `gen_func` defined above in Sect. 4, can be as simple as

```
gen_func<int, char>(...);
```

Upon parsing such a function template call, a compiler first checks that the provided template arguments, e.g. `<int, char>`, match the template parameters, e.g. `<typename T1, typename T2>`. When the template parameters are constrained, e.g. by `MyConcept<T1>`, it also checks that the template arguments satisfy the specified constraints by looking up appropriate concrete concept models, e.g. `MyConcept<int>`.

Depending on the design, when no such concept model is found, the compiler may attempt to implicitly generate them, based on available type information. This is the case in the Palo Alto design, in general, and in the pre-Frankfurt design, when either the specified constraint is an implicit concept, or under special cases.

We refer to the checking of template arguments against the specified constraints as *constraints satisfaction* and note that it corresponds to the procedure that the checking of concept models uses to satisfy concept refinements [30]. Essentially, one can effectively express refinement satisfaction as constraints satisfaction where concept definitions are treated as template definitions, concept models as template uses, and refinement specifications as constraints specifications.

The term *constraints satisfaction* replaces what used to be *requirement satisfaction*. We find this renaming necessary for consistency in our terminology, unless we find another generic term for *concept requirement (specification)*.

Once the concrete concept models, e.g. `MyConcept<int>`, are found, the compiler proceeds to instantiating the constrained template definition, replacing dependent name uses with non-dependent name uses. For all names used that are associated to concepts, e.g. `foo(x)` in `gen_func` above (Sect. 4), where

- `foo()` is a possibly undefined substituted declaration in a CMA, i.e., `void foo(T1);` or a dummy `foo()`, and
- `x` is of type `T1`,

the name uses must be replaced with uses of corresponding names in concrete concept models, e.g. `foo(x)`, where

- `foo()` is defined in a concrete concept model, as `void foo(int);` or any other viable variation, and
- `x` is of type `int`.

We call this final compilation step *entity reference rebuilding*, since it basically repeats entity reference building with concrete type information (including concrete concept models resulting from constraints satisfaction).

5.1 Implementation

ConceptClang: ConceptClang extends type-checking of templates, SFINAE, and overload resolution with constraints satisfaction, and extends template instantiation with entity reference rebuilding. No new construct is particularly created, except for helpers to the SFINAE extension and concepts-based overloading.

ConceptGCC: ConceptGCC implicitly covers constraints satisfaction and entity reference rebuilding for free. Since all names that may need rebuilding were essentially replaced by qualified forms of the same names in which the qualifiers are class template specializations. The instantiation of these specializations implicitly performs all necessary concept model lookup, and thus automatically rebuilds the entity references.

Concepts-Light: Concepts-Light does not consider CMAs during entity reference building within constrained template definitions; which precludes a necessity for entity reference rebuilding. Nevertheless, similarly to ConceptGCC, Concepts-Light implicitly performs constraints satisfaction for free, by evaluating the specialized constant expressions; which indicates success (failure) with a result of `true` (`false`).

5.2 Terminology Recap

Constraints satisfaction. Entity reference rebuilding.

6. Conclusion

We have specified a uniform terminology for C++ concepts, based on a review of influential literature on generic programming with concepts and different approaches to designing, implementing and formalizing concepts. Our specification was driven by a listing of elements of generic programming with concepts and their relationships with either structural and nominal conformance in matching types to concepts. With respect to the notion of conformance, we learned two important things. First, there are two levels at which the notion of conformance applies: constraints satisfaction and requirements satisfaction. Second, we learned that the notion of concept model is essential to the concepts feature, independently of the details of the conformance – or modeling mechanism – in place. However, concept model templates are only as essential as named conformance – or explicit modeling – is supported.

Overall, we learned three other important things. First, the checking of concept models reuse constraints satisfaction to satisfy concept refinements. Second, entity reference rebuilding is not always necessary and its need varies based on the design or implementation. Third, and perhaps the most interesting observation:

> Concept model archetypes are a particularly essential component of implementing concepts, allowing for greater reuse and modularity, since
>
> 1. they link the requirements specified in concept definitions (requirement specification) with their uses in constrained template definitions (constraints specification), and
> 2. they tell constraints satisfaction which concept model to look for or generate.
> 3. As placeholders, they are used to
>   - represent concept refinements,
>   - represent constraints specified on constrained templates,
>   - satisfy uses of templates from within template definitions, and
>   - satisfy refinements when checking concept model templates.

At the very least, any complete implementation of concepts should implement the above functionality of concept model archetypes, or any equivalent functionality.

Acknowledgments

All members of the ConceptClang team contributed to this note, via several meetings on concepts and ConceptClang. The team consists of the author, Marcin Zalewski, Jeremiah Willcock and Andrew Lumsdaine. In particular, Marcin Zalewski has reviewed an earlier draft of this note and provided insightful comments. We would also like to thank Dave Abrahams for a careful scrutiny on an earlier draft of the note.

References

[1] Harold Abelson and Gerald Jay Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, 2nd edition, September 1996.

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

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

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

[5] Matt Calabrese. Boost.generic: Concepts without concepts. `https://github.com/boostcon/2011_presentations/raw/master/thu/Boost.Generic.pdf`, 2011.

[6] C++ Standards Committee. C++ standards committee papers. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/`.

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

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

[9] D.P. Friedman, M. Wand, and C.T. Haynes. *Essentials of Programming Languages*. MIT Press, 2nd edition, 2001.

[10] Ronald Garcia, Jaakko Jarvi, Andrew Lumsdaine, Jeremy Siek, and Jeremiah Willcock. An extended comparative study of language support for generic programming. *J. Funct. Program.*, 17(2):145–205, March 2007.

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

[12] Douglas Gregor. ConceptGCC: Concept extensions for C++. `http://www.generic-programming.org/software/ConceptGCC/`, September 2008.

[13] Douglas Gregor and Jeremy Siek. Implementing concepts. Technical Report N1848=05-0108, ISO/IEC JTC1/SC22/WG21—The C++ Standards Committee, August 2005.

[14] Magne Haveraaen. Institutions, property-aware programming and testing. In *Proceedings of the 2007 Symposium on Library-Centric Software Design*, LCSD '07, pages 21–30, New York, NY, USA, 2007. ACM.

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

[16] John C. Mitchell. *Concepts in Programming Languages*. Cambridge University Press, 1st edition, October 2002.

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

[18] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[19] Michael L. Scott. *Programming Language Pragmatics, Second Edition*. Morgan Kaufmann, 2 edition, November 2005.

[20] Robert W. Sebesta. *Concepts of Programming Languages*. Addison-Wesley, 10th edition, 2012.

[21] J. Siek and A. Lumsdaine. The boost concept check library (bccl). `http://www.boost.org/doc/libs/1_51_0/libs/concept_check/concept_check.htm`, 2007.

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

[23] Jeremy G. Siek. *A Language for Generic Programming*. PhD thesis, Indiana University, Indianapolis, IN,

USA, August 2005. AAI3183499.

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

[25] Alexander Stepanov and Paul McJones. *Elements of Programming*. Addison-Wesley Professional, 2009.

[26] Bjarne Stroustrup and Andrew Sutton. A concept design for the stl. Technical Report N3351=12-0041, ISO/IEC JTC1/SC22/WG21—The C++ Standards Committee, January 2012.

[27] Andrew Sutton and Bjarne Stroustrup. Template Constraints (DRAFT), October 2012.

[28] Xiaolong Tang and Jaakko Järvi. Concept-based optimization. In *Proceedings of the 2007 Symposium on Library-Centric Software Design*, LCSD '07, pages 97–108, New York, NY, USA, 2007. ACM.

[29] Larisse Voufo. ConceptClang Project. `http://www.crest.iu.edu/projects/conceptcpp/`, May 2012.

[30] Larisse Voufo, Marcin Zalewski, and Andrew Lumsdaine. ConceptClang: an implementation of c++ concepts in clang. In *Proc. 7th ACM SIGPLAN workshop on Generic programming*, pages 71–82. ACM, 2011.

[31] Jeremiah Willcock, Jaakko Järvi, Andrew Lumsdaine, and David Musser. A formalization of concepts for generic programming. In *Concepts: a Linguistic Foundation of Generic Programming at Adobe Tech Summit*. Adobe Systems, April 2004.

[32] Jeremiah Willcock, Jeremy Siek, and Andrew Lumsdaine. Caramel: A concept representation system for generic programming. In *Second Workshop on C++ Template Programming*, Tampa, Florida, October 2001.