A Formalization of Concepts for Generic Programming

Jeremiah Willcock[1], Jaakko Järvi[1], Andrew Lumsdaine[1], and David Musser[2]

[1] Open Systems Lab, Indiana University
Bloomington IN, USA
{jewillco|jajarvi|lums}@osl.iu.edu
[2] Rensselaer Polytechnic Institute
musser@cs.rpi.edu

Abstract. Generic programming is a paradigm for systematic design and classification of software components for optimal reuse. It has been the guiding methodology behind the design of the C++ Standard Template Library and numerous other C++ libraries. Generic programming starts with algorithms, seeking to identify the minimal requirements on types necessary for correct and efficient execution of an algorithm or family of algorithms, leading to generic algorithms that can be applied to arguments of many different types. The term *concept* is used in this context to mean a set of abstractions (such as types) whose membership is determined by a set of requirements. Despite their importance, concepts are not explicitly represented in traditional programming languages. C++, in particular, lacks any means to express concepts, which has forced all concept development to be done outside the language and thus be unavailable to C++ compilers for static checking or optimization. Several other languages do have features than can approximate certain aspects of concepts, but what is missing is a uniform, language-independent definition of concepts. The definition of concepts in this paper provides a unified framework for realizing concept approximations in existing languages, and we present such descriptions for several popular languages used for generic programming.

1 Introduction

Generic programming is an emerging programming paradigm for creating highly reusable domain-specific software libraries. It is a systematic approach to software reuse that focuses on finding the most general (or abstract) formulations of algorithms and then implementing efficient generic representations of them. Fundamental to realizing generic algorithms is the notion of abstraction: generic algorithms are specified in terms of abstract properties of types, not in terms of particular types. In generic programming the term *concept* is used to mean a set of abstractions (typically types) whose membership is defined by a set of requirements [1–3]. The expression of requirements in a concept is referred to as a *concept description*. Concept requirements may be semantic as well as syntactic. A concept taxonomy for a particular domain forms the basis for development of a generic library for that domain. A concept taxonomy guides in organizing a library, selecting components for it, and providing precise yet accessible documentation.

Concept taxonomies have been used in generic programming research as early as 1981 [1, 4], but their use was popularized much later by the C++ community. The Standard Template Library (STL) [5, 6], now part of the C++ Standard Library [7], was the starting point. Other widely used libraries have since been developed following the paradigm, such as the Boost Graph Library [8], the Matrix Template Library [9], the Boost Iterator Library [10], the μBLAS Library [11], and numerous others.

While concept descriptions are an established means to document the requirements of generic algorithms, there is not a uniform definition for concepts themselves. Instead, the different approaches to concept description imply their definition. For example, the conventions used in the STL documentation [12, 13] imply one definition of concepts, while the C++ standard library specification uses a slightly different style, implying a different definition. The *Caramel* concept documentation system provides a structured (semi-formal) notation in XML close to the STL concept description format [14]. More formal (but still incomplete) approaches can be found in the field of *formal concept analysis* [2, 3] and in algebraic specification theory using sets of multi-sorted algebras [15].

While the semi-formal approaches serve practical generic programming well, and formal concept definitions succeed in describing the underlying principles of the design methodology, there are many issues not yet addressed. In particular, a formulation of concepts should address the following:

- C++ does not have a language construct for representing concepts or enforcing constraints expressed as concepts, whereas other languages can bound parametric polymorphism using various constraint mechanisms. It is important to be able to analyze these techniques with respect to generic programming in some uniform way. Connections between different languages' constraint mechanisms have been pointed out by the authors [16, 17] and others [18]. In particular, we are interested in a more precise account of how ML signatures, Haskell type classes, and interfaces in object-oriented languages compare to syntactic concept descriptions.
- In formulations of concepts as multi-sorted algebras, type dependencies must be expressed outside the definition of a concept's signature and are thus unenforceable with syntactic checking. In generic programming practice there are many functional dependencies between the types, which require that certain types can be determined from certain other types. Practical programming languages have different mechanisms for expressing such type associations, and place varying degrees of restrictions on which associations are allowed. Expressing these type associations in a formal definition of concepts would allow these restrictions to be more carefully analyzed.
- Informal concept descriptions specify required functionality as *valid expressions* instead of function signatures. A formal concept definition should encompass both approaches for requirements specification.
- Whether the conformance between types and concepts is structural (based only on satisfying the concept requirements) or nominal (based on an explicit declaration stating that a particular type models a concept) significantly affects generic programming in practice. A formal model of concepts should be able to capture this

distinction. In addition, different languages locate models for concepts differently, and a definition of concepts should capture this as well.
– Informal concept descriptions include complexity guarantees and semantic requirements, such as pre- and postconditions of operations. Thus, a formal view of concepts should be able to include such requirements.

In this paper we provide a precise definition for concepts, as they are used in practical generic programming. Equipped with this definition, we are able to address all of the above issues. The main contribution of the paper is to clarify the nature of concepts in generic programming, relating them to existing work on specifying generic interfaces. We note also that there is a recent initiative to add built-in support for concepts into C++ [19–21]; we hope that this paper will provide input to that design process.

Our definition of a concept is the following (and is developed fully in the body of this paper):

Definition: A *concept* C is defined as a triple $\langle R, P, L \rangle$, where R is a signature and P is a predicate whose domain is all structures which match R. A structure X is defined to *model* C whenever X matches R and $P(X)$ holds. In C, L is a set of functions whose common codomain is the set of models of C. In this definition, R represents the syntactic requirements of the concept, and P represents its semantic requirements. The function L is a *model lookup function* which finds models of a concept from a subset of the concept's members (commonly referred to as the *concept parameters* or *main types*). A signature R, similarly to an ML signature [22] is defined as a tuple $\langle T, V, S, E \rangle$, where T represents type requirements, V represents value requirements, S represents nested structure requirements, and E represents same-type requirements. □

The remainder of this paper is organized as follows: Section 2 describes the generic programming paradigm and reviews currently established concept notations. In Section 3 we evolve a precise definition of concepts for generic programming, starting from an algebraic concept definition, and augmenting it with constructs necessary for capturing concept features used in practical generic programming. Section 4 shows how this definition provides a unified view for generics in a variety of programming languages allowing different constraint mechanisms to be compared and contrasted—as we do for C++ templates, Haskell type classes, ML signatures, and object-oriented interfaces. That section extends our earlier work [16], where we explored practical issues of generic programming using these language features as concept representations. Concepts are closely related to algebraic specifications, and thus Section 5 describes related work on algebraic specification languages, as well as computer algebra systems and other programming languages.

2 Background

A general way to define the term "concept" as it relates to generic programming is as follows:

Definition: A *concept* is a set R of *requirements* together with a set A of *abstractions*, such that an abstraction is included in A if and only if it satisfies all of the requirements in R. □

This definition of concept is adapted from the field of formal concept analysis and concept lattices [2, 3]. In this literature, the inhabitants of concepts are called *objects* rather than abstractions, and the examples commonly given of concepts rarely include ones in which the objects themselves are sets of other entities. That is, the objects are not themselves abstractions, like types are in programming languages. On the other hand, in the development of concepts for generic programming, such as in the Tecton concept description language [15, 23, 24], the emphasis has been on concepts as sets of abstractions, but there has been less effort toward extending concept taxonomies into complete lattices.

An abstraction (commonly a type, or a sequence of types) that satisfies the requirements of a concept is said to *model* that concept. A *generic algorithm* can then be defined as one that works correctly and efficiently for every model of a concept.

A concept is said to *refine* another concept if the requirements the first concept places on its abstractions are a superset of the requirements on the second concept's abstractions. It is this refinement relation that generates a lattice of concepts, but since in generic programming practice there is little need to isolate all concepts of such a lattice, the term *concept taxonomy* is more often used. Formal concept analysis deals with extracting concepts from a set of existing objects and their properties, whereas concepts in generic programming originate from the analysis of a set of algorithms and their requirements on parameter types. Formal concept analysis is thus an analysis tool, generic programming a design tool.

This general definition of concepts covers a broad range of software constructs. Normally, the abstractions used are types or sets of types. This definition of concepts, however, also includes *algorithm concepts*, in which the abstractions are algorithm implementations [25–27]. The requirements used to constrain algorithms are then semantic properties and specifications of the behavior of the algorithms. However, the general definition of concepts is too general to provide much insight to the underlying programming language constructs or their use. Hence, we are interested in a more refined definition that still covers the practical uses of concepts in generic programming to describe requirements on parameters to generic algorithms. The rest of this section reviews such concept descriptions and their components, which motivate our definition.

The best-known example of the use of concept taxonomies in guiding software design is the C++ Standard Template Library (STL). The style of concept descriptions used for the STL concepts [12, 13] has been adopted in the C++ community. In this context, concept descriptions are understood to consist of four different kinds of requirements on one or more types, called the *main types* or *parameters* of the concept: associated types, valid expressions (or function signatures), semantic constraints, and complexity guarantees. The *associated types* of a concept specify mappings from the modeling type(s) to other collaborating types. *Valid expressions* or *function signatures* specify the operations that must be implemented for the modeling types. *Semantic constraints* describe properties of the valid expressions, such as pre- and postconditions. *Complexity guarantees* describe the maximum allowed algorithm complexities for particular operations.

Concepts can be classified as syntactic or semantic: a *syntactic concept* can contain associated type requirements and valid expressions, whereas a *semantic concept*

also includes semantic constraints and complexity guarantees [15]. Figure 1 shows an example of an STL-style semi-formal description of the Bidirectional Iterator concept. We can identify several components of interest from this concept description, which a formal definition of concepts must be able to support:

- Within the definition of a generic algorithm, it is often useful to be able to access related types to those given as type arguments to the function ("associated types"). For example, given an iterator type (e.g., a type that models Bidirectional Iterator, one often needs to access the type of the values that the iterator points to, such as finding ***int*** for iterators from the container *vector*<***int***>. These requirements are often expressed using *traits classes* in C++, which are a way of representing type functions [28].
- In contrast to requirements on associated types, a concept can also describe requirements on multiple independent types. As an example, a mathematical Vector Space concept has two independent types, the vector and scalar types. It is possible to use one vector type with multiple different scalar types, as well as using one scalar type with multiple different vector types. A formal definition of concepts should thus be able to capture the notion of associated types, which are essentially functions between types, and *multi-parameter concepts* constraining independent types.
- The Bidirectional Iterator concept shown in Figure 1 is a *refinement* of the Forward Iterator concept: all of the requirements of Forward Iterator are automatically included in Bidirectional Iterator. In particular, the description of Bidirectional Iterator does not show any associated type requirements, as all of the associated types of Bidirectional Iterator are also in Forward Iterator.
- The valid expressions specify what syntactic constructs have to be valid for objects of modeling types. An alternative mechanism for expressing valid expressions is via function signatures. These mechanisms are not equivalent; in some languages, such as C++, one valid expression can be realized with different function signatures.

The requirements discussed above form the syntactic concept Bidirectional Iterator. A syntactic concept description lists functions and types that are required in each model of a concept; however, how those functions and types must behave is not specified. In addition to requirements on the presence of functions and types, semantic concepts also specify requirements on their run-time behavior. As used in C++, semantic concept descriptions commonly include pre- and post-conditions, invariants, and complexity requirements for particular operations [12, 13]. The "Expression semantics" and "Complexity guarantees" paragraphs in Figure 1 are part of the semantic concept description. Most programming languages with support for concepts can only check syntactic concepts, although efforts for checking for semantic concepts are ongoing [25, 26]. Although the formalization proposed here includes support for semantic concepts, our primary focus in this paper is on syntactic concepts.

Concepts can also serve as criteria for overloading, or algorithm specialization [17]. Concept-based overloading allows more efficient or flexible versions of an algorithm to be used instead of a base version whenever the type arguments model particular concepts which are more than the minimal requirements for the algorithm. For example, the STL *advance* function adjusts the position of an iterator by a specified number of steps.

Description The Bidirectional Iterator concept defines types that support traversal over linear sequences of values, including support for multipass algorithms, and stepping backwards through a sequence (unlike Forward Iterators). A type that is a model of Bidirectional Iterator may be either mutable or immutable, as defined in the Trivial Iterator requirements.

Refinement of Forward Iterator

Associated types The same as for Forward Iterator

Notation Let X be a type that is a model of Bidirectional Iterator; T be the value type of X; i, j be objects of type X; and t be an object of type T

Valid expressions

Name	Expression	Type requirements	Return type
Predecrement	$--i$		$X\&$
Postdecrement	$i--$		X

Expression semantics

Name	Expr.	Precondition	Semantics	Postcondition
Predecrement	$--i$	i is dereferenceable or past-the-end. There exists a dereferenceable iterator j such that $i == ++j$.	i is modified to point to the previous element.	i is dereferenceable. $\&i == \&--i$. If $i == j$, then $--i == --j$. If j is dereferenceable and $i == ++j$, then $--i == j$.
Postdecrement	$i--$	i is dereferenceable or past-the-end. There exists a dereferenceable iterator j such that $i == ++j$.	Equivalent to { *X tmp = i;* $--i$; ***return*** *tmp;*}	

Complexity guarantees Operations on Bidirectional Iterator are guaranteed to be amortized constant time.

Fig. 1. Bidirectional Iterator concept description, adapted from [12].

There are three versions of this algorithm: for models of Input Iterator, the implementation can only move the iterator forward and takes linear time; for models of Bidirectional Iterator, the implementation can move the iterator either forward or backward, but still takes linear time; and for models of Random Access Iterator, the implementation can move the iterator any amount in any direction in constant time. Both the algorithmic complexity and the flexibility of this algorithm are improved for iterator types modeling more refined concepts. One way of implementing concept-based overloading in C++ is the *enable_if* class template, which can be used to selectively control when particular function overloads are enabled and disabled [29]; this, combined with traits classes to represent concept conformance, can be used to implement concept-based type and function specialization [17]. Object-oriented languages such as Java and C# have overloading mechanisms based on the inheritance hierarchy; using interfaces to represent

concepts, a form of concept-based specialization can be achieved in those languages as well. Our formalization does not directly address concept-based specialization, as it does not represent generic algorithms, but generic algorithms can attempt to call the model lookup functions for several different concepts on their type arguments and thus determine their concept conformance. This can then be used to dispatch to different algorithms.

We mention above that concept taxonomies can guide the design of generic libraries. From the library user's perspective however, concepts specify the requirements for calling a particular function. Ideally these requirements are also statically enforced by the compiler when calls to generic functions are attempted, and used in asserting that the definition of a generic function relies only on the documented concept requirements on its arguments. For example, in C++, the *sort()* function in the standard library requires its iterator arguments to model the concept Random Access Iterator. This concept requires operations which allow random access to the elements in the container defined by the iterators.

C++ lacks a representation of concepts, although it is possible for libraries to provide an approximation of syntactic concepts used as constraints using *concept checking classes* [30, 31]. These classes contain pieces of code that exercise all the syntactic requirements of a concept. By placing references to concept checking classes at the top of a generic function, the library developer can cause the compiler to produce better error messages for using a generic function with type that do not meet the concept requirements. A C++ library developer can also test generic libraries using *archetypes*, classes that minimally implement concept requirements. This helps to ascertain that a generic algorithm implementation does not depend on properties of its types, other than what the concept requirements specify. Both concept checking classes and archetypes are, however, not part of the language, and there is no requirement for their use, nor that they correspond to the concept description.

Several other languages have constructs that can represent certain aspects of concepts. *Signatures* in ML and *type classes* in Haskell are an obvious analogue of syntactic concept descriptions. One can also view abstract base classes, or interfaces, in object-oriented languages as concept descriptions. We have explored the practical implications of using these constructs as concept descriptions in [16], revealing that the analogy is not perfect. By providing a detailed formal definition of concepts, we gain insight on the differences and similarities of the various representations.

3 Development of the formalization

Our goal in this section is a definition of concepts motivated by generic programming practice. Our starting point is a previous formal definition by Kapur and Musser [15] of a concept as a set of multi-sorted algebras. In relation to the general definition of concept given in Section 2, Kapur and Musser's definition is more specific in using multi-sorted algebras as the abstractions (models), and [15] defines a small language called Tecton (a formalization of an earlier definition of Tecton by Kapur, Musser and Stepanov [1]) for expressing concept descriptions having both syntactic and semantic requirements. We describe why this definition is insufficient, and generalize it in steps.

First we extend algebras with constant types, and add a function for explicitly capturing the lookup of models of a concept for a set of types. With this change, associated types can be represented. To support circular functional dependencies between concept parameters, an extension to multiple model-finding functions is necessary. Finally, we extend the definition to allow specifications of type constructors, polymorphic functions, and equality of particular types.

3.1 Kapur-Musser definition

Kapur and Musser provide a definition for concepts in the context of the Tecton concept description language [15]. In this definition, a concept is defined as a *multi-sorted signature* and a set of algebras over that signature. A multi-sorted signature consists of a set of *sorts* (names for types) and a set of function names, where each function name has an associated prototype (a list of argument types and a return type, each of which is one of the sorts in the signature). A *multi-sorted algebra* over a given signature defines a type for each sort in the signature, and defines a function for each function name in the signature, matching the required prototype of that function name. The set of algebras that form a concept in Kapur and Musser's definition includes all algebras with the correct signature and which satisfy a set of semantic requirements defined by the concept. Their formalism uses multi-sorted first-order logic predicates to state semantic requirements. No examples were given in [15] of complexity guarantees, but Schwarzweller [32] has recently shown how such requirements can be expressed in Tecton.

Tecton-style concept descriptions have been used to describe several concept taxonomies, including STL container and iterator concepts [33] and many algebraic concepts like Group, Ring, etc. [24]. For example, the Group concept description expresses the properties of a structure with one binary operation, one unary operation (inverse), and one nullary operation (identity). Using an STL-style concept description notation, the set of requirements for a type *T* to model the Group concept are:

Description The Group concept.
Notation Let *T* be a model of Group, and *x* and *y* be objects of type *T*.
Valid Expressions

Expression	Return type
op(x, y)	*T*
identity()	*T*
inverse(x)	*T*

Identities

op(op(a, b), c) = op(a, op(b, c)),
op(identity(), a) = op(a, identity()) = a,
op(inverse(a), a) = op(a, inverse(a)) = identity()

This concept can be represented as a signature and a set of multi-sorted algebras as follows:

$$\textbf{\textit{Signature}} = \langle \textbf{\textit{Sorts}} = \{T\},$$
$$\textbf{\textit{Functions}} = \{\textit{op: } (T * T) \to T,$$
$$\textit{identity: } () \to T,$$
$$\textit{inverse: } T \to T\}\rangle,$$
$$\textbf{\textit{Semantics predicate}} = \ldots$$

Note that the informal notation uses valid expressions, instead of function signatures that are used in the algebraic specification. The difference between the two is discussed in Section 4. Here, ***Semantics predicate*** encompasses any non-syntactic requirements on the models. The identities above are examples of such requirements.

There are certain aspects of this definition of concepts that make it inadequate for our purposes. One major issue is that in practice we are dealing directly with types, and types are not algebras, i.e., they do not necessarily provide mappings from all needed sorts (type names) to types. That is, a type, or even a set of types, is not a complete model for a concept, and we therefore need a way of completing a model from a given type or set of types. In fact, such mechanisms are found in many languages, as described in Section 3.2, and their workings have a significant impact on support for practical generic programming in each language. For a satisfactory formalization of concepts we therefore need to include a way to complete a model of a concept given a set of types.

Another technical issue with the Kapur-Musser definition is that the components of a concept are limited to types and functions. Generics in practical programming languages, however, are more permissive: e.g., Haskell type classes can constrain type constructors [34] or polymorphic functions and C++ allows template template parameters, which are a form of type constructor as well. The Kapur-Musser definition, however, provides a basis that we extend in the following sections to address all of these issues.

3.2 Adding a lookup function and constant types

In this section, we add an explicit lookup function to capture the mechanism for finding models of a concept based on a set of types. This lookup function, for a given concept, is a partial function that accepts a tuple of types (the main types of the concept) and returns an appropriate model of the concept (algebra) for those types, if one exists. Consider the informal concept description of a simple container (not the STL Container) in Figure 2.

Description The Simple Container concept.
Notation Let U be a model of Simple Container, and x an object of type U.
Associated types *value_type* (the type of the contained elements).
Valid Expressions

Expression	Return type
first(x)	*value_type*
rest(x)	*U*
empty(x)	***bool***

Fig. 2. Simple Container concept description

Here, the parameter of this concept is U, but two other kinds of types are used in the function prototypes: a constant type ***bool*** and an associated type *value_type*. In an algebraic definition of this concept, *value_type* is a sort, but the type bound to *value_type*, as well as which model to use, will always be determined solely by the type bound to U. These aspects are reflected in the specification of Simple Container. It is described as a signature, a semantic predicate, and a lookup function (we denote the set of types with $\mathcal{T}$ and the set of models of Simple Container with $\mathcal{A}$):

$$\begin{aligned}
&\textbf{\textit{Signature}} = \langle \textbf{\textit{Sorts}} = \{U,\ \textit{value_type}\},\\
&\qquad\qquad \textbf{\textit{Functions}} = \{\textit{first: } U \to \textit{value_type},\\
&\qquad\qquad\qquad\qquad\ \textit{rest: } U \to U,\\
&\qquad\qquad\qquad\qquad\ \textit{empty: } U \to \textbf{\textit{bool}}\}\rangle,\\
&\textbf{\textit{Semantics predicate}} = \ldots,\\
&\textbf{\textit{Lookup}}\textit{: } \mathcal{T} \rightharpoonup \mathcal{A} = \ldots
\end{aligned}$$

Here, the model lookup function takes only one type (T) as a parameter. The value provided for T thus determines the value of *value_type*, and determines which functions are bound to the names *first*, *rest*, and *empty*.

This definition of concepts can accommodate the varying conformance mechanisms that different languages provide for associating models with concepts. *Nominal conformance* (used, e.g., in Haskell and object-oriented languages) requires an explicit declaration tying a particular model to a particular list of type arguments. *Structural conformance* does not require any explicit declarations of conformance; instead, the compiler automatically finds a model for a given set of types, e.g., using overload resolution, such as in C++.

Different definitions of the ***Lookup*** function reflect different strategies for locating models. For a language using nominal conformance, the lookup function searches a collection of previously defined models in order to find the correct model for a given set of types. Such model definitions contain implementations of the functions and types required by the concept. In a language using structural conformance (C++), the lookup function gathers appropriate functions and type definitions, creates an algebra matching the concept's signature and (possibly) semantic requirements, and returns this as the model to use. Section 4 contains more details of how this lookup function is defined for different conformance mechanisms and lookup behaviors.

3.3 Multiple lookup functions

Associated types can be viewed as functional dependencies between types; in particular, in Haskell, this representation is used for associated types [35]. Functional dependencies, however, are more general than associated types in that they allow circular dependencies. One can find examples of concepts containing circular dependencies, in which each of several concept parameters can be determined from any of the others. For example, a Graph concept can be defined as a composition of node and edge types:

Description The Graph concept defines the requirements for an edge type E and a vertex type V, that together model the concept.

Notation Let E as the edge type and V as the vertex type model Graph, and e be an object of type E, and v an object of type V.

Functional dependencies V determines the type E, E determines the type V.

Associated types None.

Valid expressions

Expression	Return type
out_edges(v)	*list<E>*
source(e)	*V*
target(e)	*V*

Each node has a function to access its outgoing edges, and each edge has functions to access its incident nodes. There is thus no underlying type for the entire graph, only for its nodes and edges. Each node type, however, has a defined edge type, and each edge type has a defined node type. Essentially, the node and edge types are both concept parameters and associated types at the same time. Thus, to capture this kind of concept, a single lookup function is not enough. For Graph, two lookup functions are needed: one to find a model of Graph from its node type, and another to find a model from its edge type:

$$\textbf{\textit{Lookup}} = \{\textit{edge\_type}\colon \mathcal{T} \rightharpoonup \mathcal{A},\ \textit{vertex\_type}\colon \mathcal{T} \rightharpoonup \mathcal{A}\}$$

The model returned from either of these functions defines both the node and edge types, as well as the functions in the concept, and so either type can be determined from the other. The lookup functions in this example are maps from a single type, but in the general case, they are maps from an n-tuple of types. Section 4.2 describes how a set of lookup functions can capture arbitrary sets of functional dependencies.

3.4 Concepts as ML-like signatures

The view of concepts as algebras is not sufficient to capture all varieties of requirements found in concepts as they are used in practice. This section discusses examples from STL and Haskell practice, points out kinds of requirements that fall outside of the algebraic definition, and builds toward a formalization of concepts that can express these kinds of requirements.

Type constructor requirements The concept definition based on algebras describes requirements on types. In practice, concepts sometimes describe requirements on type constructors as well. Type constructors do not operate on elements of types indexed by sorts, but rather are functions between types. The ***Monad*** type class in the Haskell Standard Prelude (Haskell's standard library) is an example of a concept on type constructors. Type constructor requirements are not frequent in C++ concept descriptions; the Allocator concept is one of the few uses of type constructor requirements in the

C++ standard library. The requirement for the *rebind* member template in allocators is equivalent to an *associated type constructor* [7, §20.1.5].

To cover type constructors we must abandon algebras as the models for concepts; at a minimum, we must allow models to contain type constructors. Also, the model lookup functions can accept type constructors as parameters, rather than only types as before.

Polymorphic function requirements An alternative view of type constructors is as polymorphic types, i.e., they are functions from types to types. Polymorphic requirements surface in other ways than as type constructor requirements, such as polymorphic function requirements. Consider the Unique Sorted Associative Container concept from the SGI STL concept taxonomy [12]. Required valid expressions of this concept include the element insertion operator *a.insert(i, j)*, where *a* is of a type that models Unique Sorted Associative Container, and *i* and *j* are of types that model Input Iterator. Here, *insert* must accept any iterator type, as long as the associated *value_type* of the iterator type is convertible to the associated *value_type* of the container type. Polymorphic functions in concept signatures express this kind of requirement. The polymorphism is bounded, where concepts serve as bounds. Haskell has a similar feature, in which a type class can require a type-class-bounded polymorphic function to exist in each model.

As an example of a polymorphic function requirement, here is a portion of the *insert* member requirement from the Unique Sorted Associative Container concept, expressed informally:

Notation Let *A* be the container type modeling Unique Sorted Associative Container and *a* be an object of type *A*. Let *I* be an iterator type modeling Input Iterator such that the value type of *I* is convertible to the value type of *A*, and *i* and *j* be two objects of type *I*.

Valid expressions

Expression	Return type
...	
a.insert(i, j)	***void***
...	

In our formal representation, this requirement is expressed as the following signature (again, only a portion is shown):

$$
\begin{aligned}
&\textbf{\textit{Values}} = \{\ldots,\\
&\quad \textit{insert:}\ \forall I\ \textbf{\textit{where}}\ (I\ \textbf{\textit{models}}\ \textit{InputIterator} \wedge\\
&\qquad \textit{InputIterator\_lookup(I).value\_type}\ \textbf{\textit{convertible-to}}\\
&\qquad\quad \textit{Container\_lookup(A).value\_type}).\\
&\qquad A \rightarrow I \rightarrow I \rightarrow \textbf{\textit{void}},\\
&\quad \ldots\}
\end{aligned}
$$

Thus, polymorphic functions required by concepts can have constrained polymorphism, modeling a feature available in Haskell and C++.

Refinement Our definition of concepts must also support refinement relationships between concepts. One way to represent concept refinement in an algebraic context is to assume that the requirements from the refined concept are included as requirements of the refining concept. This works well for simple concept hierarchies, but is less than ideal in general. A concept can refine another concept multiple times for different types. Consider again the STL Container concept. It requires both the main container type and the associated value type to be models of the Assignable concept. Using an inclusion model for refinement, this concept would need to include two versions of the assignment operator; this would either require overloading among the functions in an algebra or a mechanism to rename functions during the inclusion process. Overloading is not sufficient for our purposes: some of the duplicated names may be those of types rather than functions, and types cannot be overloaded. Renaming is an adequate mechanism, but not fully satisfying: the implementation of concepts would need to have a way to compute or remember how concept functions are renamed in order to use them later.

Instead, we replace algebras as the underlying representations for models with sets of ML-like structures. Structures can be directly nested, and so name conflicts are avoided without either overloading or renaming. Concepts then, rather than consisting of sets of algebras over a signatures, then consist of sets of structures over a signature; these signatures can be nested, and can contain requirements for polymorphic functions. Although ML signatures cannot be nested directly, we remove that restriction for a simpler formalism. The definition and rules for structures and signatures here are based on those in *The Definition of Standard ML* [22]. Because of the change to structures rather than algebras, our representation for concepts can represent concept refinement directly using nested signatures and structures.

Same-type constraints The STL container concepts illustrate another type of requirement that also must be supported by a complete concept definition: requirements that two particular types be the same type. The Container concept, in particular, requires several associated types that serve as iterators on the container. These iterator types each have a defined value type. The container itself also defines its own value type, which must be equal to the value types of the iterators. We refer to such type equality constraints as *same-type constraints*. ML sharing constraints are a form of same-type constraints.

3.5 Full definition

Definition: A *concept* C is defined as a triple $\langle R, P, L\rangle$, where R is a signature and P is a predicate whose domain is all structures which match R. A structure X is defined to *model* C whenever X matches R and $P(X)$ holds. In C, L is a set of functions whose common codomain is the set of models of C. In this definition, R represents the syntactic requirements of the concept, and P represents its semantic requirements. The function L is a *model lookup function* which finds models of a concept from a subset of the concept's members (commonly referred to as the *concept parameters* or *main types*). A signature R is defined as a tuple $\langle T, V, S, E\rangle$, where T represents type requirements, V represents value requirements, S represents nested structure requirements, and E represents same-type requirements. □

Thus the basic structure is the same as for the previous definition that used algebras. Also as before, a model of a concept is a structure matching the concept's signature and satisfying the concept's semantic predicate. The signature itself, however, is more complex. Each signature consists of four parts: type requirements T, value requirements V, nested structure requirements S, and same-type requirements E. Therefore, a signature is defined as a tuple $\langle T, V, S, E \rangle$, where T, V, and S each require a set of particular member names to exist in each model of the concept. For simplicity, it is assumed that all member names are distinct within a single signature. The contents of a signature and how they are matched by a structure are shown in tabular form in Figure 3.

Element	**Symbol**	**In signature**	**In structure**	**Constraint check**
Types	T	Names & kinds	Names & types	Type for a name has defined kind
Values	V	Names & types	Names & values	Value for a name has defined type
Nested structures	S	Names & signatures	Names & structures	Structure for a name matches defined signature
Same-type constraints	E	Pairs of types	—	Structure satisfies all equalities

Fig. 3. Elements of signature definition for concepts

The set T of type requirements in a signature is a set of pairs matching type names to required kinds. The kinds are defined as in a standard type system such as System F_ω (see, for example, [36]); for example, a type has kind $*$ and a function taking one type parameter and returning a type has kind $* \to *$. We write the pair of a name and a kind as "*name: kind*." A similar notation is used for other pairs of names and constraints. A type requirement in a signature then specifies a name for a type that must exist in each model, and a kind to which the type must belong. The type requirements in a concept can represent either main types of the concept or associated types; main and associated types are only distinguished by the model lookup functions. For example, the Simple Container concept in Figure 2 has the type requirements $\{U: *, \mathit{value\_type}: *\}$; each model must define two types named U and *value_type*.

The set V of value requirements in a signature is analogous to the set of type requirements, but V constrains values by types rather than types by kinds. Therefore, an element of V is a pair of a name and a required type, which is usually a function type. A model of a concept must define a value with the specified name and type for each pair in V. The types used as constraints in V can be either constant types from outside the concept (described in Section 3.2), member types defined in T, or elements of nested structures from S. Type function applications in the style of System F_ω can also be used.

In many concepts, value requirements are only used to specify monomorphic functions. For example, the pairs *first:* $U \to$ *value_type*, *rest:* $U \to U$, and *empty:* $U \to$ ***bool*** are monomorphic function requirements for the Simple Container concept shown in Fig-

ure 2. However, we pointed out concepts containing requirements on polymorphic functions (such as the *insert* function in the Unique Sorted Associative Container concept described in Section 3.4). Therefore, to allow requirements on polymorphic values, the types used as constraints can be universally quantified (cf. universally quantified types in System F_ω). Further, the quantified variables in polymorphic values may be bounded by other concepts. As an example of expressing a polymorphic concept-bounded function requirement, consider a function *print* that can output a given type *T* on any type of stream, as long as the stream type models a particular concept. A possible specification as an element of V is:

print: $\forall$*OStream* ***where*** *OStream* ***models*** *OutputStream. (OStream* $*$ *T)* $\rightarrow$ ***void***

The elements of the set S require that any structure matching the signature containing S must contain nested structures with given names. In particular, each element of S is a pair of a name and a signature; matching structures must then contain a nested structure with the given name and which matches the given signature. The signatures in the elements of S are themselves signatures with the same 4-element structure as in our definition of concepts. Thus each concept can contain a nested tree of required signatures, and a structure modeling a concept is checked recursively against this tree of signatures.

Nested structures and signatures are mainly used to model concept refinement. For example, according to the refinement relation shown in Figure 1, a model of Bidirectional Iterator must contain a nested structure modeling Forward Iterator; same-type constraints ensure that the models describe the same iterator type. Section 4.1 describes a detailed example of refinement viewed as nested signatures.

The fourth signature component, E, defines same-type constraints (called sharing constraints in ML). Each element of E consists of a pair of types (of any form) of kind $*$; in any structure matching the signature, both types must evaluate to the same type. The pairs are written as *a = b*, to emphasize that they express equality constraints. For example, the constraint *distance_type* **= *int*** (where *distance_type* is an associated type and ***int*** is a built-in type) requires that every structure for this signature defines its member *distance_type* as ***int***. More usefully, same-type constraints can enforce that models of a refined concept are defined consistently with those of the refining concepts. For example, same-type constraints can prevent the definition of a model of Bidirectional Iterator for ***int*** $*$ which contains a model of Forward Iterator for some other type than ***int*** $*$. Same-type constraints on associated types of a concept and its refined concepts are common, such as the constraint on value type of the STL Container concept and the value types of the associated iterator types of Container.

Our definition of signatures and structures for representing concepts are based on those in Standard ML, and in particular the book *The Definition of Standard ML* [22]. The same-type constraints are exactly the same as sharing constraints in ML; they both require certain pairs of types to be the same type in all structures matching a signature. Also, same-type constraints with constant types (such as *distance_type* **= *int***) can be used to enforce certain transparent type definitions in otherwise opaque structures, as is done with manifest types [37] or translucent sums [38]. Our formalization does not reflect whether a particular language uses fully opaque, fully transparent, or translucent

sums to implement models, but same-type constraints can be used to enforce a level of transparency.

This final definition of concepts and signatures provides full support for all commonly used features of concepts and concept-like constructs in generic programming practice. The combination of an arbitrary function to find models of a concept and a sophisticated definition of what a concept can require from its models produces a flexible representation for concepts. At the same time, this definition allows for language specific idiomatic expression of concepts in languages used for generic programming. By having a single uniform definition for concepts, different language features and different design decisions in support of generic programming can be more meaningfully analyzed.

4 Practical concept representation viewed through our definition

This section describes our representation of concepts and models applied to several programming languages and paradigms: C++, Haskell, ML, and object-oriented languages with generics. The view of these representations and mechanisms as instances of our general formalism demonstrates that the formalism can capture a broad spectrum of generic programming practice. Particularly, we cover the representations of associated types and concept refinement, and show how the model lookup functions in our formalism vary for each language. The granularity of the formalism proves suitable for analyzing differences between languages in features that are relevant to generic programming.

4.1 C++

STL-style concept descriptions in C++ are documentation and are not used by the compiler to enforce interface contracts. Instead, based on the expressions in which generic parameters are used in a generic function, the C++ compiler extracts requirements for the parameters. Thus, the compiler effectively synthesizes syntactic concepts, containing valid expression and associated type requirements, that the parameters must model. The resulting concepts contain the minimal requirements that allow a particular function to compile, and do not necessarily coincide with the documented concept descriptions. For example, the *advance* function in the C++ Standard Library is prototyped as follows:

template *<**class** InputIterator, **class** Distance>*
void *advance(InputIterator& i, Distance n);*

The function moves an iterator *n* steps. The documented requirement for the *InputIterator* type is that it models the Input Iterator concept. However, the implementation of *advance* does not refer to the *value_type* associated type, or dereference the iterator *i*, and thus the requirements extracted by the compiler are looser than the documented requirement. In the following discussion of the C++ manifestation of the model lookup functions and the mechanisms for generating models for concepts, we view the documented concepts as refinements of the compiler-generated concepts. In practice, the compiler need not satisfy the requirements that are outside the compiler-generated concepts.

Lookup functions The C++ model lookup function (almost all concepts only use one) is more complex than that for many other languages. STL-style concepts do not have an explicit relationship between models and concepts; instead, the functions and types that implement a particular model are found when each generic algorithm is instantiated. For example, neither the types and functions that form a model of the Standard Library's Bidirectional Iterator concept (cf. Figure 1), nor any other construct in a C++ program, relates the model to the concept.

In C++, rather than finding models, the lookup function must create them. Given a particular concept and set of type arguments for it, the model creation process searches for a match for each function, associated type, or refined concept in the given concept; if any of these is not found, the argument types do not model the concept. For functions, standard C++ argument-dependent lookup and overload resolution are used to find the correct definition of the function. This may require implicit conversions on the arguments of the function, and thus can effectively compose a new function, not written by the programmer, to satisfy a particular function requirement. Baumgartner and Russo suggested that C++ be extended with *signatures*; their implementation for the GNU C++ compiler used a similar process for creating the equivalents of models [39].

The process of model creation also resolves types to satisfy the associated type requirements. Associated types in C++ are expressed with *traits classes* [28]. Traits classes are templates that act as type functions mapping from the main types of a concept to its associated types. For example, the *iterator_traits* template gives access to the value type of an iterator *It* with the expression *iterator_traits*$<$*It*$>$*::value_type*, where *value_type* is a member type definition in *iterator_traits*. In our formalism, associated type requirements are named pairs in the T component of the concept signature, like the main types of the concept. Associated types, however, are not parameters to the concept's model lookup function, but rather obtained from the models produced by the lookup function. This part of the model lookup function in C++ involves finding the correct specializations of the traits class templates for the set of main types of the concept, and inserting the type members of that traits class into the model being created.

Refinement Concept refinements are represented in our formalism as nested signatures inside a concept's signature. The semantic requirements from the refined concept are included in the semantic requirements of the refining concept. Figure 4 shows how the refinement relation between the STL concepts Bidirectional Iterator and Forward Iterator (cf. Figure 1) is captured in our formalism. For brevity, the name *ForwardIterator* is used as a placeholder for the nested signature describing the syntactic requirements of Forward Iterator.

Note that even though the STL-style concept descriptions exhibit refinement structure, their compiler-generated counterparts do not have any such structure. Thus the model creation process makes no distinction between requirements obtained via refinement and requirements stated explicitly.

Function requirements as valid expressions STL-style concept descriptions express requirements on functions as *valid expressions*, not by requiring the existence of functions with particular prototypes. A valid expression denotes a C++ expression that must

Signature = ⟨***Types*** = {*iterator_type:* ∗},
Values = {},
Structures = {*forward_iterator_iter: ForwardIterator*},
Same-type constraints = {*forward_iterator_iter.iterator_type = iterator_type*}⟩,
Semantics predicate = . . .
Lookup: $\mathcal{T} \rightharpoonup \mathcal{A}$ = . . .

Fig. 4. A representation of the C++ concept Bidirectional Iterator in our formalism. The concept has one main type: *iterator_type*. Bidirectional Iterator refines Forward Iterator: the name *forward_iterator_iter* refers to this refined concept, and allows accessing its components, including the associated type *iterator_type*.

compile for any model of the concept, under the assumption that variables in the expression have certain types. The semantics of valid expressions are also often given in concept descriptions.

Valid expressions are a natural notation for constraints in C++; there are often several ways a given expression can be implemented and thus one valid expression can cover a large set of function prototypes. For example, an operator may be defined either as a member or non-member function. A call to any function may involve built-in and/or user-defined implicit conversions, in which case a single valid expression corresponds to a composition of several function prototypes. As an example, the expression $-a$, where a is of type A, can be implemented with several function prototypes, including:

- *A* ***operator***−*(A);*
- *A* ***operator***−*(****const*** *A&);*
- *A A::****operator***−*();*
- ***template*** <***class*** *T*> *T* ***operator***−*(T);*

The valid expression approach seemingly conflicts with the function requirements in our concept formalism. However, valid expressions can be modeled in our approach by converting them into function signatures by creating extra associated types. The approach is the same as that used for generating concept archetypes in the Caramel system [14, 40]. In this approach, we view valid expressions as parse trees and create extra, hidden associated types for the intermediate nodes of the parse trees.

For example, consider the valid expression $a * b + c$ whose return type is required to be ***double***. Clearly this valid expression corresponds to a combination of several different function signatures, at least the $+$ and $*$ operators, but potentially more. Thus, this requirement is expressed as a set of function signature requirements. In this process, a new associated type, say *a_times_b*, is created to represent the result type of $a * b$, thus allowing the valid expressions to be divided into two simpler expressions:

- $a * b$, required to return *a_times_b*,
- $x + c$ with x of type *a_times_b*, required to return ***double***.

These valid expressions can then be encoded as function signatures in concept representations. This process may need to introduce equality constraints between some of

the generated types and other types in the concept. For example, the same subexpression can occur in several required valid expressions. In such a situation, the requirements on their return types must be combined. If we add a valid expression requirement "$a * b$ returns a type convertible to ***double***" to the example above, a new associated type *a_times_b_2* is generated. Now, *a_times_b* and *a_times_b_2* must be the same type, because they originate from the same valid expression. In Caramel, a form of common subexpression elimination is used to find associated types that are the same type.

Summary Mapping C++ concepts into our formalism lets us make several observations. First, from the generic programming point of view the extremely complex function overload resolution mechanism of C++ serves a simple purpose: it defines the lookup functions that find models for concepts. Clearly, the specification of the lookup functions are not as an algorithm dedicated for this purpose, but rather a by-product of other language mechanisms — and thus complicated. Second, the lookup functions are not merely finding definitions for required functions, but also generating new function definitions if necessary. Third, we can note that in practice the sets of main types and associated types in C++ concepts seem to be distinct. This means that one lookup function suffices for each concept. Fourth, valid expressions are representable using function signatures, and thus seem to be equally expressive to function signatures in the C++ context.

4.2 Type classes

A *type class* [41] describes constraints on one or more types. In particular, function prototypes in a type class require correctly named and typed functions to be defined for the type parameters of a type class. Type classes can also provide default implementations for the functions that they require; these implementations are in terms of other required functions. Models of concepts in Haskell are provided explicitly, using *instance declarations*. An instance declaration of a particular type class for some types contains the definitions of the functions that the type class requires for those types. A tuple of types is said to be an *instance* of a type class when an instance declaration has been defined for those types; the function definitions in an instance declaration are also sometimes referred to as the instance for those types. Note that multi-parameter type classes and functional dependencies (used for associated types) are extensions to the Haskell 98 standard [42], but are supported in most mainstream implementations of Haskell.

Haskell's syntactic type classes, and type classes with semantic properties like those in Isabelle [43], map straightforwardly to our formalism: a type class definition corresponds to the signature component, and instance declarations define the lookup functions. In the basic case of a type class without functional dependencies among its type parameters, the type parameters are all parameters to the concept's lookup function and all of the functions contained in the type class are value requirements of the signature. Superclasses (refined concepts) of a type class are modeled as nested signatures. For example, the *VectorSpace* type class, defined in Haskell as:

class *(Field s, AdditiveAbelianGroup v)* $\Rightarrow$ *VectorSpace v s* ***where***
  vs_mult :: $v \rightarrow s \rightarrow v$

$sv\_mult :: s \to v \to v$

can be represented in our formalism as follows (the signatures of the refined concepts are not expanded out, *Field* and *AdditiveAbelianGroup* are placeholders for those):

$$\begin{aligned}
&\textbf{\textit{Signature}} = \langle \textbf{\textit{Types}} = \{v: *, \\
&\qquad\qquad\qquad s: *\}, \\
&\qquad\qquad \textbf{\textit{Values}} = \{vs_mult: v \to s \to v, \\
&\qquad\qquad\qquad\quad sv_mult: s \to v \to v\}, \\
&\qquad\qquad \textbf{\textit{Structures}} = \{field_s: Field, \\
&\qquad\qquad\qquad\qquad additive_abelian_group_v: AdditiveAbelianGroup\}, \\
&\qquad\qquad \textbf{\textit{Same-type constraints}} = \{field_s.t = s, \\
&\qquad\qquad\qquad\qquad\qquad additive_abelian_group_v.t = v\}\rangle, \\
&\textbf{\textit{Semantics predicate}} = \ldots \\
&\textbf{\textit{Lookup}}: (\mathcal{T} * \mathcal{T}) \rightharpoonup A = \ldots
\end{aligned}$$

The semantic predicate for this type class always returns $true$, as Haskell type classes do not include semantic constraints. If the semantic constraints in the mathematical definition of a Vector Space were desired, the semantic predicate could require that any structures used as instances of the type class satisfied the algebraic properties of a vector space. The single model lookup function accepts a vector and scalar type and returns the appropriate model, if there is one.

Lookup functions In Haskell, nominal conformance is used to match types to models: subclasses and instance declarations create a database representing the concept taxonomy and the models relations. Therefore, the model lookup functions in Haskell query this database to find models. Lookup is thus simpler than in C++: models are defined, there is no need to generate functions to create models. However, definitions of type class instances can be polymorphic. An example of a such an instance declaration, in pseudo-code, is "for all types *t* where *t* is an instance of *Eq*, *Array t* is an instance of *Eq*." This complicates the model lookup process. In our example, the instance for arrays relies on the instance of *Eq* for *t*, and so instances are effectively combined by the lookup function.

Haskell implementations (as extensions to the Haskell 98 standard [42]) allow functional dependencies between type parameters of a type class [35]. Functional dependencies require that given certain values of some of the type parameters, other parameters are completely determined. Functional dependencies can represent associated types but also allow the expression of more complex relationships between types. In particular, functional dependencies can be circular, as demonstrated by the *Graph* concept in Section 3.3. For representing such functional dependencies in our formalism, multiple lookup functions are used. More precisely, one lookup function exists for each maximal set of type class parameters that uniquely determine all other parameters. Each of these lookup functions can obtain the full set of type class parameters, and use that set to find an instance of the type class. The original functional dependency paper ([35]) shows an algorithm using "improving substitutions" based on defined instances of a type class as the implementation for functional dependencies; the core idea of this is to use the

concept parameters which are known to find the rest based on the functional dependencies. That paper also suggests a further extension to give names to the function implied by the functional dependency, which is similar to our model lookup function but only maps main types to associated types, not to whole models.

One interesting extension of type classes that has been proposed is that of *named instances* [44]. This mechanism allows a particular instance to be given a name, and then a particular call to a generic function can use the named instance rather than the default instance of a type class for a particular set of types. For example, a Semigroup concept requires an associative binary operation. A type, such as integer, can model this concept by either an addition or multiplication operation. Using the named instance extension to type classes, a generic algorithm could by default use the additive definition of the instance, but could be explicitly told in a function call to use the multiplicative definition instead. This can be expressed in our concept formalism by allowing the programmer to override the model lookup function in a call to a generic function by explicitly providing the model for the required concept and parameters directly.

Refinement The type class equivalent of a refining concept is called a *subclass* of the refined type class; the refined type class is then referred to as a *superclass* of the refining type class. In order for a type to be an instance of a subclass, it must first be an instance of all of that class's superclasses; this requirement is checked when each instance is defined. Thus, an instance of a particular subclass for a given set of arguments can only exist when instances of the that class's superclasses exist for those arguments. This mechanism is in direct correspondence with concept refinement in our formalism: superclasses are nested signatures. We can note that the refinement structure of concepts is explicitly retained in their type class representations, unlike in C++ compiler-generated concepts. This also reflects the standard implementation of type classes using dictionary passing [41].

Summary The nominal conformance used by Haskell leads to relatively simple model lookup functions, at least compared to C++. Polymorphic model definitions can add some complexity, but the lookup process still consists of finding or combining existing models rather than creating completely new ones.

To capture circular functional dependencies, multiple lookup functions are necessary. While expressing such dependencies in C++ is also possible, it requires fairly contrived mechanism to be used. In concepts described as type classes, circular functional dependencies are expressed more naturally.

4.3 ML signatures

A *structure* in ML defines a collection of named types, values (usually functions), and nested structures. A *signature* specifies a collection of named types, values with their types specified, and nested signatures. These are used to constrain the interfaces to *functors*, which are functions from structures to structures. Thus, a given functor will accept one or more structures as input, each constrained to match a particular signature, and will produce a structure as output. A structure is said to match a signature when all

types in the signature are present in the structure, all values in the signature are defined in the structure (with the types defined in the signature), and all nested signatures in the signature are matched by nested structures in the structure.

ML signatures and structures provide a very direct representation of concepts and their models; in fact, the signature component of our concept formalization borrows its structure from ML signatures [22]. In particular, polymorphic members, nested signatures, and sharing constraints have direct counterparts, allowing full-fledged ML signatures to be captured in our formalization.

The interesting observation about ML constructs mapped into our formalism is the lack of lookup functions. An ML functor expects the entire input structure as an explicit argument — there are no mechanisms for finding a structure based on a subset of the elements of the structure, say, given a subset of the required types. The lack of lookup functions means that generic algorithms must be explicitly instantiated. This hampers the usefulness, in regard to generic programming, of the otherwise powerful genericity mechanisms of ML, as shown in [16].

4.4 Object-oriented interfaces

In modern strongly-typed object-oriented languages, such as Java and C#, types are of two varieties: *classes* and *interfaces*. An interface describes requirements on a set of methods (functions on an object) without giving their implementations. These requirements are given in the form of function signatures. A class is a concrete or partially abstract data structure, and may include all or part of an implementation. Using Java terminology, classes are said to *implement* interfaces, and a subclass is said to *extend* its superclasses. An interface can also extend other interfaces, which includes all of the requirements of the superinterfaces into the subinterface. More recently, some object-oriented languages have been extended with generics, which allow classes, interfaces, and methods to exhibit constrained parametric polymorphism. The constraints are usually based on the subclass relationship. Examples of languages with generics include Eiffel [45], Java generics [46], and Generic C# [47, 48].

Object-oriented languages with generics can approximate concepts and generic programming to an extent, as shown in [16] and [49]. Interfaces are used to represent concepts; interface extension represents concept refinement. Associated types can be represented as extra type parameters to generic interfaces. Types are declared to model concepts using the "implements" relation. These interfaces fit into our formalization of concepts, but they have some restrictions. For example, each interface can only constrain one main type, and that main type must be the first (***this***) parameter to every function in the interface. Thus, all lookup functions must only have one type parameter, and each concept can only have one lookup function. In practice, these restrictions lead to serious problems in practical generic programming [49].

Lookup functions The lookup function for object-oriented languages finds an implementation of a given interface for a given class; it may also fill in some of the type parameters of a generic interface. Object-oriented languages do not have fully general functional dependencies, and so there is always only one model lookup function. The

process of finding an implementation of an interface is based on the class hierarchy: a type is defined to model a concept if it is a subclass of the concept's interface. In most languages, this subclass relationship is fixed when the class is defined; this leads to a loss of flexibility. The implementation of model lookup then searches the superclasses of the given class to find whether any instance of the concept's interface is a superclass. Once an instance is found, its type arguments become the associated types in the model. The function definitions for a model are usually found through a structure such as a virtual function table. Thus, the lookup function searches the class hierarchy, and creates a table of already-existing functions and types to form concept models.

Associated types In our previous work, associated types were shown to be problematic to represent in most object-oriented languages [16]. In that work, they were represented using extra type parameters to generic interfaces; this provides a correct representation, but is often inconvenient to use and leads to a loss of modularity. Our formalization does not always express these sorts of problems with associated types: whether types are given implicitly or explicitly to generic algorithms is not represented, and extra type parameters to interfaces are parts of models in our framework. However, some languages, such as Eiffel, require all types, including associated types, to be passed explicitly to generic algorithms; this corresponds to a lookup function which requires all of the concept's types, not just the main types, as arguments. Other languages that are able to determine associated types automatically are viewed in our formalization as having support, as issues such as redundant constraints are not represented in the formalization. Thus, although support for associated types varies widely between languages, our formalism does not represent that aspect of generic programming directly.

Different object-oriented languages Various object-oriented languages differ in what features they support. The discussion here refers mainly to mainstream object-oriented languages with generics, such as Eiffel, Java generics, and Generic C#. Other languages, such as Cecil [50], provide different levels of support. For example, Cecil provides *multimethods*, which allow methods to be dispatched based on the types of all of their arguments, rather than just the first. This removes the previously mentioned restriction that the main type of a concept must be the first parameter type of every required function. Other languages support *virtual types* [51, 52], which provide an equivalent to associated types, although virtual types belong to objects rather than types. Our previous work [49] shows the effects of these features, and others, on generic programming support.

Summary Similarly to Haskell, object-oriented languages use nominal conformance, and so their model lookup functions are fairly simple. The restrictions in mainstream object-oriented languages on what kinds of function and type constraints can be expressed do not significantly affect our formalism — they just restrict which of the possible constraint types on the formalism can be used. Concepts in object-oriented languages always have a single lookup function, and it operates on a single type parameter; the ways of defining the modeling relation are also fairly restricted. Our formalism

allows many of the limitations of object-oriented languages for generic programming to be shown more directly.

5 Related work

Some aspects of our formal definition of concepts — including semantic requirements — are found in various other algebraic specification languages besides Tecton. The Spectrum specification language [53], for example, uses a version of type classes and allows arbitrary first-order logic formulas as semantic constraints; its underlying formalism is based on multi-sorted algebras, however. As another example, CASL (Common Algebraic Specification Language) [54], is a synthesis of ideas from previous algebraic specification languages such as OBJ [55] and Larch [56, 57] and other research on algebraic specifications. It is a relatively large language in comparison to Tecton; for example, CASL supplies several mechanisms for hiding part of a signature, while Tecton has none (except for marking some functions as private). This makes it more likely to be able to express requirements on models of concepts for use in real programming languages, but the size and complexity of CASL as a whole may be a barrier to acceptance into the programming community. It will therefore be advantageous to extract from CASL and perhaps other specification languages a subset sufficient for expressing concepts according to our formal definition, while omitting features unnecessary for that purpose.

Other algebraic specification languages such as Pluss [58] and GSBL [59, 60] have more in common with Tecton than CASL, including small size and similarities in abstract syntax (the concrete syntax is somewhat different), and, in the case of GSBL, provision for generic specifications without explicit parameters, where specific instances can be obtained by replacing some sub-specification by another one. With GSBL, Pluss, and CASL, the formal semantic definition is based on category-theoretic notions; we would prefer instead to retain the relative simplicity of a formal semantics in terms of multi-sorted algebras similar to Tecton's, but extended sufficiently to encompass the additional features identified in our formal definition.

As our formal definition includes several features of ML signatures, a natural question is whether Extended ML [61], an algebraic specification language designed explicitly for use in development of SML programs, offers a complete solution. However, the stated goal of Extended ML is to be a specification language for use only with SML and not with other programming languages. Thus it lacks the generality seen in other many other formal specification languages, especially in that it does not provide parameterized signatures.

Some computer algebra systems also use constraint mechanisms similar to concepts, such as Scratchpad and its derivative Axiom [62–64]. These systems use a notion of "categories," which are similar to concepts in Tecton; they use the multi-sorted algebra formalism but use nominal conformance to represent concepts with the same operations but different semantics. Scratchpad and Axiom include category-based algorithm specialization, which is not found in normal type class systems such as that in Haskell. Santas extends an Axiom-like type system with ML-like structures, functors, and signatures, which allow support for associated types [65]. This system also adds

type classes to Axiom. The functors defined there, like ML functors, must be explicitly applied; type classes do not have this problem, but are limited to a single type parameter.

As previously mentioned in Section 3, our definition of concepts is based on ML structures and signatures [22], and same-type constraints express equivalents to sharing constraints, manifest types [37], and translucent sums [38]. Two restrictions exist in ML that do not in our representation: signatures cannot be directly nested, and constrained polymorphism is not allowed in structure members. In our representation, one signature contains another as a nested structure requirement by including a nested signature; in ML, a name must be given to the nested signature externally, and then it is referenced in the outer signature. Also, ML functions are either monomorphic or use unconstrained parametric polymorphism; our system allows concepts as bounds for both polymorphic types and polymorphic functions.

Several other formalizations use various forms of dispatching in the context of modules or module-like constructs. Harper and Morrisett show an alternative way of implementing type classes using an equivalent to a *typecase* mechanism and a language with types explicitly represented as function parameters, which allows run-time dispatching based on the argument types of a polymorphic function [66]. This allows type functions to be defined to compute different result types based on the structures of their argument types. *Qualified types* are used to formalize type classes, subtyping, and several other varieties of constrained polymorphism [67]. In that model, type class dictionaries (implementations of the functions required by the type class) are implemented using arbitrary *evidence* structures, and the method for finding an appropriate evidence structure for a given set of types is also by an arbitrary relation with constraints that make it equivalent to the partial function approach used in our formalization; however, the qualified types work does not allow multiple lookup relations. Jones does not define a module-like structure for the models used for type classes, and he also does not relate his model to other languages such as C++ or Standard ML.

6 Conclusions

Although concepts provide the notions for rigorously defining interfaces of generic algorithms, concepts themselves have not been well-defined. A semi-formal notation is commonly in use in generic libraries in C++, but it has not been clear which parts of that formalism are essential to concepts, which are to accommodate the C++ language, and which are idiosyncrasies of the STL and the Standard Library. Similarly, it has not been clear how ideas about concepts would most effectively map to other programming languages that support generic programming. The definition of concepts presented here should clarify these issue by combining features from various implementations of generic programming into a single, unified definition of concepts. Generic programming can now be considered a language-independent paradigm, rather than having varied, but similar, realizations in different languages.

Our definition is based on a previous formalization of concepts by Kapur and Musser [15], but extended with one or more lookup functions to find models of a concept based on their parameters. The parameters of a concept and the contents of models have also been generalized to fully support generic programming practice. The param-

eters of a concept have been extended to allow concepts which constrain type functions as well as types, and the bodies of models have been extended to be similar to ML structures [22]. These structures allow more direct support for concept refinement, as well as polymorphic members in models. We have also added a representation for same-type (or sharing) constraints, to support the equivalent functionality from C++ concepts or ML signatures. To show the generality of our formalism, we have showed how it can express C++ concepts, Haskell type classes, ML signatures, and object-oriented interfaces used for constrained polymorphism.

This paper defines a set of requirements needed for a good concept representation (motivated by practical concept descriptions), but it still remains to define a type system and language with full concept support. Besides providing a basis for comparing features of different languages for generic programming, our formalism also provides insight for evolving existing languages to better support generic programming. For example, one might extend ML to have implicit instantiation, or extend C++ to have direct support for specifying concepts. Alternatively, our formalization also admits the prospect of an entirely new language defined specifically for generic programming.

7 Acknowledgments

This work was supported by NSF grant EIA-0131354 and a grant from the Lilly Endowment. The first author was supported by a Department of Energy High Performance Computer Science Fellowship. We are grateful to Jeremy Siek for numerous detailed discussions on generic programming and for many helpful comments on this work.

References

1. Kapur, D., Musser, D.R., Stepanov, A.A.: Tecton: A language for manipulating generic objects. In Staunstrup, J., ed.: Proceedings of a Workshop on Program Specification. Volume 134 of LNCS., Aarhus, Denmark, Springer (1981) 402–414
2. Wille, R.: Restructuring lattice theory: An approach based on hierarchies of concepts. In Rival, I., ed.: Ordered Sets. Reidel, Dordrecht-Boston (1982) 445–470
3. Wille, R.: Concept lattices and conceptual knowledge systems. Computers and Mathematics with Applications **23** (1992) 493–522
4. Kapur, D., Musser, D.R., Stepanov, A.: Operators and algebraic structures. In: Proc. of the Conference on Functional Programming Languages and Computer Architecture, Portsmouth, New Hampshire, ACM (1981)
5. Stepanov, A.: The Standard Template Library — how do you build an algorithm that is both generic and efficient? Byte Magazine **20** (1995)
6. Stepanov, A., Lee, M.: The Standard Template Library. Technical Report HPL-94-34(R.1), Hewlett-Packard Laboratories (1994) `http://www.hpl.hp.com/techreports`.
7. International Standardization Organization (ISO): ANSI/ISO Standard 14882, Programming Language C++, 1 rue de Varembé, Case postale 56, CH-1211 Genève 20, Switzerland (1998)
8. Siek, J., Lee, L.Q., Lumsdaine, A.: The Boost Graph Library: User Guide and Reference Manual. Addison-Wesley (2002)
9. Siek, J., Lumsdaine, A.: A modern framework for portable high performance numerical linear algebra. In: Modern Software Tools for Scientific Computing. Birkhäuser (1999)

10. Abrahams, D., Siek, J., Witt, T.: The Boost.Iterator Library. Boost. (2004) http://www.boost.org/libs/iterator/doc/index.html.
11. Walter, J., Koch, M.: The Boost uBLAS Library. Boost. (2002) www.boost.org/libs/numeric.
12. Silicon Graphics, Inc.: SGI Implementation of the Standard Template Library. (2004) http://www.sgi.com/tech/stl/.
13. Austern, M.H.: Generic Programming and the STL. Professional computing series. Addison-Wesley (1999)
14. Willcock, J., Siek, J., Lumsdaine, A.: Caramel: A concept representation system for generic programming. In: Second Workshop on C++ Template Programming. (2001)
15. Kapur, D., Musser, D.: 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 12180 (1992)
16. Garcia, R., Järvi, J., Lumsdaine, A., Siek, J., Willcock, J.: A comparative study of language support for generic programming. In: Proceedings of the 18th ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications, ACM Press (2003) 115–134
17. Järvi, J., Willcock, J., Lumsdaine, A.: Concept-controlled polymorphism. In Pfennig, F., Smaragdakis, Y., eds.: Generative Programming and Component Engineering. Volume 2830 of LNCS., Springer Verlag (2003) 228–244
18. Gibbons, J.: Patterns in datatype-generic programming. In: Workshop on Declarative Programming in the Context of Object-Oriented Languages, Uppsala. (2003)
19. Stroustrup, B., Dos Reis, G.: Concepts — design choices for template argument checking. Technical Report N1522=03-0105, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++ (2003) http://anubis.dkuug.dk/jtc1/sc22/wg21.
20. Stroustrup, B.: Concepts — a more abstract complement to type checking. Technical Report N1510=03-0093, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++ (2003) http://anubis.dkuug.dk/jtc1/sc22/wg21.
21. Stroustrup, B., Dos Reis, G.: Concepts — syntax and composition. Technical Report N1536=03-0119, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++ (2003) http://anubis.dkuug.dk/jtc1/sc22/wg21.
22. Milner, R., Tofte, M., Harper, R.: The Definition of Standard ML. MIT Press (1990)
23. Kapur, D., Musser, D.R., Nie, X.: An overview of the Tecton proof system. Theoretical Computer Science **133** (1994) 307–339
24. Musser, D.R., Schupp, S., Schwarzweller, C., Loos, R.: Tecton Concept Library. Technical Report WSI-99-2, Fakultät für Informatik, Universität Tübingen (1999)
25. Gregor, D., Schupp, S.: Making the usage of STL safe. In Gibbons, J., Jeuring, J., eds.: Proceedings of the IFIP TC2 Working Conference on Generic Programming, Boston, Kluwer (2003)
26. Gregor, D.: Static Analysis of Generic Component Composition. PhD thesis, Rensselaer Polytechnic Institute (forthcoming)
27. Musser, D.R.: Algorithm concepts. http://www.cs.rpi.edu/~musser/gp/algorithm-concepts (2003)
28. Myers, N.C.: Traits: a new and useful template technique. C++ Report (1995)
29. Järvi, J., Willcock, J., Hinnant, H., Lumsdaine, A.: Function overloading based on arbitrary properties of types. C/C++ Users Journal **21** (2003) 25–32
30. Siek, J., Lumsdaine, A.: Concept checking: Binding parametric polymorphism in C++. In: First Workshop on C++ Template Programming. (2000)
31. McNamara, B., Smaragdakis, Y.: Static interfaces in C++. In: First Workshop on C++ Template Programming. (2000)

32. Schwarzweller, C.: Towards formal support for generic programming. http://www.math.univ.gda.pl/~schwarzw (2003) Habilitation thesis, Wilhelm-Schickard-Institute for Computer Science, University of Tübingen.
33. Musser, D.R.: Tecton description of STL container and iterator concepts. http://www.cs.rpi.edu/~musser/gp/tecton/container.ps (1998)
34. Jones, M.P.: A system of constructor classes: overloading and implicit higher-order polymorphism. In: FPCA '93: Conference on Functional Programming and Computer Architecture, Copenhagen, Denmark, New York, N.Y., ACM Press (1993) 52–61
35. Jones, M.P.: Type classes with functional dependencies. In: European Symposium on Programming. Number 1782 in LNCS, Springer-Verlag (2000) 230–244
36. Pierce, B.C.: Types and Programming Languages. MIT Press (2002)
37. Leroy, X.: Manifest types, modules, and separate compilation. In: Proceedings of the 21st Annual ACM Symposium on Principles of Programming Languages. (1994) 109–122
38. Lillibridge, M.: Translucent Sums: A Foundation for Higher-Order Module Systems. PhD thesis, Pittsburgh, PA (1997)
39. Baumgartner, G., Russo, V.F.: Signatures: A language extension for improving type abstraction and subtype polymorphism in C++. Software–Practice and Experience **25** (1995) 863–889
40. Willcock, J.: Concept representation for generic programming. Master's thesis, University of Notre Dame du Lac (2002)
41. Wadler, P., Blott, S.: How to make ad-hoc polymorphism less ad-hoc. In: ACM Symposium on Principles of Programming Languages, ACM (1989) 60–76
42. Jones, S.P., Hughes, J., et al.: Haskell 98: A Non-strict, Purely Functional Language. (1999) http://www.haskell.org/onlinereport/.
43. Wenzel, M.: Using axiomatic type classes in Isabelle (manual) (1995) www.cl.cam.ac.uk/Research/HVG/Isabelle/docs.html.
44. Kahl, W., Scheffczyk, J.: Named instances for Haskell type classes. In Hinze, R., ed.: Proc. Haskell Workshop 2001. Volume 59 of ENTCS. (2001) See also: http://ist.unibw-muenchen.de/Haskell/NamedInstances/.
45. Meyer, B.: Eiffel: the Language. First edn. Prentice Hall, New York, NY (1992)
46. Bracha, G., Cohen, N., Kemper, C., Marx, S., et al.: JSR 14: Add Generic Types to the Java Programming Language. (2001) http://www.jcp.org/en/jsr/detail?id=014.
47. Kennedy, A., Syme, D.: Design and implementation of generics for the .NET Common Language Runtime. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Snowbird, Utah (2001) 1–12
48. Microsoft Corporation: Generics in C# (2002) Part of the Gyro distribution of generics for .NET available at http://research.microsoft.com/projects/clrgen/.
49. Järvi, J., Lumsdaine, A., Siek, J., Willcock, J.: An analysis of constrained polymorphism for generic programming. In Davis, K., Striegnitz, J., eds.: Multiparadigm Programming in Object-Oriented Languages Workshop (MPOOL) at OOPSLA, Anaheim, CA (2003)
50. Chambers, C., the Cecil Group: The Cecil Language: Specification and Rationale, Version 3.1. University of Washington, Computer Science and Engineering. (2002) www.cs.washington.edu/research/projects/cecil/.
51. Madsen, O.L., Moller-Pedersen, B.: Virtual classes: a powerful mechanism in object-oriented programming. In: OOPSLA, ACM Press (1989) 397–406
52. Thorup, K.K.: Genericity in Java with virtual types. In: ECOOP. Volume 1241 of Lecture Notes in Computer Science. (1997) 444–471
53. Grosu, R., Nazareth, D.: Towards a new way of parameterization. In: Third Maghrebian Conference on Software Engineering and Artificial Intelligence. (1994) 383–392

54. CoFI Language Design Task Group: CASL—the CoFI algebraic specification language—summary (2001) `http://www.brics.dk/Projects/CoFI/Documents/CASL/Summary/`.
55. Goguen, J.A., Winker, T., Meseguer, J., Futatsugi, K., Jouannaud, J.P.: Introducing OBJ. In: Applications of Algebraic Specification using OBJ. Cambridge University Press (1992)
56. Garland, S.J., Guttag, J.V., Horning, J.J.: Debugging Larch Shared Language specifications. IEEE Trans. Software Engineering **16** (1990) 1044–1057
57. Guttag, J.V., Horning, J.J.: Report on the Larch Shared Language. Science of Computer Programming **6** (1986) 103–134
58. Bidoit, M.: The stratified loose approach, a generalization of initial and loose semantics. Number 332 in LNCS. In: Recent Trends in Data Type Specification. Springer-Verlag (1988) 1–22
59. Clerici, S., Orejas, F.: GSBL: an algebraic specification language based on inheritance. Number 322 in LNCS. In: Proc. 1988 European Conference on Object Oriented Programming. Springer-Verlag (1988) 78–92
60. Clerici, S., Orejas, F.: The specification language GSBL: an algebraic specification language based on inheritance. Number 534 in LNCS. In: Recent Trends in Data Type Specification. Springer-Verlag (1991) 31–51
61. Kahrs, S., Sannella, D., Tarlecki, A.: The definition of Extended ML: a gentle introduction. Theoretical Computer Science **173** (1997) 445–484
62. Davenport, J.H., Trager, B.M.: Scratchpad's view of algebra I: Basic commutative algebra. In: Design and Implementation of Symbolic Computation Systems. (1990) 40–54
63. Davenport, J.H., Gianni, P., Trager, B.M.: Scratchpad's view of algebra II: A categorical view of factorization. Technical Report TR4/92 (ATR/2) (NP2491), Downer's Grove, IL, USA and Oxford, UK (1992)
64. Davenport, J.H.: The AXIOM system. Technical Report TR5/92 (ATR/3) (NP2492), Downer's Grove, IL, USA and Oxford, UK (1992)
65. Santas, P.S.: A type system for computer algebra. In: Design and Implementation of Symbolic Computation Systems. (1993) 177–191
66. Harper, R., Morrisett, G.: Compiling polymorphism using intensional type analysis. In: Principles of Programming Languages, San Francisco, California (1995) 130–141
67. Jones, M.P.: Qualified Types: Theory and Practice. Distinguished Dissertations in Computer Science. Cambridge University Press (1994)