THESIS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

Generic Programming with Concepts

Marcin Zalewski

CHALMERS | GÖTEBORG UNIVERSITY

Department of Computer Science and Engineering
Chalmers University of Technology and Göteborg University
SE-412 96 Göteborg
Sweden

Göteborg, 2008

Generic Programming with Concepts
Marcin Zalewski
ISBN 978-91-7385-176-3

Doktorsavhandlingar vid Chalmers tekniska högskola, Ny serie Nr 2857.
ISSN 0346-718X
Department of Computer Science and Engineering
Research Group: Software Methodologies and Systems

Technical Report no. D46
Department of Computer Science and Engineering
Chalmers University of Technology and Göteborg University
SE-412 96 Göteborg, Sweden
Telephone +46 (0)31–772 1000

Printed at Chalmers, Göteborg, 2008

Abstract

Generic programming is an indispensable ingredient of modern programming languages. In the C++ programming language, generic programming is implemented by the means of parameterized fragments of code, called *templates*, in which parameters are constrained by *concepts*. This thesis consists of six publications investigating different aspects of generic programming with concepts—formal semantics of the concepts language feature, the mathematical foundations of concepts as a specification tool, change impact analysis of generic libraries, and multi-paradigm, multi-language library development.

Formally specifying the semantics of programming languages is a difficult task, one that is taken up rarely due to its complexity. In this thesis we provide a formal semantics of the *separate type checking* with concepts. We also describe some potential problems we discovered while translating the informal wording into formal rules. A formal semantics, such as ours, makes it easier to discuss language design, to improve the quality of the informal specification, and may serve as a model for compilers and other tools. Concepts as *institutions* is another view on the semantics of concepts we provide in this thesis. Institutions describe the parts of logics: *signatures* are the vocabulary, *sentences* are phrases that can be said given a vocabulary, and *models* are the subjects of the phrases. The institutions of concepts we provide make concepts into logics; the most basic general institution frames concepts as signatures and concept maps as sentences and models at the same time. The logic view on concepts makes it possible to apply techniques from the field of algebraic specification in the realm of C++ programming.

Generic programming with concepts also requires more practical support: in this thesis we describe a *conceptual change impact analysis for generic libraries*. Our change impact analysis is applied at the specification level of a generic library, which comprises the underlying concept hierarchy and generic interfaces of library functions and data structures. We apply the analysis to a real and important problem, a proposed change to the iterator concepts hierarchy of the Standard Template Library of C++.

Finally, we consider multi-paradigm development of generic libraries. We investigate how the datatype-generic paradigm applies to an imperative language such as C++: we provide a possible solution and outline the important differences between C++ and a functional language that must be taken into account. We also consider similarities between generic programming in Haskell and C++: based on a particular library we provide a translation from Haskell generic interfaces to the corresponding interfaces in C++.

Publications

This thesis is a compilation of the following publications:

1. M. Zalewski and S. Schupp. A semantic definition of separate type checking in C++ with concepts. *Journal of Object Technology*, 2009. Accepted for publication.

2. M. Zalewski and S. Schupp. Change impact analysis for generic libraries. *Proc. 22nd IEEE Internat. Conf. on Software Maintenance (ICSM)*, pages 35–44, Sept. 2006.

3. M. Zalewski and S. Schupp. C++ concepts as institutions. A specification view on concepts. In *Proc. ACM SIGPLAN Symp. on Library-Centric Software Design (LCSD)*, 2007. To appear.

4. M. Zalewski, A. P. Priesnitz, C. Ionescu, N. Botta, and S. Schupp. Multi-language library development: From Haskell type classes to C++ concepts. In J. Striegnitz, editor, *Proc. 6th Internat. Workshop on Multiparadigm Programming With Object-Oriented Languages (MPOOL)*, July 2007.

5. G. Munkby, A. P. Priesnitz, S. Schupp, and M. Zalewski. Scrap++: Scrap your boilerplate in C++. In *Proc. ACM SIGPLAN Workshop on Generic Programming (WGP)*, pages 66–75. ACM Press, 2006.

6. M. Zalewski. A semantic definition of separate type checking in C++ with concepts—abstract syntax and complete semantic definition. Technical Report 2008:12, Department of Computer Science and Engineering, Chalmers University, 2008.

The publications are preceeded by an introduction chapter.

All publications, except publications 4 and 6, have been fully peer-reviewed. Publication 4 has been peer-reviewed but without the extensive comments that have been provided in other cases. The publications are edited to fit the format of the thesis but are otherwise unchanged.

Acknowledgements

I would like to thank my adviser, Sibylle Schupp, who not only got me interested in the research in the first place, but also provided constant support and invaluable feedback throughout my graduate studies. I would also like to thank my officemates, Kyle Ross, Gustav Munkby, and Jean-Philippe Bernardy (chronological order), for many interesting exchanges and a pleasant working environment; Andreas Priesnitz, a member of my research group, has been a great research inspiration, a source of many interesting arguments (some heated but all friendly), and always ready to eat good food or drink a good coffee. The department as a whole has been inspiring, interesting, and supportive; I really appreciate the espresso machine with a constant supply of free coffee and milk. Some people I would like to thank in particular are Aslan Askarov, who is always ready to try something new and who convinced me to start climbing, Devdatt Dubhashi, from whom I have learned much about Marxism and many other interesting things, Harald Hammarström, for being Harald, Daniel Hedin, for interesting conversations and fun time in a virtual world, Rogardt Heldal, for being a great friend, and Karol Ostrovsky, for long discussions about photography and photographic equipment. On a broader scale, I would express my appreciation of Göteborg, and Sweden in general, which is a great place to live in. Last, but definitely not least, I thank my family for support, especially my wife who followed me abroad on a 5-years-long adventure.

Contents

Introduction **1**
1 Generic Programming . . . 1
2 Concepts . . . 2
2.1 Concept Analysis . . . 2
2.2 Concepts in the Standard Template Library . . . 3
2.3 Formalizations . . . 5
3 C++ Concepts Tutorial . . . 7
3.1 A Min(imal) Example . . . 7
3.2 Associated Types and Requirements . . . 8
3.3 Concept Refinement and Concept-Based Overloading . . . 10
4 Comparison with Related Language Features . . . 12
4.1 Comparison Criteria . . . 12
4.2 Generics in Java and C# . . . 13
4.3 Haskell Type Classes . . . 14
5 Specification and Programming with Concepts . . . 17
5.1 Algebraic Specification . . . 17
5.2 Constrained Polymorphism . . . 20
6 Concept Applications . . . 21
7 Supporting Concepts . . . 22
7.1 Formalization . . . 23
7.2 Software Analysis and Transformation . . . 25
8 Summary and Contributions . . . 26
9 Outlook . . . 27

Paper I – A Semantic Definition of Separate Type Checking in C++ with Concepts **29**
1 Introduction . . . 31
1.1 Notation . . . 32
2 Concepts, Name Lookup, and Implementation Binding . . . 32
2.1 Introduction to Separate Type Checking . . . 33
2.2 Refinement, Name Lookup, and Implementations Binding . . . 34
3 Formalization . . . 38
3.1 Definition and Conventions . . . 38
3.2 X++ Language . . . 39
4 Semantics of Separate Type Checking . . . 40
4.1 Program Correctness . . . 41
5 Illustration and an Argument for Correctness . . . 48
5.1 Examples of Separate Type Checking . . . 49

5.2 Correctness Argument . . . 56
6 Interpretation of the Informal Definition . . . 63
6.1 Name Lookup . . . 63
6.2 Implementation Binding . . . 64
6.3 Type Checking of Concept-Constrained Templates . . . 66
7 Related Work . . . 66
8 Conclusions and Future Work . . . 68

Paper II – Change Impact Analysis for Generic Libraries **71**
1 Introduction . . . 73
2 Related Work . . . 74
3 Concepts in C++ . . . 76
4 Example . . . 77
5 Conceptual Change Impact Analysis . . . 79
5.1 Conceptual Dependence Graph . . . 79
5.2 Impact Propagation . . . 80
5.3 Filtering for Specific Impact . . . 80
6 Case Study . . . 82
6.1 Iterator Concept Taxonomies . . . 82
6.2 Experiment Setup . . . 83
6.3 Compatibility . . . 84
6.4 Constraints Change of Parameters . . . 86
7 Conclusions . . . 86
8 Future Work . . . 87

Paper III – C++ Concepts as Institutions **89**
1 Introduction . . . 91
2 Motivation . . . 92
3 Background . . . 95
3.1 C++ Concepts . . . 95
3.2 Categories and Functors . . . 98
3.3 Institutions . . . 100
4 Concepts as Specifications . . . 102
4.1 Signatures and Models . . . 103
4.2 Concept Institutions . . . 108
5 Related Work . . . 114
6 Conclusions and Future Work . . . 115

Paper IV – Multi-Language Library Development **117**
1 Introduction . . . 119
2 Background and Terminology . . . 120
3 Translation Rules . . . 123
3.1 Top-Level Translation Steps . . . 124
3.2 Internal translation steps . . . 125
4 Conclusions and Future Work . . . 126
Appendix A: Complete Translation Rules . . . 129

Paper V – Scrap++: Scrap Your Boilerplate in C++ **131**
1 Introduction . . . 133
2 The Paradise Example . . . 134
3 SYB in C++ . . . 135
3.1 Recursive Traversal . . . 135
3.2 One-Layer Traversal . . . 136
3.3 Type Extension . . . 140
4 Defining One-Layer Traversal . . . 141
5 Generating Generic Folds . . . 142
6 SYB for Classes . . . 144
7 Generalization and Extension . . . 145
7.1 Type-Case Extensions . . . 146
7.2 Parallel traversals . . . 146
7.3 Generic queries . . . 146
8 Related Work . . . 147
9 Conclusions and Future Work . . . 148
Appendix A: Complete example . . . 151

Paper VI – A Semantic Definition of Separate Type Checking in C++ with Concepts. Abstract syntax and the complete semantic definition. **157**
1 Introduction . . . 159
2 Syntax . . . 160
2.1 Grammar Rules . . . 160
2.2 Subrules . . . 174
3 Helper Judgements . . . 174
4 Type System . . . 179
5 Separate Type Checking Safety . . . 196
6 Statistics . . . 197

References **199**

Introduction

Support for *geneneric programming*, in its different incarnations, is an indispensable ingredient of modern programming languages. Generic programming has long been the staple of *software libraries* in the Haskell (Peyton Jones, 2003) and C++ (ISO, 2003b; Stroustrup, 1997) languages and has recently been introduced in mainstream object-oriented languages, such as Java (Gosling et al., 1996) and C# (Kennedy and Syme, 2001), under the name of *generics*.

1 Generic Programming

Summarizing Gibbons (2007), *generic programming* is about making programs more flexible without compromising safety. Yet, making programs more flexible can be achieved in many different ways and, consequently, the understanding of what actually constitutes generic programming varies between different communities. Gibbons notes that the common factor connecting the different kinds of generic programming is parameterization of programs by some *non-traditional* parameters. What is traditional and what is not, however, depends strongly on the programming tradition of a community.

Gibbons distinguishes 7 main styles of genericity:

- *genericity by value*, which corresponds to the nowadays fundamental constructs of functions and procedures;
- *genericity by type*, in which programs can be parameterized by types;
- *genericity by function*, where programs can be parameterized by functions;
- *genericity by structure*, in which algorithms are separated from data structures through the use of abstract requirement specifications (Siek et al., 2002);
- *genericity by property*, where genericity by structure is augmented by logical constraints on the abstract requirements;
- *genericity by stage*, where programs themselves can be manipulated by other programs; and
- *genericity by shape*, or *datatype genericity*, where programs are parameterized by the shape of the data.

All of the generic programming approaches allow capturing commonalities between some classes of programs—the difference is in which commonalities can be extracted.

In this thesis we consider generic programming as it is known in the C++ language (ISO, 2003b; Stroustrup, 1997). This kind of generic programming in C++ relies heavily on the idea of *concepts* and combines genericity by structure and property.

2 Concepts

> Generic programming is a sub-discipline of computer science that deals with finding abstract representations of efficient algorithms, data structures, and other software concepts, and with their systematic organization. The goal of generic programming is to express algorithms and data structures in a broadly adaptable, interoperable form that allows their direct use in software construction. (Jazayeri et al., 2000)

Although the above quotation does not define and, for that matter, even mention *concepts*, it provides the motivation for concepts in generic programming. In fact, the seminar at Schloss Dagstuhl, at which the quoted declaration was made, is an important point in the history of generic programming with concepts as practiced in the C++ community today.

Concepts represent the "abstract representations of efficient algorithms, data structures, and other software concepts," and also allow their "systematic organization." Yet, for a given problem domain, a taxonomy of concepts is not apparent or inherent. On the contrary, a crucial part of the generic programming approach is the careful process of *concept analysis* in which essential data structures and algorithms are identified "in a broadly adaptable, interoperable form that allows their direct use in software construction."

The Standard Template Library (STL) (Austern, 1998; Musser et al., 2001; Stepanov and Lee, 1994 (revised in October 1995 as tech. rep. HPL-95-11) of C++ was the first popular, successful library based on a carefully chosen taxonomy of concepts describing common forms of containers and *iterators* for traversing data. STL is a real-world, practical C++ library and since C++ does not directly support concepts, concepts in STL are a design and documentation device. The usual implementations are structured after the underlying concept taxonomy but only some of the conceptual framework can be directly used for error checking and optimization in the C++ *template* code that comprises any STL implementation.

The experience with STL and other libraries (e.g., the libraries from the Boost initiative (Boost)) demonstrated the usefulness of generic programming with concepts. At the same time, it became apparent that C++ templates alone do not harness the full power of concepts for error checking, optimization, and interoperability. There have been multiple formalizations of concepts, starting with the pre-STL ideas from the algebraic specification language Tecton (Kapur et al., 1981b), and later formalizations that bring concepts closer to C++ programming. These efforts have recently culminated in planned inclusion of support for concepts into C++0x (Gregor et al., 2008e).

In the remainder of this section we briefly discuss the process of concept analysis, the use of concepts in STL, and the most important formalizations of concepts.

2.1 Concept Analysis

A generic library consists of generic algorithms—"parameterized procedural schemata that are completely independent of the underlying data representation and are derived

Description The BIDIRECTIONALITERATOR concept defines types that support traversal over linear sequences of values, including support for multi-pass algorithms, and stepping backwards through a sequence (unlike FORWARDITERATORS). A type that is a model of BIDIRECTIONALITERATOR may be either mutable or immutable, as defined in the TRIVIALITERATOR requirements.

Refinement of FORWARDITERATOR

AssociatedTypes The same as for FORWARDITERATOR

Notation Let `X` be a type that is a model of BIDIRECTIONALITERATOR; `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	Expr.	Return Type
Predecrement	`--i`	`X&`
Postdecrement	`i--`	`X`

Expression Semantics

Expr.	Precondition	Semantics	Postcondition
`--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`.
`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 BIDIRECTIONALITERATOR are guaranteed to be amortized constant time.

Figure 1.1: Definition of the BIDIRECTIONALITERATOR concept of STL (SGISTL).

from concrete efficient algorithms" (Musser and Stepanov, 1989). Such generalization of concrete algorithms requires systematic identification and organization of *concepts* of a given problem domain, where each concept describes a generic specification of a family of types (a type that meets the specification is said to *model* the concept). The process of "lifting" concrete algorithms, called *concept analysis*, results in a set of generic algorithms and a corresponding concept taxonomy that captures the requirements of these algorithms.

STL, now a part of C++ standard, was the first main-stream library to be based on the result of systematic concept analysis. The successful application of concept analysis in STL has inspired many other generic libraries, including Boost Graph Library (Siek et al., 2002), the Boost Iterator Library (Abrahams et al., 2008), the μBLAS Library (Walter and Koch, 2008), the Computational Geometry Algorithms Library (CGAL), or the Generic Linear Algebra Software (GLAS) initiative.

2.2 Concepts in the Standard Template Library

Concepts in STL are specified in a de-facto standardized documentation. In illustration, figure 1.1 shows the description of the BIDIRECTIONALITERATOR concept adopted from the documentation of STL (SGISTL). The row "Refinement of" specifies that the BIDIRECTIONALITERATOR includes all the requirements of the FORWARDITERATOR concept. The "Associated Types" row and "Valid Expressions" table specify syntactic requirements of the concept while the "Expressions Semantics" table and "Complexity Guarantees" row specify semantic requirements.

The concepts of STL belong to the four categories of container, iterator, functor, and utility concepts. Perhaps the most interesting category are the iterator concepts,

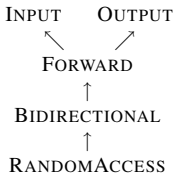


Figure 1.2: The taxonomy of iterator concepts in STL (SGISTL).

which capture the operations related to range traversal and data access. The taxonomy of iterator concepts is shown in figure 1.2. Each iterator concept captures a particular kind of iterators; the OUTPUTITERATOR concept, for example, specifies iterators for ranges that can be traversed only once and that contain writable but not necessarily readable data, while the RANDOMACCESSITERATOR concept specifies all iterators with efficient random access operation that allow both writing and reading of data.

Most of the generic algorithms of STL are specified in terms of the iterator concepts. For example, the `accumulate` algorithm, elsewhere also known as a generic `fold` algorithm, is specified as follows:

```
template <class InputIterator, class T, class BinOp>
T accumulate(InputIterator first, InputIterator last,
             T init, BinOp op)
{
  for (; first != last; ++first)
    init = op(init, *first);
  return init;
}
```

The algorithm `accumulate` traverses the range between the iterators `first` and `last` and combines the elements of the range using the binary operation `op` (which models the BINARYFUNCTION concept) with `init` as the initial value. There are two important things to notice. First, the algorithm is expressed in abstract terms without any reference to any particular data structure. Second, the requirements of the algorithm are expressed in terms of concepts and the algorithm can be safely applied to any data structures that model these concepts. For example, `accumulate` can be applied to a vector of integers as well as a stream of strings read from the standard input:

```
// accumulate applied to a vector of ints
vector<int> v;
... initialization of v ...
int i = accumulate(v.begin(), v.end(), 0, plus<int>());
// accumulate applied to a stream of strings
string result = accumulate(istream_iterator<string>(cin),
    istream_iterator<string>(), "", plus<string>());
```

Although STL had a tremendous influence on subsequent generic libraries in C++, many years of experience exposed certain problems with its concept taxonomy. Most importantly, iterator concepts entangle the concerns of range traversal and data access (Siek, 2001). Because of the unnecessarily strong coupling between value-access and traversal requirements, some generic algorithms are under-generalized and some iterator types incorrectly categorized with respect to their traversal protocol (see, e.g., C++ Standard Library issue 96 (Iss96) and Sutter's paper (Sutter, 1999)). A new taxonomy

of iterator concepts (Gregor et al., 2008d; Siek et al., 2004) is currently discussed by the C++ standard committee. This discussion shows that concept taxonomies even of widely used and accepted libraries are subject to change—the impact of changing the iterator concepts on the users of STL is the subject of the case study in Paper II.

2.3 Formalizations

Although concept analysis has been performed in many generic libraries, it lacks a complete theoretical background. The origins of concepts in generic programming can be traced to the Tecton algebraic specification language (Kapur and Musser, 1992); however, the Tecton formalization of concepts does not fully coincide with concepts in practice. Willcock et al. (2004) have extended the original Tecton formulation and Siek and Lumsdaine (2005) extended System F (Girard, 1972) with concept-constrained polymorphism. Most recently, direct support for concepts has been proposed for C++ (Gregor et al., 2006b; Järvi et al., 2006; Dos Reis and Stroustrup, 2006) and concepts are almost certain to be included in C++0x (DevX.com, 2008).

2.3.1 Concepts in Tecton

In Tecton terminology, a *concept* is a set of many-sorted algebras; the definition below is taken from Kapur and Musser (1992).

Definition 1 (*Concept*)**.** We assume the existence of two disjoint sets of identifiers (strings), called *sorts* and *function identifiers*. An *indexed family of sets* is a family of pairs (s,S), where s is a sort and S is a set. An *indexed family of functions* is a family $\mathscr{F}$ of pairs (σ,f), where f is a function and σ is a *function description* consisting of a function identifier and an *arity*, which is a list of distinct sorts (called *domain sorts*) and another sort (called the *range sort*). Then a *(many-sorted) algebra* is

- an indexed family $\mathscr{S}$ of sets, such that $\mathscr{S}$ is a mapping (i.e., if $(s,S_1) \in \mathscr{S}$ and $(s,S_2) \in \mathscr{S}$ then $S_1 = S_2$);
- an indexed family $\mathscr{F}$ of functions, such that $\mathscr{F}$ is a mapping and for each pair (σ,f) in $\mathscr{F}$, where σ has domain sorts $s_1,\ldots,s_2$ and range sort r, f is a function with domain $\mathscr{S}(s_1) \times \ldots \times \mathscr{S}(s_n)$ and range $\mathscr{S}(r)$.

Finally we define a *concept* as a set of many-sorted algebras.

The main goal of Tecton is to facilitate *organization and understanding* of abstract notions irrespective of implementations questions (Kapur et al., 1981b). Usually, abstraction languages allow one to ignore implementation details, that is, the "how" of a computation. Tecton, in addition, allows one to also abstract from parts of the computation itself, that is, the "what" of the computation, making it possible to capture only the essential properties of a notion without being forced to specify unneeded details and to carry these essential properties to specializations. Examples of concept taxonomies described in Tecton include the descriptions of STL container and iterator concepts (Musser, 2008) and of basic algebraic concepts (Musser et al., 1999). In addition, Schwarzweller (2003) has shown how to express complexity guarantees, a crucial part of the STL concepts, in Tecton concept descriptions.

2.3.2 A Formalization of Concepts for Generic Programming

In a programming language, abstractions (members of a concept) are types rather than many-sorted algebras. Types, unlike algebras, do not necessarily provide an explicit mapping from all sorts (type names) and functions required by a concept to actual types and operations defined for these types. Consequently, a language-dependent lookup function is necessary to define a complete mapping from a set of types that determines the concept modeling relation to all particular types and operations required by a given concept. Another shortcoming of the many-sorted algebras definition, in the context of programming languages, is that components of concepts are limited to sorts (representing types) and functions. In generic programs, however, concepts may constrain type constructors as in the Haskell `Monad` class (Jones, 1993), for example, or in the `Allocator` class in STL where the `rebind` member behaves like a type constructor. Polymorphic functions are similar to type constructors; the type of a polymorphic function depends on the type arguments the function is instantiated with.

Willcock et al. (2004) give a definition of concepts that addresses the shortcomings of the algebraic definition. Many-sorted algebras are replaced with ML-like structures (Milner et al., 1990a) and a concept is defined as a set of structures over a signature rather than as a set of algebras over a Tecton concept description (a Tecton signature). The ML structures and signatures are extended with a possibility of nesting and signatures are furthermore extended to allow polymorphic functions. Moreover, and fundamentally differently from the algebraic definition, the extended definition includes a language-dependent *lookup function* that completes a model of a concept, given a set of determining types. Concepts are defined as follows:

Definition 2. 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. (Willcock et al., 2004)

The set T of type requirements matches types to required kinds (kinds as in System F_ω (Pierce, 2002)), the set V specifies the set of values constrained by constant types, concept member types, elements of nested signatures, or type functions in the style of System F_ω with parameters bound by other concepts, and the set E contains those pairs of types of kind $*$ that specify same-type constraints.

2.3.3 Formalization of concepts in F^G

F^G, introduced by Siek and Lumsdaine (2005), defines concepts from a language design point of view. While in Tecton and in the formalization of concepts by Willcock et al. (2004) concepts are a way of organizing and understanding abstract notions, F^G treats concepts strictly on the level of language (i.e., type system and semantics) design. F^G denotes a type system and semantics for concepts based on System F (Girard, 1972), which "combines some of the best features of Haskell and ML related to generic programming" (Siek and Lumsdaine, 2005, p.3). Concepts are similar to type classes

in Haskell in that type classes are based purely on parametric polymorphism. Yet, since F^G does not include Hindley-Milner type inference, there are also crucial differences. For example, in F^G two concepts can share a member name, in contrast to Haskell, where type classes in the same module must assign different names to their members. *Associated types* and *same-type constraints* in F^G are comparable to nested types and type sharing in ML respectively. In difference from ML functors, which provide genericity on the module level, genericity in F^G is on the more fine-grained level of functions.

2.3.4 Concepts for C++0x

Already in 1994, Stroustrup, in his "Design and Evolution of C++" book, discussed the need for checking of template arguments in C++ and outlined possible ways of doing so. Still, the ideas were not developed and tested enough to make it into the C++ language. More than a decade later, concepts are now formally considered by the C++ standardization committee for inclusion into the language (Gregor et al., 2008f) (a list of other C++ standardization papers related to concepts can be found at `http://www.generic-programming.org`). The current concepts proposal allows one to capture the syntactic requirements through associated types, templates, and *pseudo-signatures* of associated functions, and some of the semantic requirements through *axioms*, which roughly correspond to conditional equations in algebraic specifications. The modeling relation between types and concepts is established through *concept maps*, which map the concrete syntax of types to the abstract syntax of concepts. Concepts can be used in templates to *constrain* type parameters. Both concept maps and constrained templates are checked at the time of their definition, resulting in *separate type checking* of templates and their arguments (with some exceptions discussed by Järvi et al. (2006) and Gregor (2008c)). The concept tutorial in section 3 gives a brief introduction to concepts for C++0x.

3 C++ Concepts Tutorial

In this section, we give a brief tutorial-like introduction to C++0x concepts (loosely based on a tutorial by Gregor (2008b)).

In C++, generic libraries are implemented using *templates*, which are pieces of code parameterized by types and values. The requirements on these type parameters are implicit. For example, the STL concepts, such as the one shown in figure 1.1, document the contract between the designer and the user of a generic library but they are not supported by the C++ language. In effect, when using generic code a user may have to face cryptic error messages that refer to code that the user was never intended to see. Concepts as a language feature allow explictly specifying and automatically checking the contract between the designer and the user.

3.1 A Min(imal) Example

One of the simplest function templates, `min()`, picks a minimal value:

```
1 template<typename T>
2 const T& min(const T& x, const T& y) {
3   return x < y? x : y;
4 }
```

Despite being simple, the `min()` function places 2 requirements on its template parameter `T`: two values of type `T` must be comparable by a less-than operator (`<`) and the result of the comparison must be convertible to `bool`. In C++0x, the requirements can be stated explicitly as a concept (using the new `concept` keyword):

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

template<typename T>
requires LessThanComparable<T>
const T& min(const T& x, const T& y) {
  return x < y? x : y;
}
```

The concept LESSTHANCOMPARABLE has a single parameter `T` (every concept must have at least one parameter) and its body contains a *pseudo-signature* of a less-than operator that returns `bool`. The conceptualized `min()` algorithm *constrains* its template parameter `T` in a `requires` clause, to model the LESSTHANCOMPARABLE concept.

The conceptualized `min()` is *separately type-checked* by the compiler: the compliance of the `min()` function and the types to which it is to be applied is checked independently before `min()` can be called with these types. At the call site, when `min()` is used, the compiler only checks if a concept map for the argument type has been defined. A concept map for `int` may be defined as follows:

```
concept_map LessThanComparable<int> { }
```

If the body of the concept map is left empty, as in the code snippet above, the compiler tries to fulfill the requirements of the LESSTHANCOMPARABLE concept automatically, finding that `int` has a built-in, appropriate less-than operator. On the other hand, less-than comparison for `int`s can be given a definition different from the built-in one explicitly in a concept map:

```
concept_map LessThanComparable<int> {
  bool operator<(const int& x, const int& y)
    { return abs(x) < abs(y); }
}
```

Furthermore, for concepts applicable to many types, such as the LESSTHANCOMPARABLE concept, the `auto` keyword can be used to have the compiler define concept maps automatically if possible:

```
auto concept LessThanComparable<typename T> {
  bool operator<(const T& x, const T& y);
}
```

3.2 Associated Types and Requirements

Listing 1.1 lists a POLYGON concept and an algorithm that computes a perimeter of a polygon. One possible polygon is a simple implementation of a triangle:

```
class triangle {
public:
  int sides[3];
```

```
concept Polygon<typename P> {
  int num_sides(const P&);
  double side_length(const P&, int index);
}

template<typename P>
requires Polygon<P>
double perimeter(const P& poly) {
  double sum = 0;
  for (int i = 0; i < num_sides(poly); ++i)
    sum += side_length(poly, i);
  return sum;
}
```

Listing 1.1: POLYGON concept and perimeter calculation algorithm.

```
};

int num_sides(const triangle& tri) { return 3; }
int side_length(const triangle& tri, int index) {
  return tri.sides[index];
}

concept_map Polygon<triangle> { }
```

The signature of the `num_sides` function for triangles matches the corresponding signature in the concept exactly but the return type of the `side_length` function is `int` instead of the required `double`. Such mismatch is allowed as long as the required conversions follow the usual C++ rules. While converting `int` to `double` works, the conversion imposes floating-point arithmetic, possibly decreasing performance and precision. Instead of hard-coding the type of polygon lengths, the POLYGON concept can require an *associated type* to represent length, making the concept more generic:

```
concept Polygon<typename P> {
  typename length_type; // associated type
  int num_sides(const P&);
  length_type side_length(const P&, int index);
}
```

Then the `perimeter()` algorithm has to be rewritten to use the associated type introduced in the concept but a naive rewrite introduces errors:

```
template<typename P>
requires Polygon<P>
Polygon<P>::length_type perimeter(const P& poly) {
  Polygon<P>::length_type sum = 0; // error!
  for (int i = 0; i < num_sides(poly); ++i)
    sum += side_length(poly, i); // error!
  return sum; // error!
}
```

The new version of the algorithm uses the associated type `Polygon<P>::length_type` to perform its computations. This type, however, is "blank"—it does not come with the necessary operations. The `length_type` associated type is essentially a number; as

such it has to support operations that can be performed on numbers, which are captured by the following concept:

```
auto concept Numeric<typename T> {
  T::T(const T&);              // copy construction
  T::T(int);                   // construction from an int
  T::~T();                     // destructor
  T& operator+=(T&, const T&); // addition
}
```

Given the NUMERIC concept, the signature of the `perimeter()` algorithm can be rewritten as follows:

```
template<typename P>
requires Polygon<P> && Numeric<Polygon<P>::length_type>
Polygon<P>::length_type perimeter(const P& poly);
```

The new requirements clause first constrains the parameter `P` to be a POLYGON and then further constrains POLYGON's associated type `length_type` to model the NUMERIC concept. A careful reader, though, may notice that `length_type` has to be NUMERIC to be used in a meaningful way, forcing the `requires` clause to contain two requirements for all algorithms that operate on the lengths of polygon sides. Assuming that the library designer deems it appropriate that the associated type `length_type` should always model the NUMERIC concept, this intent can be stated as an *associated requirement*:

```
1 concept Polygon<typename P> {
2   typename length_type;
3   requires Numeric<length_type>;
4   int num_sides(const P&);
5   length_type side_length(const P&, int index);
6 }
```

Line 3 hard-codes the NUMERIC requirement on the associated type `length_type` in the POLYGON concept. This requirement avoids unnecessary repetition of constraints in each algorithm that operates on the lengths of polygon sides and it signals the intent of the library designer that a polygon with non-numeric sides does not "make sense" in the library's framework. With the addition of the associated requirement the signature of the `perimeter()` function can be rewritten as follows:

```
template<Polygon P>
P::length_type perimeter(const P& poly);
```

Since the sole requirement on the parameter `P` now is the POLYGON concept, the requirement can be stated in the "predicate" form `Polygon P`, where the name of the concept, `Polygon` in this case, replaces the `typename` keyword in template parameters list. The concept mechanism includes many such "convenience" features (type conversion in pseudo-signatures, implicitly generated concept maps, etc.); while these features make coding easier, they complicate any attempt to formally specify the semantics of concepts.

3.3 Concept Refinement and Concept-Based Overloading

Concepts often form hierarchies; for example, every equilateral polygon is a polygon but the relation does not always hold in the other direction. This *refinement* relation is

directly supported by C++ concepts:

```
concept EquilateralPolygon<typename P> : Polygon<P> { }
```

Here, the EQUILATERALPOLYGON concept "inherits" all the operations of the POLYGON concept but in addition it requires that the lengths of all sides are the same[1] and every type that models the EQUILATERALPOLYGON concept also models the POLYGON concept. A concept map stating that a `square` class models the EQUILATERALPOLYGON automatically generates an implicit concept map for the POLYGON as well:

```
class square {
public:
  int length;
};

concept_map EquilateralPolygon<square> {
  typedef int length_type;

  int num_sides(const square&) { return 4; }

  int side_length(const square& sq, int) {
    return sq.length;
  }
}
// concept_map Polygon<square> implicitly defined
```

Given the two polygon concepts, the `perimeter()` algorithm can be overloaded, based on the concepts that its arguments model. Perimeter may be computed efficiently for equilateral polygons:

```
template<EquilateralPolygon  P>
P::length_type perimeter(const P& poly) {
  return num_sides(poly) * side_length(poly, 0);
}
```

The compiler chooses (at the call site) the *more specialized* version of the algorithm where specialization is determined by the refinement relation lattice: if a type models only the POLYGON concept, the perimeter is computed by linearly traversing the lengths of all sides but if a type models both the POLYGON and the EQUILATERALPOLYGON concepts, the perimeter can be computed in one step by multiplying the length of all sides by the number of sides.

In this brief overview we introduce the most commonly used concept features but we elide the discussion of advanced concept uses and of the complex mechanism involved in checking concepts. The wording considered by the C++ standardization committee (for the most recent version at the time of writing see Gregor et al., 2008f) is the authoritative source on the semantics of concepts. More information about concepts, including the list of issues in the current definition, is maintained at `generic-programming.org`.

[1] The additional requirement is implicitly signaled by the name of the concept. Some of such requirements can be stated explicitly using *axioms*, which correspond to conditional equations.

Criterion	Definition
Multi-type concepts	Multiple types can be simultaneously constrained.
Multiple constraints	More than one constraint can be placed on a type parameter.
Associated type access	Types can be mapped to other types within the context of a generic function.
Constraints on associated types	Concepts may include constraints on associated types.
Type aliases	A mechanism for creating shorter names for types is provided.
Separate compilation	Generic functions can be type checked and compiled independently of calls to them.
Implicit argument deduction	The arguments for the type parameters of a generic function can be deduced and do not need to be explicitly provided by the programmer.

Table 1.1: Glossary of evaluation criteria (Garcia et al., 2007).

4 Comparison with Related Language Features

So far, we have claimed that concepts are a suitable language construct for generic programming, but it will be instructive to reflect upon their suitability and also to assess the suitability of related features in other programming languages, including those from other paradigms—how well do, for example, generics in Java or Generic C# support the design of a generic library? The most widely cited papers in this context are the studies by Garcia et al. (Garcia et al., 2003, 2007) that compare generics of altogether 8 languages: C++ without concepts, Java, C# generics, Eiffel, Standard ML, Objective Caml, Cecil, and Haskell. Their studies are of practical nature: the authors designed a small, but realistic graph library and fully implemented it in each of the languages. The result of their experiment is a set of 8 language features that the authors consider necessary for generic software design. None of the languages evaluated support all 8 features, but, perhaps unexpectedly, the Haskell language scores highest.

The obvious two threats to the validity of the study are that an implementation of the model library might not be idiomatic, and that the set of comparison criteria lacks a formal argument. Still, those criteria were important enough to guide the designers of Haskell and C++: with explicit reference to those studies, the Haskell designers extended the language by *associated types*, to get the full score. Similarly, C++ designers extended their language by concepts.

In this section, we first summarize the comparison criteria, then we discuss generic programming in Java and C#. Finally, we provide an extended comparison with type classes in the Haskell language, the construct that is closest to C++ concepts.

4.1 Comparison Criteria

To compare generic programming features in different languages, it is useful to first establish the kinds of features necessary and then give them, for the purpose of comparison, common names. The ingredients of generic programming, using concepts

terminology, can be summarized as follows:

- A *concept* specifies the common syntax and semantics of a set of types.
- *Refinement* is a relation between concepts, corresponding to an inclusion relation between the set of types that concepts specify.
- *Modeling* is the relation between types and concepts; a type is said to *model* a concept if it satisfies the requirements of a concept.
- *Algorithm* is a generic piece of code, parameterized by types.
- *Constraints*, stated in terms of concepts, restrict which arguments can be given for type parameters.

This terminology allows connecting the features in different languages.

Garcia et al. name 8 comparison criteria by which generics in different languages can be compared; the criteria are listed in table 1.1. Each criterion names a feature useful for generic programming. The first two criteria make it possible to modularize implementations and specifications, the next two permit "bundling" of types and constraints, type aliases make long generic types more convenient to use, separate compilation eases the development process, and implicit argument deduction makes it easy to invoke generic algorithms.

4.2 Generics in Java and C#

Generics in both Java and C# are integrated with object-orientation. Concepts correspond to interfaces in both languages, modeling corresponds to implementing of, in Java, and inheriting from, in C#, interfaces, refinement is represented as inheritance between interfaces, and constraints are given via interfaces. The following snippet of code (Garcia et al., 2007) is a simple example of a generic program in Java (the code in C# is the same modulo syntax differences):

```
1  public interface Comparable<T>
2  {  boolean better(T x);  }
3
4  public class pick {
5    public static <T extends Comparable<T>>
6    T go(T x, T y)  { if (x.better(y)) return x; else return y; }
7  }
8
9  public class Apple implements Comparable<Apple> {
10   public Apple(int r)  { rating = r; }
11   public boolean better(Apple y)  { return rating > y.rating; }
12   private int rating;
13 }
14
15 public class Main {
16   public static void main(String[] args) {
17     Apple a1 = new Apple(3), a2 = new Apple(5);
18     Apple a3 = pick.go(a1, a2);
19   }
20 }
```

A concept COMPARABLE is given as a parameterized interface that contains one method for comparing one object to another. An algorithm is represented as a class with a parameterized method, with parameters given between angle brackets. Constraints are given together with parameters; in this case, the parameter `T` must extend the `Comparable` interface instantiated with the parameter `T` itself. When the method `better` is called, it is checked against the constrained interface. The type `Apple` extends the interface `Comparable` instantiated with `Apple` itself; this establishes the modeling relation and ensures that `Apple` implements the `better` method. Finally, the generic method is called on line 17. The type parameter is implicitly deduced to be the type `Apple`.

Generics in object-oriented languages, including Java and C#, differ significantly from C++ templates. In Java, the type information given by generic interfaces is only used during compile time to guarantee safety but does not appear at run time—this loss of information is called *type erasure*. Type erasure maintains compatibility with previous versions of Java but it disables run-time operations that depend on type information, constructor calls on type parameters are not allowed, for example. C#, on the other hand, generates run-time representations of generics to allow for efficient implementation and reflection, among others.

Object-oriented generics put a high premium on simplicity and compatibility with the existing object-oriented features; accordingly, Java and C# support only some of the criteria from table 1.1. Both languages do not support constraining multiple types simultaneously. Although interfaces with multiple type parameters are allowed, they do not group constraints. For example, a constraint `T extends C<T, P>` requires repetition of all constraints from `C` on `P`. Both languages support multiple constraints, since a parameter may be required to extend more than one interface. Associated types may be represented as multiple parameters to an interface but this representation requires that all associated parameters are spelled out everywhere they are used. In addition, since simultaneous constraints are not supported, all constraints on associated types must be repeated, possibly resulting in bulky interfaces. Neither Java nor C# support type aliases, which makes complex generic interfaces difficult to use. Because inheritance has to be declared when a class definition is given, retroactive modeling is not possible in Java; in C# partial classes allow retroactive inheritance within a single compilation unit. Java supports implicit instantiation fully: parameter types are unified with argument types and constraints may be used to infer the type of parameters not unified with any explicit argument. C# requires that either all parameter types are given by unification with arguments or that the user specifies parameter types explicitly, in addition to run-time arguments. This is necessary because C# generates specialized versions of generic types. Finally, both languages naturally support separate compilation since generics are based on object-oriented features that already separate-compile.

4.3 Haskell Type Classes

In the original comparison study by Garcia et al., Haskell was the only language that fully supported all generic programming features listed in table 1.1, with the only proviso of difficult handling of associated types. C++ was found to also support all relevant features except separate compilation, but its lack of *explicit* support for concepts and constraints made half of the features irrelevant. Since the publication of the study, however, both Haskell and C++ have changed: the current version of GHC (the Glasgow Haskell Compiler) provides new features for associated datatypes (Chakravarty et al., 2005b) and associated type synonyms (Chakravarty et al., 2005a); as already discussed, C++0x, the next version of C++, will turn concepts from mere documentation artifacts

into a fully supported language feature. Due to those language extensions, the deficiencies that the earlier comparison revealed—in particular on part of C++—no longer exist: C++0x now provides as much support for concepts, associated types, and retroactive modeling as Haskell does. If one were to apply the previous evaluation criteria again, Haskell and C++0x would, thus, be indistinguishable as generic-programming languages, excluding the difference in separate compilation support, which remains. A follow-up study (Bernardy et al., 2008) has therefore further refined the set of evaluation criteria. In this subsection, we first show, by way of example, the commonalities of Haskell type classes and C++ concepts, then we discuss the differences in the language features, many stemming from the differences in philosophies and basic design.

The following two code snippets reformulate the previous Java example from Garcia et al. (2007). First the Haskell version:

```
1  class Comparable t where
2    better :: (t, t) -> Bool
3
4  pick :: Comparable t => (t, t) -> t
5  pick (x, y) = if(better(x,y)) then x else y
6
7  data Apple = Apple Int
8
9  instance Comparable Apple where
10   better (Apple x, Apple y) = x > y
11
12 a1 = Apple 3; a2 = Apple 5
13 a3 = pick(a1, a2)
```

Next the C++ version:

```
1  concept Comparable<typename T> {
2    bool better(T, T);
3  }
4  template<Comparable T>
5  T pick(T x, T y) { if(better(x,y)) return x; else return y; }
6
7  struct Apple { int r; }
8
9  concept_map Comparable<Apple> {
10   Apple better(Apple x, Apple y) { return x.r > y.r; }
11 };
12 Apple a1, a2;  a1.r = 3;  a2.r = 5;
13 Apple a3 = pick(a1, a2);
```

The examples can be matched, line by line. The Haskell `Comparable` class corresponds to the C++ concept. The Haskell `pick` implementation has the type explicitly spelled out, but this is not strictly necessary. In Haskell, the `better` function requirement is visible in the surrounding code (no duplicates may be defined) and type inference finds the requirement for `Comparable` from the function application in `pick`. An `Apple` type containing an integer is declared in both snippets (on line 7) and the modeling declaration is given as an `instance` in Haskell and a `concept_map` in C++. Finally, some apples are created and then picked on the two last lines of both snippets.

The example shows how close generic programming features in both languages are, it is really difficult to see a significant difference. Yet, type inference in Haskell already makes an important difference visible: in C++, constraints must be given explicitly and

generic code is then checked against the environment given by the constraints, while in Haskell, type classes extend the global environment so type inference can simply use function names to infer constraints. Next, we give examples of some interesting differences, but we first remark on the motivation behind generic programming in both languages. Haskell is intended for high-level programming with emphasis on clarity and correctness. C++ on the other hand, wants to provide high-level features around low-level, high-performance code—a programmer should be able to lay her hands on the bare metal of the hardware but then treat specific implementations as the high-level concepts they represent.

An interesting class of features is related to the convenience of generic programming: both languages provide such features (Bernardy et al., 2008, table 2) but each in a different way, matching other language features and a general philosophy. Both languages allow default definitions, Haskell offers structure-derived modeling and model lifting, and C++ can generate modelings from refinements, implicit definitions, and by automatic modeling. Default definitions in a concept can be provided in terms of other entities available in the concept and the surrounding scope. For example, in C++:

```
concept Comparable<typename T> {
  bool better(T, T) { return true; };
}
```

The above example makes everything `better` than everything else, but forgoing correctness, a modeling does not have to give an implementation for `better` anymore. In Haskell, which is based on algebraic data types, it is often possible to build a modeling by inspection of the structure of a datatype. A structure-derived modeling for a boolean type can be declared as follows:

```
data Bool = False | True
  deriving Eq
```

`Eq` names an equality comparable concept, and equality can be trivially derived for algebraic data types such as `Bool`. Structure-derived modeling is limited, however, to compiler-supported concepts. In C++, such derivations may be neither possible nor meaningful, but it may be possible to treat a C++ type in a structural way using *interface traversal* of type structures (Paper V). Haskell also provides special support for type wrappers, a commonly used language feature to give a type a different name without generating a different representation. A short example demonstrates model lifting for wrappers:

```
newtype Age = Age Int
  deriving (Hashable)
```

Here, a wrapper `Age` is created for `Int` erasing all modelings that `Int` might have. Yet, assuming `Int` models a concept `Hashable`, it may be desired that `Age` models `Hashable` too, without getting other models of `Int`. C++ convenience features are based on the assumption that concepts may have large interfaces and form complex refinement hierarchies. Furthermore, C++ separates names in concepts from surrounding scope so name matching can be used to discover implementations (inference in Haskell removes this possibility, see the previous discussion). The following example illustrates modeling from refinement and implicit definitions:

```
concept A<typename T> { typename foo; }
concept Plus<typename T> : A<T> { T operator+(T, T); }
```

```
3 concept_map Plus<int> { typedef int foo; }
```

The concept map definition above does not provide an implementation for `operator+` but the built-in one is found for `int` and is implicitly used by the concept map. The concept map for `Plus` also generates a concept map for `A` using the definition given for `foo`. An automatic concept, marked by the keyword `auto`, tells the compiler that an explicit concept map defined by a user is not necessary at all, and that generation of concept maps should be attempted on demand, when one is needed. The two sets of features can be considered for porting from one language to the other (Bernardy et al., 2008), but quite clearly, they each are a close fit for the context in which they are used.

An interesting difference, the only one remaining, judging by the criteria in table 1.1, is separate compilation. Whether separate compilation is needed or not depends very much on the goal of a language. Separately compiled code makes for better program build times and possibly smaller code, and it makes it possible to withhold source code from library users. On the other hand, separate compilation incurs the cost of run-time indirection and it makes instance-specific optimization difficult. Motivated much by their respective design philosophy, Haskell and C++ make different choices for supporting separate compilation. In C++, the overriding motivation is efficiency, assuming that generic programming will only be used if it gives the efficiency C++ promises in the first place. Efficiency has such priority that even separate type checking safety in some caveat cases is given up for the sake of optimization (Gregor, 2008c; Järvi et al., 2006). In Haskell, on the other hand, the convenience of separate compilation trumps some optimization.

In conclusion, we note that Haskell and C++ are probably the two widely-used languages in which generic programming paradigm has gained a strong foothold. Given their different origins, it is surprising how close the languages came together and how much interest there is in the cross-pollination of ideas.

5 Specification and Programming with Concepts

Concepts bind abstract ideas with the practice of programming. On the one hand, concepts are mathematical entities described by high-level mathematical laws, and, on the other hand, they are a practical programming device that harnesses the power of the high-level, abstract description, enabling better code organization and reuse. Consequently, concepts synergistically combine *algebraic specification* and programming language design, where they enable *constrained polymorphism*—in this section we provide a brief discussion of these two fields, as related to concepts.

5.1 Algebraic Specification

The basic premise behind algebraic specification (Wirsing, 1990) is that software systems can be viewed as sets of data with some operations on them (Tarlecki, 2003). Traditionally, the field of algebraic specification was dominated by equational specification and initial algebra semantics, perhaps first explicitly introduced by the ADJ group (Goguen et al., 1978). Since then, algebraic specification became more flexible, there are many different approaches to semantics, and closer links to the actual software; especially, *structuring* of specification has been researched to help manage the complexity of specifying real software.

5.1.1 Abstraction

The goal of algebraic specification is to represent properties of software independently of its implementation details. The subject "abstract" algebra is the structure; for instance, in group theory, an "abstract group" is an isomorphism class of groups, independently of the details of each concrete group. In essence, two algebras have "same structure" if, and only if, they are *isomorphic* (Goguen et al., 1978).

Software, or data types in particular, can be *described* by such abstract algebras. An algebraic description specifies the structure one can expect from a data type independently of any implementation details. Such description then specifies an *abstract data type*.

5.1.2 Basic Algebraic Specification

The basic algebraic specification relies on the following rough intuitive analogy (Tarlecki, 2003)[2]:

module	$\rightsquigarrow$	algebra
module interface	$\rightsquigarrow$	algebra signature
module specification	$\rightsquigarrow$	class of algebras

Here, *module*, *module interface*, and *module specification* represent software artifacts and *algebra*, *algebra signature*, and *class of algebras* represent the algebraic specification of the software artifacts. The definition of software artifacts and their exact correspondence to the algebraic concepts is specific to a particular situation; in case of C++, module roughly corresponds to one or more types, module interface to the syntactic requirements of a concept, and module specification to the semantic requirements of a concept, of which some may be explicitly stated as axioms and some only informally described in documentation. As for the abstract representation, universal algebra (Grätzer, 1979) has been studied for many years independently of its computer science applications, where *many-sorted* or *heterogeneous* (Birkhoff and Lipson, 1970) algebra is naturally applicable. The formal definition of the correspondence between the actual software and its algebraic description is difficult to define, most often it is only informal.

An *algebraic (many-sorted) signature* $\Sigma = (S, \Omega)$ consists of a set S of *sort names* and of a set $\Omega = \langle \Omega_{w,s} \rangle_{w \in S^*, s \in S}$ of *operation names*, indexed by their *input* and *result sorts*. A Σ-algebra $A = (|A|, \langle f_A \rangle_{f \in \Omega})$ consists of a family $|A| = \langle |A|_A \rangle_{s \in S}$ of *carrier sets* for each sort $s \in S$ and of *operations* $f_A \colon |A|_{s_1} \times \ldots \times |A|_{s_n} \to |A|_s$, one for each operation name $f \in \Omega_{s_1 \ldots s_n, s}$; in other words, an operation is a function that maps a tuple of elements from appropriate carrier sets to an element from the carrier set of the result sort. The class of all Σ-algebras is denoted $Alg(\Sigma)$.

Without much more detail, properties of Σ-algebras may be captured by *equations*, where the sides of equations are constructed by applying (possibly repeatedly) the operations of a signature to *variables*, which each are given a sort; a class of all equations that can be formed for a given signature Σ is denoted $Eq(\Sigma)$. Then, given a signature Σ and a set of equations $\phi \in Eq(\Sigma)$, there is a class of algebras that *satisfy* these equations, called *models* of ϕ, denoted by $Mod(\phi) = \{A \in Alg(\Sigma) \mid A \models \phi\}$.

In the traditional study of universal algebra (and logic), the signature is most often fixed albeit arbitrary. In computer science, the ability to change a signature (add,

[2] Sections 5.1.2 and 5.1.3 are a short summary of the overview of abstract specification theory by Tarlecki (2003).

change, remove, or glue signature components) is much more important: it allows one to build more complex specifications from simpler ones and to "evolve" the specification with changing requirements. This capability is introduced by the notion of signature morphisms. Given two signatures, $\Sigma = (S, \Omega)$ and $\Sigma' = (S', \Omega')$, a *signature morphism* $\sigma : \Sigma \to \Sigma'$ maps the sort and operation names of one signature to another, preserving signature profiles (input sorts and the result sort) according to the sort mapping. Stated simply, a signature morphism determines, in a natural way, the translation of "Σ-syntax" to "Σ'-syntax." This translation includes *translation of variables*, *translation of terms*, and *translation of equations*. Furthermore, a signature morphism $\sigma : \Sigma \to \Sigma'$ determines a translation of "Σ'-semantics" to "Σ-semantics" (note the contravariance), called *sigma reduct*, which translates Σ'-algebras to Σ-algebras. The syntax and semantics translations are linked by the *satisfaction property*, which requires that satisfaction, or "truth", is preserved under the change of notation (names of the operations and sorts) and restriction or extension of irrelevant context.

We have given only a basic introduction of the algebraic constructs used to specify software. Our goal is to briefly expose the reader to the representation syntax and semantics used in algebraic specification. Also, it is important to note once again the need for the semantic link between actual software and algebras. This link is often the most complex part of algebraic specification and for that reason it is usually specified only informally.

5.1.3 Algebraic Specification with Institutions

The traditional approach to algebraic specification has proven too limited as real specification languages have been developed. Some of the parameters of algebraic specification systems that vary depending on the needs include:

- the formulae (Horn clauses, first-order, higher-order, modal formulae, ...),
- the notion of algebra (partial algebras, relational structures, error algebras, Kripke structures, ...),
- the notion of signature (order-sorted, error, higher-order signatures, sets of propositional variables, ...),
- the notion of signature morphisms, with different translations of formulae and model reducts.

Varying the notions results in a "population explosion" of logical systems used in software specification. To deal with the numerous approaches, Goguen and Burstall (1992) introduced the notion of *institutions*. The notion of institution axiomatizes logic in a categorical (MacLane, 1971) setting. While abstracting from details, the categorical formulation is still sufficient to develop basic semantics of a software specification system. Institutions are discussed more in depth in Paper III; for the current discussion, it suffices to say that an institution provides signatures, models, and signature morphisms with appropriate model reducts that fulfill the satisfaction condition. Additional constraints may be added to institutions to "strengthen" the class of logics they represent (narrowing the class of logics that can be expressed), allowing stronger and richer semantics for a specification system while retaining a degree of independence from the details of a particular logic.

5.1.4 Examples of Applications

Many algebraic specification systems have been designed. Examples include the Tecton system mentioned earlier in this introduction or the modern CASL system (Bidoit and Mosses, 2004; Mosses, 2004). Tecton relies on the traditional notions of algebras, while CASL is formalized around the notion of institutions; consequently, the logic of Tecton is hardwired but the semantics of CASL is divided into the generally applicable part and the CASL specific logic. Thanks to that division, much of CASL development can be reused with the logic changed "under the hood."

Another kind of algebraic specification applications is in actual software construction. Kahrs et al. (1997) developed an extended version of the ML (Milner et al., 1990a) language. The extension allows specification of modules directly in the language, without providing implementation in the initial stages of development. Then the specification can be gradually refined into an executable program and its correctness can be proven against the original specification. A further example of practical use of algebraic specification is the LILEANNA language developed by Tracz (1993) based on the ideas developed earlier by Goguen (1984). LILEANNA is a *library interconnect language* (LIL) that allows combination of modules into more complex modules based on their specification.

In Paper III we outline how concepts can be treated as an algebraic specification tool. The benefits of such treatment are twofold. First, the semantic constraints that concepts can currently impose can be extended from the simple equational specification to different logics. Second, the institutional formalization makes it possible to reuse the mechanisms for structuring developed in the context of algebraic specification languages, similarly to the way in which these concepts are used in LILEANNA.

5.2 Constrained Polymorphism

The kind of generic programming discussed in this thesis relies on *polymorphic types*. A polymorphic type, which can describe functions or data structures, is parameterized by other types that can be supplied later. In general, a polymorphic type is of a form $\forall t \Rightarrow f(t)$, where t is a type variable and f possibly depends on t. Many variations are possible; for instance, most modern languages, including C++, allow more than one parameter. Yet, for the purpose of a general discussion the simple form illustrates polymorphism well.

Given a polymorphic type, the important question is what can be "done" with the type parameter. Since a type of the form $\forall t \Rightarrow f(t)$ is quantified for all types, the only operations that can be done are the ones that are possible over all types, namely nothing. In a functional language, a parametric list type can be expressed as follows:

```
1 data List a = Nil | Cons a (List a)
```

The `List` type has one parameter, `a`, and two constructors, `Nil` and `Cons`, signifying an empty list and a pair of an element and a list. In such definition, nothing is expected of the parameter, it is simply stored in the list. Cardelli and Wegner (1985) call such polymorphism *universal*, since every argument is treated in the same way: there are no "special" cases.

Constrained polymorphism limits which types can be used as arguments to a polymorphic type. In a simple formulation, an unconstrained polymorphic type can be thought to represent the set of types

$$\{f(\tau) \mid \tau \in Type\},$$

where $f(\tau)$ is obtained by substituting τ for t in $f(t)$ and *Type* represents all non-quantified types. A constrained polymorphic type, or qualified type (Jones, 1994), has the form $\forall t.\pi(t) \Rightarrow f(t)$ and it represents the set of types

$$\{f(\tau) \mid \tau \text{ is a type such that } \pi(\tau) \text{ holds}\}.$$

C++ concepts are used to constrain template parameters; in that sense a C++ template can be considered to have the following form (Järvi et al., 2003):

$$\forall \overrightarrow{t}.c_1(\overrightarrow{t_1}) \wedge \ldots \wedge c_n(\overrightarrow{t_n}) \Rightarrow f(\overrightarrow{t}),$$

where $\overrightarrow{t}$ corresponds to the list of template parameters, c_i signifies concepts, and $\overrightarrow{t_i} \subseteq \overrightarrow{t}$ signifies lists of parameters constrained by each concept. This constrained polymorphic type represents the set of types

$$\{f(\overrightarrow{\tau}) \mid \overrightarrow{\tau_1} \text{ models concept } c_1, \ldots, \overrightarrow{\tau_n} \text{ models concept } c_n\}.$$

In difference to other forms of constrained polymorphism, such as the subtype-bounded polymorphism of Cardelli and Wegner (1985), concepts not only constrain the types that can be used as arguments to a polymorphic type but they also introduce operations and associated types that can be used with type parameters. As discussed in section 3, the polymorphic type is checked against the syntactic signature provided in concepts, and the modeling relation is established in concept maps, where the syntax of concepts is assigned particular implementations for particular types. In C++ without concepts, the polymorphism of templates is seemingly unconstrained. Yet, when a template is *instantiated* with particular types, these types must provide the operations and the associated types used in the template—in a sense, although concepts are implicit, the polymorphism in C++ without concepts in the end is not really universal.

In this thesis, we formalize constrained polymorphism in C++ with concepts (Paper I) and consider how the similar mechanism of Haskell type classes (Wadler and Blott, 1989) can be translated to C++ concepts.

6 Concept Applications

The Standard Template Library (STL), discussed earlier in this chapter, was the first C++ library based on a conceptual hierarchy. In the current version of C++ (ISO, 2003b), STL comprises a set of class and function templates, where template parameters are *required* to model concepts. Concepts are expressed as documentation (see figure 1.1) and the modeling relation is not checked by the compiler. STL has inspired other generic libraries. Some examples include the Boost Graph Library (BGL) (Siek et al., 2002), Boost Metaprogramming Library (MPL) (Abrahams and Gurtovoy, 2004), and Boost Multi-Array library (Garcia and Lumsdaine, 2005); the BGL and Multi-Array libraries apply concepts in a way similar to STL, while the MPL applies concepts at the metaprogramming level: in MPL, concepts require compile-time operations in contrast to the run-time operations required by the STL concepts.

Concepts have been applied in a variety of ways in the context of C++. Wang and Musser (1997) propose *dynamic verification* for generic C++ programs. They represent concepts as *run-time analysis oracles*, which represent the syntactic part of a concept, and *inference engines*, which represent the semantics of concepts. Using this representation of concepts, Wang and Musser are able to verify generic algorithms once for any

template arguments that model appropriate concepts. Schupp et al. (2001) propose *optimizer generators* where simplification rules are *concept-based*, that is simplification rules are given for groups of types that model a given concept in a type-independent manner. Haveraaen (2007) proposes using concepts for test generation, where concepts are considered as a form of software specification (with conditional equations). The specifications attached to concepts are used for automatic test generation. Haveraaen's institution is similar to our formalization in Paper III but while Haveraaen considers concept maps to be signature morphisms, in our formalization concept maps represent sentences (i.e., the statement of semantics). In another paper (Paper II), we propose a *change impact analysis* for generic libraries. The basic premise of our work is that a generic library is defined by its generic interface, which consists of a conceptual hierarchy that describes the problem domain that the library operates on and the signatures (and specifications) of generic algorithms. Our analysis helps to determine the impact of changes on the conceptual level of library.

Recently, new concept applications have been enabled (and inspired) by the linguistic support for concepts in C++0x (Gregor, 2008a; Gregor et al., 2008e). For example, much of the standard library of C++ has been conceptualized, that is, rewritten using the new concepts feature (Gregor and Lumsdaine, 2008a,b,c; Gregor et al., 2008a,b,c,d). This body of code and specifications provides a good example of how generic libraries will be written when concepts officially become a feature of C++. Järvi et al. (2007) have proposed a library composition technique based on C++ concepts. In their work, libraries are adjusted to a common interface using concept maps, and, furthermore, libraries based on run-time polymorphism are mixed with libraries that rely on static polymorphism through concepts. (Pirkelbauer et al., 2008) apply a similar idea in the context of STL. The last example we cite is the concept-based optimization framework described by Tang and Järvi (2007). The framework relies on the feature of *axioms* introduced by concepts and applies high-level optimizations based on semantic properties of concepts, similar to the optimization framework by Schupp et al. (2001). The framework of Tang and Järvi, however, depends only on built-in C++0x features and does not require any extensions to the language.

The kind of applications as the ones enumerated in this section demand increased support for concepts, both for the direct users of concepts and for developers of future tools that operate on generic software. In the next section, we describe how the work presented in this thesis supports generic programming with concepts.

7 Supporting Concepts

Concepts have been part of the generic programming practice in C++ at least since introduction of the STL. Surprisingly, despite that tradition, there is little tool support for concepts. Most effort, up to date, has been extended into *concept checking*; for instance, Siek and Lumsdaine (2000) developed a library that improves error messages triggered by mistakes in the use of generic libraries. The introduction of concepts into the C++ language (Gregor et al., 2006b, 2008f) is the pinnacle of these developments. One of the main contributions of this thesis is the support for the development of generic software using concepts. Our contributions fall roughly into two categories: the practically-oriented software analysis and transformation techniques and the more foundational formalizations of concepts, meant to support construction of concept-related tools in general.

7.1 Formalization

In section 2.3, we discuss the important formalizations of concepts that influenced the tradition of generic programming in C++. Our work presented in this thesis complements the previous developments. Specifically, in Paper I (along with Paper VII) we formalize the semantics of *separate type checking* with concepts as specified by Gregor et al. (2008e), and in Paper III we define concepts as an institution (see section 5.1).

While the previous formalizations develop certain aspects of concepts formally, the actual definition of C++ concepts given by Gregor et al. (2008e) is in the form of a language manual, written in the style of the C++ standard (ISO, 2003b). As such, the concept wording specifies the *grammar* of concepts, in a BNF-like format, and the *semantics* of concepts. Specifying semantics is a difficult task and the semantics of concepts is given in natural language as the rest of the C++ standard. It is clear that such statement of semantics is prone to ambiguity, omissions, and contradictions (see Miller (2008) for a list of issues in the current C++ standard) but giving precise semantics is considered to be too expensive; indeed most of the popular languages do not have precise semantics.

Our formalization of concepts does not attempt to capture the complete interaction between concepts and other C++ constructs. Instead, we concentrate on the aspects of concepts that differ from the other mechanisms in C++; in particular, we define *separate type checking* with concepts, specifying the semantics of *name lookup*, *concept maps*, and *constrained templates*. In the remainder of this subsection, we introduce the basics of formal semantics of programming languages.

7.1.1 Formal Semantics

Formal semantics can be given in different ways, for different parts of the language, and at different levels of abstraction. The basic division can be made based on how the "meaning" of a program is defined (Nielson and Nielson, 1992; Schmidt, 1986):

- *Operational semantics* can be compared to an interpreter—a program is understood as a history of some interpreter states and the transitions that were made between these states.
- *Denotational semantics* maps a program directly to its meaning, called denotation (usually a mathematical function). The mapping is called a *valuation function*.
- *Axiomatic semantics* captures specific properties of the programming language as *assertions* providing a *logical system* for proving correctness of programs.

Of course, each method of semantic description has strengths and weaknesses and the methods complete each other (Schmidt, 1986): one could start with an axiomatic description to state the most important properties of a language, then move to the denotational semantics to understand and reason about the meaning of the language, and finally use operational semantics as a model for implementation and, for example, to prove type safety of the language.

Semantics can be further divided into *dynamic semantics* and *static semantics*. In general, dynamic semantics describes the behavior of a program at runtime and static semantics describes the rest, although the distinction is not always clearcut. Static semantics describes the context-sensitive parts of semantics (such as type checking, which may depend on the context of previous declarations, for example), traditionally

checked at compile time, such as type checking, scope resolution, and storage calculation; dynamic semantics describes the evaluation of a program in which the actual "meaning" is produced (Plotkin, 1981; Schmidt, 1986). Concept checking in C++ is part of the static semantics—it is performed as part of the type checking process. In difference to unconstrained templates in C++, concept-constrained templates are checked only *once*, in the context in which they are defined; an unconstrained template is also checked in the context in which it is used, where the symbols missing in the declaration context are looked up in the surrounding scope and in the scope in which template arguments are defined (in C++, this is termed argument-dependent lookup).

We formalize the semantics of concepts in the operational style, specifically in the *structural operational style* (Plotkin, 1981). A structural operational semantics is an unordered collection of rules of the form:

$$\frac{A_1, \cdots, A_m}{B},$$

where the numerator, $A_1, \cdots, A_m$, constitutes the (unordered set of) *premises* of the rule and the denominator, B, is the conclusion of the rule (Kahn, 1987; Plotkin, 1981). Intuitively, if all the formulae in the numerator hold, the formula in the denominator holds. The rules inductively define semantic relationships that establish the meaning of the program. Examples of such relationships include (examples from Kahn, 1987):

- A typing relationship $\rho \vdash E : \tau$, where an expression E has a type τ in the environment ρ.
- A translation relationship $\rho \vdash E_1 \rightarrow E_2$ that translates an expression E_1 in the language L_1 to an expression E_2 in the language L_2, where the environment ρ records the assumptions on the identifiers.
- A simple evaluation relationship $s_1 \vdash E \Rightarrow s_2$, which, given a state s_1 and an expression E, produces the new state s_2.

In our work, the crucial relationship, *program instantiation*, has the form $C; M; T \vdash P \Downarrow C'; M'; T'$; it relates a program P and an environment $C; M; T$ to a new environment $C'; M'; T'$.

Kahn (1987) compares the semantics of a programming language to a logic and reasoning about the language to theorem proving. In that sense, a semantic definition provides axioms and inference rules that characterize the various semantic predicates involved in that definition. Furthermore, inferences are broken down into steps in such a way that each step involves only one language construct (Prawitz, 1971). For example, the rule in figure 2.5 (page 44) describes how a template is processed:

$$\frac{1.\ tl = \texttt{template}\,\texttt{requires}\,\overline{cid_i}^{\,i}\ tn\,\{e\} \qquad 2.\ \overline{cid_i \text{ defined in } C}^{\,i} \qquad 3.\ tl \text{ not defined in } T \qquad 4.\ C; \emptyset; \overline{cid_i}^{\,i} \vdash e : \tau \qquad 5.\ C; M; tl\,T \vdash P \Downarrow C'; M'; T'}{C; M; T \vdash tl\,P \Downarrow C'; M'; T'}$$

In Kahn's terminology, the conclusion of the rule is a *sequent*, while the premises are either conditions, simple formulae conveying a general restriction on the applicability of a rule, or themselves sequents, the predicates of the semantics. The premises of the rule quoted above comprise 2 simple formulae (premises 1 and 3) and 3 sequents (premises 2, 4, and 5). The rule defines processing of templates: a program P that begins with a template tl (the program $tl\,P$) results in updated context $C'; M'; T'$ given the initial context $C; M; T$ and assuming that all the preconditions in the numerator of

the rule hold. This style of giving semantic is *structural* because it is syntax-directed; rules give program behavior structurally, or primitive recursively, in the syntax (the idea of structural recursion is due to Burstall, 1969; Plotkin, 2004).

Finally, our semantics is given in the *big-step* style or, following Kahn, natural style. Natural semantics shows how *overall* results are obtained, while structural operational semantics [3] shows the individual steps taken to obtain the result. The difference may be illustrated by a simple example of statement processing: in natural semantics a statement and a state produce a new state, $\langle S, s\rangle \Rightarrow s'$, and in structural operational semantics a statement and a state can produce a simpler, new statement and a new state or a final state for the simplest statements, $\langle < S, s > \rangle \Rightarrow \gamma$, where γ is either $\langle S', s'\rangle$ or a new state s. We refer the reader to Nielson and Nielson (1992) for a summary of differences between the two styles, and here we just conjecture that natural style is more readable for static semantics such as separate type checking with concepts.

7.2 Software Analysis and Transformation

In very broad terms, software analysis is the process of discovering some properties of software and software transformation is changing software in a way that preserves some properties while modifying others. We consciously use the term *software analysis* rather than the term *program analysis* (Nielson et al., 2004), which usually encompasses the "traditional" analysis, such as common subexpression analysis or reaching definitions analysis. *Software transformation* is as broad if not even a broader term; the program transformation wiki[4] enumerates program transformation, re-engineering, software evolution, and software architecture as some of the possible "entry points" into the field of software transformation.

In this thesis, we consider two particular problems that fall into the categories of software analysis and software transformation, specifically in the context of generic libraries. The first problem (Paper II) is that of understanding the impact of changes made to a generic library. In generic libraries, changes can occur at the concrete level of an implementation or at the abstract level of the concept taxonomy and the generic algorithms. We address the issue of change impact in our *conceptual* change impact analysis (CCIA); however, analyzing the impact of changes at the concept level of generic libraries still leaves much to explore.

Impact of the changes at the level of implementation has been a topic of previous research and may be classified according to the sources of change. Of particular interest to such analysis are non-local changes (Bohner and Arnold, 1996a,b), in imperative programming introduced, for example, through side effects on global variables, in object-oriented programming, for example, through modification of the inheritance tree and dynamic method dispatch, and in aspect-oriented programming through renaming or moving of methods. As it holds for most change-impact analyses (e.g., Bohner, 2002; Fyson and Boldyreff, 1998; Stoerzer and Graf, 2005; Tonella, 2003; Zhao, 2002), the main goal of our CCIA is to support software evolution, in our case the evolution of libraries. Yet many other applications of change-impact analysis exist, for example, for efficient regression testing (e.g., Orso et al., 2003; Ren et al., 2004, 2005). For a discussion of related work, see section 2 in Paper II.

In Paper IV, we consider multi-language library development, where a generic library is prototyped in Haskell and implemented in C++. The implementation of the

[3] Both styles are structural in the sense that they are syntax-directed, the names are derived from the titles of original publications: Kahn (1987) and Plotkin (1981).

[4] `www.program-transformation.org`

library in C++ may be quite different but its conceptual structure is usually very similar. To facilitate the multi-language development, we propose a translation of Haskell type classes and generic algorithms to their C++ counterparts.

8 Summary and Contributions

All papers in this thesis have generic programming as the common topic, all are also specifically about generic programming in C++, in one way or another. Generic programming in C++ relies on concepts and every paper, save Paper V, is about generic programming with concepts. Still, even Paper V is about generic programming in C++ but it treats *datatype* generic techniques, ported from the functional language Haskell.

The main contribution in Paper I is the formalization of the semantics of concepts, which up until now was only specified in natural language. Our formalization is useful in the analysis of the current concept design put in front of the C++ standardization committee. A formal definition of concept semantics, such as ours, is an aid in understanding and discussing of generic programming with concepts, and can serve as the specification for implementing compilers. Furthermore, we give an informal introduction to concepts and we illustrate separate type checking with several examples. Paper VI provides full details of the semantic definition discussed in Paper I.

There are two main contributions in Paper II. First, we present a change impact analysis for generic libraries. Second, we apply the analysis to a proposed change to STL iterator concepts. Our analysis differs from previous change impact analyses in the subject of change and the nature of impact. CCIA operates on the level of conceptual specifications of a generic library, that is, the concept taxonomy and the generic algorithms, while most of the previous analyses operate on the level of low-level program constructs. Our case study is a separate contribution. The changes proposed to STL iterators will have an impact on many users; our analysis demonstrates that, in fact, the changes have some unintended impact.

In Paper III, we connect C++ concepts with the field of generic specification by defining the *institution* of concepts. This connection makes it possible to apply methods developed in the field of algebraic specification to concepts. Furthermore, such institution makes it possible to consider an algebraic specification system where models are real programs rather than algebras. In most algebraic specification systems, a connection between specifications and real programs is not made, but our research opens up such possibility.

Paper IV develops a translation from generic interfaces in Haskell to the corresponding generic interfaces in C++. Such translation uncovers interesting similarities and differences between Haskell's type classes and C++'s concepts. Furthermore, the translation has a practical effect of multi-paradigm development, where a program is first prototyped in Haskell and then implemented in an "industrial-strength" version in C++—this development model is supported by a concrete example.

In Paper V, we develop a "scrap your boilerplate" (Lämmel and Peyton Jones, 2003) generic programming technique for C++. Since C++ types do not have an accessible "shape," as types in Haskell do, we introduce a shape through metaprogramming means. We then propose a generic traversal technique.

Statement of personal contribution The technical contributions in Paper I, Paper II, and Paper VI are mine, while the contributions in Paper III, Paper IV, and Paper V are shared with my coauthors. I have substantially contributed to writing of Paper III and

Paper IV, while paper Paper V was mostly written by my coauthors. The papers coauthored with my supervisor, Sibylle Schupp, are written by me with Sibylle's significant input into both research and writing.

9 Outlook

We consider three main directions of further research. The first possible direction is extending the formalization presented in Papers I and VI with additional language constructs, such as instantiation of generic algorithms. In addition to language constructs directly related to concepts, the formalization could benefit from including other, basic C++ constructs, such as references, and limited object-oriented part of C++—these features interact with concepts in sometimes unexpected ways. Furthermore, our semantic definition could be extended to a *language prototype*, using a rewriting tool such as PLT Redex (Matthews et al., 2004) or the Maude system (Clavel et al., 2002). A "semantic prototype" of the concept-constrained templates would be easier to modify and more transparent than the ConceptGCC compiler (Gregor, 2008a). A second direction of possible future research is extending the concept system itself. A particular issue with concepts is visibility of names and correct propagation of implementations. We have considered an extension of the notion of scopes, *visibility regions*, that would facilitate easier reasoning about names in concepts. A concrete problem that could be addressed is *sibling-to-sibling* propagation of implementations, which is not possible with concepts. Finally, we consider further developing the notion of concepts as institutions and applying algebraic specification techniques to C++. As the first step, our formalization of concepts can be combined with the idea of concepts as institutions: the formalization can serve as a basis for properly formalizing the satisfaction relation between concepts and their models. Furthermore, we plan to investigate applying algebraic specification-building operations (Sannella and Tarlecki, 1988) in the context of C++ concepts. Such operations would allow programmers to create new concept hierarchies and implementations using concept expressions, similar to module expressions of LILEANNA (Tracz, 1993).

Paper I

A Semantic Definition of Separate Type Checking in C++ with Concepts

This paper is extended by a correctness argument for separate type checking safety (theorem 1) and by examples of the application of the semantics.

A Semantic Definition of Separate Type Checking in C++ with Concepts

Marcin Zalewski and Sibylle Schupp

Abstract

We formalize the informal definition of C++ *concepts* that is currently considered by the C++ standardization committee for inclusion in the next version of the language. Our definition captures the basic semantics of separate type checking, where *concept-constrained templates* are checked separately from their uses and comprises of three main parts: a non-standard name lookup, type checking of constrained templates, and *implementation binding* in concept maps. The formalization reveals two possible problems in the informal definition: hiding of names is not respected and incompatible implementations can be bound to concept entities. Furthermore, our definition allows formulating intuitively correct code that is rejected by the informal specification.

1 Introduction

Generic libraries in C++ have long relied on *templates*, pieces of code parameterized by types and values, to implement reusable data structures and algorithms. Such generic code can be applied to families of types; in C++, application of generic code requires *instantiation* of templates at compile time. While the families of types to which generic libraries can be applied are infinite, types in each family share the characteristics specific to the domain *concept* that these types represent.

A concept, then, can be considered, on the one hand, as a predicate on types, and, on the other hand, as a specification of an abstract data type. Indeed, concepts have long been used in C++ generic libraries, beginning with the Standard Template Library (Musser et al., 2001), to document the syntax and semantics required from template parameters. True to the specification aspect of concepts, the origins of this tradition can be traced back to the algebraic specification language Tecton (Kapur and Musser, 1992). Since concepts have become an integral part of generic programming in C++, direct support for concepts in the C++ language has been proposed by several authors (Gregor et al., 2006b; Dos Reis and Stroustrup, 2006). Concepts are now de facto accepted (by the C++ ISO standardization committee (Meredith, 2008)) into C++0x, the next revision of C++, which is expected to be completed by the end of this decade. As a language construct, concepts allow for *separate type checking* where polymorphic, generic code (templates) and concrete types are checked separately, against the contract represented by concepts. Then, at the point of *template instantiation*, when the concrete types are substituted for template parameters, a compiler only has to verify that the contract was checked earlier; if it was, the template is guaranteed to work correctly (the guarantee may be weakened to permit optimizations, see (Järvi et al., 2006)). The definition of concepts for C++0x is given in natural language (Gregor et al., 2008e), in the style of the C++ standard. The wording of the definition takes about 80 pages alone, and the complete C++ language definition spans several hundreds of pages.

Our main contribution is a formal definition of separate type checking with concepts, with semantic judgments defined by inductive relations over the abstract syntax of a language that captures the core functionality of concepts. Much of our semantic definition treats the interaction of name lookup and implementation binding with *concept refinement*, a mechanism that facilitates construction of concept hierarchies. Name lookup in concepts differs from the usual lookup rules in C++; for example, function

overload sets, which are typically limited to the scope of one syntactic entity, can span multiple concepts in a concept hierarchy. The implementation binding mechanism is entirely new to C++ in that implementations are entirely separated from the types for which they are defined. Furthermore, implementation binding may be non-local—the refinement relation propagates implementations automatically, creating *implicit concept maps*. Every concept map introduced implicitly must be *compatible* with the concept maps introduced previously; this condition is quite difficult to define and is different from other mechanisms of C++. Our formalized account of these features enables one to clearly understand all intricacies of the design, disambiguating the natural language wording and, also, making hidden design choices visible. In addition, our semantic rules may serve as a starting point for a compiler implementer and provide a basis for a future, formal separate type checking safety proof.

The premise of separate type checking with concepts is that once constrained templates are checked against their *requirements* and concept maps are well-formed, templates can be safely instantiated *without* any additional type checking, provided that all necessary concept maps have been defined. The checking of templates against constraints and the binding of implementations in concept maps is captured, in our formalization, by the *program instantiation* judgment, which relates a program to an environment if and only if the program is correct. This judgment is the main premise of an extended version of the separate type checking safety theorem proposed by Dos Reis and Stroustrup (2006) as a future work task for the C++ concept system. Program instantiation and the safety theorem are described in detail in section 4, examples of application of the semantics and an argument for correctness of the theorem are given in section 5. The complete set of semantic judgments is available in the accompanying technical report (Paper VI). We introduce C++ concepts informally in section 2, supporting the discussion with code examples.

1.1 Notation

The semantics of concepts given in this paper is based on the latest version of the concept wording at the time of writing (Gregor et al., 2008e) (available online at `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/`), hereon referred to simply as "the wording." Moreover, we refer to parts of the wording by its sections and paragraphs, for instance, "clause 3.3.10 para. 1" for the first paragraph of section 3.3.10. The C++0x language extended by the wording is referred to as ConceptC++ and the current version of the language is simply C++. The language used for our formalization is referred to as X++.

2 Concepts, Name Lookup, and Implementation Binding

In C++, a template is either a class or a function parameterized by types (also by values but we do not discuss that here). When a template is defined, the compiler postpones all but syntactic correctness checks for the parts of the template that depend on type parameters. At instantiation time, the template is type-checked again with type arguments substituted for parameters. Since full type checking does not occur until instantiation time, the developers and the users of generic libraries must agree on the semantics and the syntax that template arguments must provide to guarantee the correct operation of generic code when applied to these arguments. In the current C++ practice the

agreement is described in the documentation of a library where *syntactic* and *semantic* requirements of each template are listed. Concepts for C++ provide direct support for expressing and checking of requirements of templates within the language. Simply, concepts enable *separate type checking* of the generic template code and of the particular types to which the generic code can be applied.

In the remainder of this section, we provide an informal introduction to separate type checking, including the details of refinement, name lookup, and implementation binding. The discussion is supported by code examples that comply with the language specification in the concept wording (Gregor et al., 2008e).

2.1 Introduction to Separate Type Checking

```
1  struct NotNumber {
2    typedef NotNumber result_type;
3    result_type operator+(NotNumber) { return result_type(); }
4  } a, b;           // declare a and b of type NotNumber
5  concept Addable<typename T> {
6    typename result_type;
7    result_type operator+(T, T);
8  }
9  concept_map Addable<int> {
10   typedef int result_type;
11   result_type operator+(int a, int b) { return a+b; }
12 }
13
14 template<typename T>
15 typename T::result_type plus(T a, T b) { return a+b; }
16
17 template<typename T> requires Addable<T>
18 result_type cplus(T a, T b) { return a+b; }
19
20 int main() {
21   plus(1, 2);     // error, int has no result_type
22   plus(a, b);
23   cplus(a, b);    // error, no Addable<NotNumber> concept map
24   cplus(1, 2);
25 }
```

Listing 2.1: Type checking of constrained and unconstrained templates

Listing 2.1 shows a simple example, illustrating the basics of separate type checking. Two template functions, adding two values of an addable type, are defined, one is a usual, *unconstrained* C++ template (lines 14–15) and the other is a *constrained* template with a `requires` clause (lines 17–18). The unconstrained template returns a value of type `T::result_type` and, consequently, requires that types with which it is used have a member type `result_type`. Since nothing is known about the type parameter `T` and its members, the `typename` keyword must be placed before the name `T::result_type` to explicitly inform the compiler that the name will designate a member type. Then, two values `a` and `b` are added and the result is returned. At this point, a compiler cannot check that the addition produces a result of the expected type `T::result_type` because the definition of the addition operation depends on template arguments and can only be looked up when the template is instantiated. The constrained template is very similar but it requires that its arguments model the `Addable` concept. The concept, defined on lines 5–8, has two requirements, one for an associated type `result_type` and one for an associated function `operator+`, which takes two arguments of the addable type

`T`. The constrained template is checked at the time of its definition, against the syntax introduced in the `Addable` concept. Since `Addable` provides the associated type `result_type` and the associated `operator+` that returns this type, the template type-checks. In difference to the unconstrained template, `result_type` does not have to be preceeded with the `typename` keyword or qualified by the parameter `T` because it can be looked up in the `Addable` concept.

When the unconstrained template is called on lines 21–22, the types of the arguments are substituted for parameters and the template is type-checked. In the first call, the template called with two integers fails to instantiate, because `int`, a built-in type, does not have a member `result_type`. Even worse, there is no way to add such member to a type that has already been defined or to add it at all to built-in types such as `int`. On line 22, `plus` is called with two arguments of type `NotNumber`, which, as its name suggests, is not a number. Incidentally, however, `NotNumber` matches the syntactic requirements of the `plus` template and the instantiation succeeds even though the addition operator does not have the desired semantics. The constrained template has the exactly opposite behavior. When, on line 23, `cplus` is instantiated with arguments of type `NotNumber`, a concept map `Addable<NotNumber>` does not exist and the instantiation process is halted with a corresponding error. For `int`, however, an `Addable<int>` concept map is defined on lines 17–18. This map provides implementations for the associated names of the concept `Addable`; in particular, `result_type` is bound to `int` and `operator+` to integer addition.

The process of separate type checking consists, in summary, of 3 major steps:

1. Checking of *constrained templates* in the context of *concept requirements* on their type parameters;

2. *implementation binding* in *concept maps* checked against the signatures in corresponding concepts;

3. instantiation of templates that were successfully type-checked with arguments for which appropriate concept maps exist.

Type checking of constrained templates, thus, is essentially a name lookup problem; the constraints on the template parameters establish an environment in which names used in a constrained template are looked up. Once the names are found, a template body is checked as if the template parameters were usual types. In the second step, particular implementations are bound to the names provided by concepts. Name lookup and implementation binding must match; for example, if an associated function is chosen between duplicate signatures, these signatures must later have identical implementation so the semantics of a program does not depend on the order of definitions.

2.2 Refinement, Name Lookup, and Implementations Binding

In the example in listing 2.1, there was only one concept and only one concept map but generic libraries are always more complex. Instead of giving many large concepts, generic libraries are founded upon conceptual hierarchies in which concepts *refine* other, simpler concepts. The semantics of name lookup and implementation binding is complicated by refinement. Name lookup in constrained templates must be performed throughout refinement hierarchies. Implementation bindings can be made at different points of these hierarchies, requiring implementation compatibility checking and propagation.

2.2.1 Name Lookup

Name lookup in constrained templates applies to names of types and functions. We first discuss the two independently, in listings 2.2 and 2.3, and then, in listing 2.4, give an example of how the two lookups interact.

```
concept A<typename T>              { typename t; }
concept B<typename T> : A<T>       { typename t; }
concept C<typename T>              { typename t; }
concept D<typename T> : B<T>, C<T> { }
concept E<typename T> : A<T>, C<T> { }

template<typename T> requires D<T>
struct Test1 { typedef t test1;        typedef A<T>::t test2;
               typedef B<T>::t test3;  typedef C<T>::t test4;  };

template<typename T> requires E<T>
struct Test2 { typedef t test1;                              };
```

Listing 2.2: Name lookup of associated types

In listing 2.2, a refinement hierarchy is formed where concept `B` refines concept `A`, concept `D` refines concepts `B` and `C`, and concept `E` refines concepts `A` and `C`. Each of the concepts `A`, `B`, and `C` introduces a requirement for an associated type `t`, while concepts `D` and `E` do not introduce any new requirements but only refine other concepts. Two template structures demonstrate how name lookup proceeds. The type parameter of the first structure, `Test1`, is constrained by `D`. In definition of `test1`, which simply aliases the name `test1` to `t`, `t` is looked up in the scope of requirements, specifically in `D<T>`, which is the only requirement. In `D` there are three types named `t`, from `A`, `B`, and from `C`, but the `t` of `A` is *hidden* by the `t` in `B`. Still, there are two `t`s to choose from, in `B` and `C`. In our semantic definition (section 4), we state that the two `t`s must be the same in this case, although the concept wording (Gregor et al., 2008e) requires that for the lookup to succeed without ambiguity, the equivalence of the two types must be declared explicitly, through compiler-supported concepts. Our decision to assume the equivalence of types stems from a fix to an error that we have discovered in the wording—we discuss the details in section 6. In cases like this, where types are equivalent, a representative still has to be chosen as the lookup result; the wording proposes the one found in the depth-first, left to right traversal of the refinement hierarchy in the order prescribed by the syntactic order in refinement lists. Consequently, `test1` becomes an alias for the `B<T>::t`. In the remaining definitions, `t` is looked up in the qualifying scopes; `B<T>::t` and `C<T>::t` are the same type since they are both visible and in the same refinement hierarchy, but `A<T>::t` may name a different type since it is hidden by `B<T>::t`. In the second template, `Test2`, constrained by the requirement `E<T>`, there are two `t`s visible, one in `A` and one in `C`. Since there is no hiding, `t` in the definition of `test1` resolves to `A<T>::t` because of the depth-first traversal strategy (note that `A<T>::t` and `C<T>::t` are the same type).

Listing 2.3 gives an example of function name lookup in the presence of refinement. Two concepts, `A` and `B`, have an associated function `void f()`. The concepts `C` and `D` both refine `A` and `B` but `C` introduces its own associated function `f`. In the template `f1` constrained by `C`, the functions from `A`, `B`, and `C` are visible and `f` from `C` is selected. Although incidentally the result of the name lookup is very similar to that in inheritance, name lookup proceeds entirely differently. The refinement hierarchy is traversed, starting at `C`, and an overload set that contains all functions named `f` found

```
concept A<typename T>              { void f(); }
concept B<typename T>              { void f(); }
concept C<typename T> : A<T>, B<T> { void f(); }
concept D<typename T> : A<T>, B<T> { }

template<typename T> requires C<T>
void f1() { f(); }  // C<T>::f

template<typename T> requires D<T>
void f2() { f(); }  // A<T>::f

template<typename T> requires A<T>, B<T>
void f3() { f(); }  // error, ambiguous
```

Listing 2.3: Name lookup of associated functions

in the hierarchy is constructed—this is different from class hierarchies where name lookup may only result in overload sets containing names from a single class (clause 10.2 of the C++ standard). The overload set may contain functions with identical signatures, declared in different concepts in the refinement hierarchy; these signatures are assumed to refer to the same *specification* of a function. Consequently, rather than causing ambiguity, one of the identical signatures is chosen as a unique representative while the others are removed from the overload set. As in type lookup, the strategy for picking representatives is to pick the signature found first in a depth-first traversal of a refinement hierarchy and `C<T>::f` is thus chosen. In the template `f2`, the situation is similar and `A<T>::f` is chosen. In the template `f3`, there are two requirements, `A<T>` and `B<T>`. In this case, the two signatures `A<T>::f` and `B<T>::f` are visible, but since they are not part of the same refinement hierarchy, as in `f2`, it can be no longer safely assumed that the two signatures represent the same function and the lookup results in ambiguity.

```
concept A<typename T>        { typename t; void f(t); }
concept B<typename T> : A<T> { typename t; void f(t); }

template<typename T> requires B<T>
void f1(A<T>::t a, B<T>::t b) { f(a); f(b); }  // calls A<T>::f and B<T>::f

template<typename T> requires B<T>, A<T>
void f2(A<T>::t a, B<T>::t b) { f(a); f(b); }  // error, ambiguous
```

Listing 2.4: Name lookup of associated types and functions

The last example of name lookup, in listing 2.4, shows how lookups of associated types and functions interact. The refinement hierarchy is simple, concept `B` refines concept `A`. Both concepts require an associated type `t` and an associated function `void f(t)`. The `t` in `B` shadows the `t` in `A` and the two functions thus have different signatures because the two `t`s are not equivalent. The function template `f1` takes two arguments of types `A<T>::t` and `B<T>::t` and then calls function `f` with each of the arguments. In the first case, `A<T>::f` is called and in the second `B<T>::f`, as a result of overload resolution with both functions in the overload set and the correct calls chosen based on the arguments. The second template, `f2`, adds `A<T>` to the list of requirements. Name lookup is performed in each of the requirements and `A<T>::f` is discovered twice, causing ambiguity in the call `f(a)` but the second call succeeds since there is only one `void f(B<T>::t)` available. In a conceivable alternative design, the second requirement for

A<T> could be "merged" with the same requirement introduced through B<T> but this is not the semantics in the current concept wording (Gregor et al., 2008e).

2.2.2 Implementation Binding

Concepts provide an environment of function and type names. In the previous section, we have shown how this environment is used to check constrained templates. Next, we show how particular implementations are bound to the names provided by concepts. For convenience, only some implementations have to be provided at certain points of the refinement hierarchy; these are then propagated along the refinement relation. At the same time, the compiler checks that no entities have multiple, possibly incompatible, realizations, and that no entities are left unimplemented. Consequently, the main parts of implementation-binding semantics are *implicit concept map generation*, which propagates implementations, and *compatibility checking*, which ensures there are no conflicting implementations.

```
1  concept A<typename T>              { void f(); }
2  concept B<typename T>              { void f(); }
3  concept C<typename T> : A<T>, B<T> { }
4
5  concept_map A<int> { void f() { return; } }
6  concept_map B<int> { void f() { return; } }
7  concept_map C<int> { }  // error, two implementations for f
8
9  concept_map C<bool> { void f() { return; } }
10 // concept maps for A<bool> and B<bool> implicitly generated
11
12 concept_map A<float> { void f() { return; } }
13 concept_map C<float> { }  // error, no implementation of f to push to B<float>
```

Listing 2.5: Implementation binding for functions

Listing 2.5 shows how implementations are bound to associated function requirements. Two concepts, A and B, each require an associated function void f(). Concept C only refines A and B without adding any new requirements. On lines 5–7, concept maps for A<int> and B<int> are defined, each binding an implementation to function f. Refining concepts must provide definitions for all the requirements from all the concepts they refine but they can do so *explicitly* or *implicitly*. An implicit definition "pulls" definitions from refined concepts and an explicit one "pushes" to refined concepts. The concept map for the refining concept C on line 9 defines the function f explicitly, while the concept maps on lines 7 and 13 make the definition implicit. When a concept map for C<int> is defined, the implementations from its refined concepts are propagated and there are two implementations for f. Even though the two implementations are obviously the same, the wording does not force compilers to compare functions because, in general, such comparison is not feasible. Consequently, the concept map for C<int> fails. The concept map for C<bool>, on line 9, includes a definition for f although it is not required by the concept C itself; this definition is used to implicitly generate maps for A<bool> and B<bool>. Finally, the concept map on line 13 fails because it does not contain an explicit definition to push to its refined concept B and the implementation from A<float> seen on line 12 is not allowed to be pushed "sideways" to B<float>. Implementation binding for associated types is very similar, except that more than one definition is allowed since it is easy to check whether two type names reference the same type. Thus, concept maps such as those on lines 5–7 would be accepted for types

if both definitions were to the same type.

3 Formalization

Our formalization is carried out for a subset of ConceptC++, comprising concepts, templates, and concept maps; for ease of reference we name the language X++. In this section we discuss the definition, conventions, and the ConceptC++ constructs included in X++. In the next section, we give a semantic definition of separate type checking.

3.1 Definition and Conventions

X++ is defined using Ott (Sewell et al., 2007), a tool for a "working semanticist." The metalanguage of Ott is used to define the syntax of X++ and the semantic judgments given as inductive relations over the syntax. From that metadescription, Ott can generate LaTeX output as well as formal definitions for Coq, HOL, and Isabelle/HOL. Our semantics of X++ comprises 62 judgment rules, corresponding to 62 relations, with the total of 229 rule clauses specifying those relations. In this paper we show only a fraction of the rules, the complete definition is available in the accompanying technical report (Paper VI).

The definition of X++ is checked by Ott: the grammar is parsed and sort-checked, inconsistencies in judgment forms and metavariable naming conventions are prevented, and bounds of list forms are checked. Sequences are indicated by overline: $\overline{x}$ is a sequence of xs, $\overline{x_i}^{\,i}$ is the same but with indexes, $\overline{\overline{x_i}^{\,i}}$ is a sequence formed by concatenating a number of sequences of $\overline{x_i}$, $\overline{x_i\,\overline{x_i}^{\,i}}$ is a sequence formed by prepending an element x_i to a sequence $\overline{x_i}$ and then concatenating the resulting sequences. Formulas can also be sequenced; for example, $\overline{x_i = y_i}^{\,i}$ compares each x_i and y_i from some sequences $\overline{x_i}^{\,i}$ and $\overline{y_i}^{\,i}$. Syntax printed in `monotype` signifies terminals representing C++ syntax. Every sequence may be treated as a set when necessary. Some rules use function syntax to indicate that in principle they could be written as functions rather than as relations. Grammar rules can be annotated with an M for metarules. Metarule annotation causes Ott not to generate constructors for the particular rule in a theorem prover output and not to consider this rule when parsing example input. The [l] annotations signify that a rule is not part of the user syntax and is only used on the metalevel in the semantic rules. Finally, the annotation S states that a rule is a metarule but it can be parsed in concrete syntax.

In the metatheory, we use *options*, indicated by adding a question mark to an identifier. For example, *cid*? can either be a *cid* or *None*. Given a list of options, the function *rm*? removes all the *None* options; for example, $rm?\,(\overline{cm?})$ gives a list $\overline{cm}$ with all *None* options removed. In some rules we use set comprehension, where $\{x \in y \mid \cdots\}$ gives all elements x in y that fulfill some condition. Set comprehension is assumed to be an operation on sequences that removes all elements that do not fulfill the comprehension criterion and all duplicates.

Errors are not handled explicitly. Semantic judgments capture the expected behavior of a compiler only for a correct program. An ill-formed program will simply cause one to get "stuck" when searching for a derivation tree of a semantic relation.

e	::=			P	::=		program
	|	var	variable		|	d	M	one
	|	`obj` τ	object		|	$\overline{P_i}^{\,i}$	M	flatten
	|	$f\,(\overline{e_i}^{\,i})$	application	d	::=		definitions
	|	v_{int}	integral		|	cp		concept
	|	v_{bool}	boolean		|	tl		template
	|	(e)	S parenthesis		|	cm		concepts map
					|	icm	[I]	implicit map

Figure 2.1: Grammar of expressions and top-level program syntax

cp	::=		concept
	|	con `refines` $\overline{cid}\,\{\,\overline{ty^a}\;\overline{f^a}\,\}$	def.
tl	::=		template
	|	`template` $req\;tn\,\{\,e\,\}$	def.
req	::=		requirements clause
	|	`requires` $\overline{cid_i}^{\,i}$	def.
cm	::=		concept map
	|	`concept_map` $cid\,\{\,\overline{tydef}\;\overline{fdef}\,\}$	def.
	|	$icm\,.\,cm$	M[I] implicit concept map

Figure 2.2: Grammar of concepts, templates, and concept maps

3.2 X++ Language

Figure 2.1 shows the syntax of X++ expressions and programs, while figure 2.2 lists the grammar for the three basic constructs of X++ (for the complete grammar, see (Zalewski, 2008)): concepts (Gregor et al., 2008e, clause 14.9.1), concept-constrained templates (Gregor et al., 2008e, clause 14.10), and concept maps (Gregor et al., 2008e, clause 14.9.2). An X++ program P comprises definitions of concepts, templates, and concept maps. The syntax of a user program, which is omitted in figure 2.1, is denoted by P_{user} and is the same as the syntax of a program in figure 2.1 except that it does not include implicit concept map definitions (the *icm* production for d). Expressions are kept simple and the syntax is self-explanatory.

In none of the examples in section 2 did type parameters in templates and concepts play a role and accordingly X++ excludes those parameters. Although some parts of the semantics of concepts cannot be formulated without concept and template parameters, the name lookup and implementation binding semantics we consider is independent of a particular choice of template arguments.

A concept consists (figure 2.2) of a concept name *con* followed by a refinement clause, which contains a list of concept identifiers, $\overline{cid}$, followed by the concept body. A concept identifier, *cid*, represents a concept instantiated with some arguments, and in X++ is simply just a concept name *con*. For example, in the refinement "`B<typename T> : A<T>`," concept `B` refines a concept instance `A<T>`. In X++, the difference between a concept name *con* and a concept identifier *cid* is that a concept name is used in a definition of a concept and the concept identifier is used to refer to that concept. While in the concept wording (Gregor et al., 2008e), refinement is specified as a relation between concepts, in X++, refinement is a relation either between concepts and concept instances (in concept definitions) or between concept instances (in concept uses). Direct refinement ($\prec_1$) and its transitive closure ($\prec$) is described by the following 2 rules:

$$\frac{1.\ \mathit{refined}\,(C, \mathit{cid}) = \overline{\mathit{cid}}}{C \vdash \mathit{cid} \prec_1 \overline{\mathit{cid}}} \qquad \frac{1.C \vdash \mathit{cid} \prec_1 \overline{\mathit{cid}_i}^{\,i} \qquad 2.\overline{C \vdash \mathit{cid}_i \prec \overline{\mathit{cid}}_i}^{\,i}}{C \vdash \mathit{cid} \prec \overline{\mathit{cid}_i\,\overline{\mathit{cid}}_i}^{\,i}}$$

The rules should be interpreted as an inductive relation over abstract syntax, where, for each fraction, top and bottom are related. In the context of programming languages, the rules can be read top-to-bottom, considering only one direction of the relation: the simpler premises on the top give a semantic conclusion on the bottom. In our presentation we number premises for easy reference. The first rule determines the meaning of direct refinement denoted by $\prec_1$. The sole premise is that the *refined* function, defined elsewhere, returns a sequence of concept identifiers $\overline{\mathit{cid}}$ given a list of defined concepts C and a concept identifier *cid*. The conclusion states that *cid* directly refines $\overline{\mathit{cid}}$ in the context of C (indicated by the turnstile). This rule is the base case of refinement. The second rule defines the transitive closure $\prec$ of direct refinement $\prec_1$. The first premise extracts the list of directly refined concepts, containing i elements, and the second premise applies the refinement relation to each one of the directly refined concepts obtaining i new lists $\overline{\mathit{cid}}$. This is repeated recursively, since the second premise is the refinement ($\prec$) relation itself, until the leaves of the refinement tree, i.e., concepts with empty refinement clause for which $i = 0$, are reached. In the conclusion, all the results are concatenated together giving a list of all refined concepts. An important property of the refinement relation definition is that the syntactic order of concept identifiers in concept refinement clauses is preserved. This property is not apparent from the semantic rules we have listed because it depends on the function *refined*, which extracts a list of concept identifiers from a refinement clause preserving their syntactic order (see (Zalewski, 2008)). The list of refined concepts, $\overline{\mathit{cid}_i\,\overline{\mathit{cid}}_i}^{\,i}$, in the transitive closure rule, preserves the depth-first, syntactic order in which concepts were visited, since every concept identifier is followed directly by the ordered list of concept identifiers reached from it in the traversal.

A concept body (figure 2.2) contains concept requirements comprising associated types $\overline{\mathit{ty}^a}$ and associated functions $\overline{f^a}$. In ConceptC++, the requirements can be freely intermixed but for simplicity, X++ requires that associated types precede associated functions. A template (figure 2.2) consists of a requirements clause, a template name *tn*, and a template body with, for simplicity, a single expression. The requirements clause, next, consists of concept identifiers, each representing a requirement that must be fulfilled by a corresponding concept map. In X++, there is no distinction between function templates (e.g., `cplus` in listing 2.1) and class templates (e.g., `Test1` in listing 2.2). Lastly, a concept map provides type and function definitions necessary to satisfy the requirements corresponding to a particular concept id *cid*. In X++, there can be only one concept map per concept. The second syntax rule for concept maps, at the bottom of figure 2.2, is used in the metatheory. It states that given an *implicit* concept map *icm*, which, as explained in section 4, is a tuple of a concept map and a concept identifier, the concept map part of the tuple can be extracted using syntax akin to field access.

4 Semantics of Separate Type Checking

In section 2 we have informally outlined how separate type-checking safety is achieved. Here, we define separate type checking safety formally, in the form of a theorem:

Theorem 1 (Separate type-checking safety)**.** *Let C, M, and T represent an environment of the defined concepts, the defined concept maps, and the defined templates,*

respectively. Let P_{user} be a program, tl be a template, req be a requirements clause of a template. Assume

1. $P_{user} \Downarrow C; M; T$*, i.e., the program* P_{user} *is correct and produces the environment* $C; M; T$ *when processed,*
2. $tl = \texttt{template}\, req\, tn\, \{ e \} \in T$*, i.e., tl is a specific template in the environment,*
3. $req = \texttt{requires}\, \overline{cid_i}^{\,i}$*, i.e., the requirements of the template tl are given by concept identifiers* $\overline{cid_i}^{\,i}$*, and*
4. $\overline{cm(M, cid_i) = cm_i}^{\,i}$*, i.e., for every concept identifier* cid_i *a concept map* cm_i *can be found in the list of defined concept maps.*

Then, instantiating the template tl is guaranteed to succeed. Template instantiation can be reduced to binding symbols from each requirement cid_i *to definitions in the corresponding* cm_i*, and no additional checks are necessary.*

The theorem states that given a correct program, i.e., one that instantiates ($\Downarrow$), a concept-constrained template can be *safely* instantiated without any additional type checking, which otherwise is necessary for unconstrained templates. The particular implementations of the types and functions used in the template are taken from the concept maps corresponding to the requirements rather than from the surrounding or argument-dependent namespaces as in the instantiation of unconstrained templates. The first premise states that the program P_{user} instantiates to an environment $C; M; T$. The premises 2–4 simply introduce names for a template and the concept maps that are referred to in the conclusion of the theorem.

In the remainder of this section we present in detail the crucial *program instantiation* relation that is the centerpiece of our semantic definition. The two major tasks of the program instantiation relation are processing of constrained templates and implementation binding in concept maps—the discussion is organized accordingly. In this section, we concentrate on the discussion of our semantics and in section 6 we discuss its correspondence to the wording. We informally argue the correctness of theorem 1 in section 5; there, we also illustrate the working of the semantics by particular examples.

4.1 Program Correctness

The program instantiation relation takes a context $C; M; T$ and a program to a new context $C'; M'; T'$. The relation consists of 5 rules, one for the empty program and one per d-production in figure 2.1. The rules for templates and implicit concept map definitions are shown in figures 2.5 and 2.7 and discussed in detail in the following subsections. The rules for processing concepts, concept maps, and empty programs are omitted here but can be found in the companion technical report (Paper VI). The rule for processing concepts contains only a basic check that a concept is not already defined in the context. The rule for processing concept maps simply forwards all the work to the rule for processing implicit concept maps and the rule for processing empty programs is trivial.

It is important to note that while our semantic definition of program instantiation defines which programs are correct, there is no effort made to establish in what way an incorrect program is wrong. If a program is incorrect, it is simply impossible to show that it instantiates but no "error diagnostic" can be obtained as to why. This is a usual

property of formal semantic definitions—error handling has to be added by compiler writers.

Both type checking of constrained templates and checking of concept map definitions depend on the formalization of name lookup in concepts. Accordingly, we preceed the discussion of the program instantiation rules with a presentation of the rules for name lookup.

4.1.1 Name Lookup

Name lookup for concepts, described in clause 14.9.3.1 (of (Gregor et al., 2008e)), is quite different from other name lookups in C++; we have given some examples in listings 2.2, 2.3, and 2.4, in section 2.2.1. The main difference is that functions and types may be declared multiple times throughout a refinement hierarchy but, in difference to other name lookups in C++, without causing ambiguity—if there are duplicates, a single representative is chosen. Clause 14.9.3.1 designates the unique representative to be the type or function found first in the depth-first traversal of the refinement structure, where refinement lists preserve the textual order in which refinements were given in a program. The choice of the unique representative depends on the context in which an entity is looked up. Because of this dependence, names have to be *normalized* in a context; normalization for our language simply replaces all occurrences of type names with their unique representatives in the given context, each representative specified by a fully qualified name (e.g., `A::t` for `t` or `B::t` for `C::t`). In addition, due to shadowing, a name looked up in a refinement hierarchy must be normalized to its unique representative twice, once in the context where it is defined and then in the context in which it is looked up. Since function names are not qualified by scope in our language, normalization to unique representatives applies only to type names. Every lookup function in our semantics performs normalization to ensure names are not mixed up. Furthermore, in other rules, entities involving type names that are not accessed via normalizing lookups must be normalized explicitly. Performing normalization throughout the rules allows distinguishing the names that should be the same in some context from the names that should be different.

$$\frac{1.\,con\,(cid) = con \qquad 2.\mathit{find\text{-}concept}\,(C, con) = cp \qquad 3.\,\overline{f^a}\,(cp) = \overline{f^a} \qquad 4.\mathit{overload\text{-}set}\,(f, \overline{f^a}) = \overline{sig} \qquad 5.\mathit{normalize}\,(C, cid, \overline{sig}) = \overline{sig}'}{\mathit{find\text{-}scope}\,(C, cid, f) = \overline{sig}'}$$

$$\frac{1.\,C \vdash cid \prec \overline{cid_i}^{\,i} \qquad 2.\mathit{find\text{-}scope}\,(C, cid, f) = \overline{sig} \qquad 3.\overline{\mathit{find\text{-}scope}\,(C, cid_i, f) = \overline{sig_i}}^{\,i} \qquad 4.\,\overline{sig}' = \overline{sig}\,\overline{\overline{sig_i}}^{\,i} \qquad 5.\mathit{normalize}\,(C, cid, \overline{sig}') = \overline{sig}'' \qquad 6.\mathit{rdup}\,(\overline{sig}'') = \overline{sig}'''}{\mathit{find\text{-}rec}\,(C, cid, f) = \overline{sig}'''}$$

Figure 2.3: Function name lookup in constrained context

Figure 2.3 lists the rules for function name lookup. The first rule describes how an overload set is extracted from each concept locally and the second describes the recursive search for associated functions in a concept hierarchy. The first two premises of the first rule extract the name *con* of a concept from a concept identifier and check that such concept indeed exists in the current environment. The third premise extracts a list of associated functions, $\overline{f^a}$, from the concept and in premise 4 an overload set is constructed, extracting signatures for functions named f from $\overline{f^a}$. Premise 5 is the most involved. For every signature in the overload set, the types in the signature are

looked up in the current concept. This is done so that signatures can be distinguished later from other signatures with the same unqualified parameter types; for example, `void f(t)` may be `void f(X::t)` in one concept and `void f(Y::t)` in another. To distinguish these functions later, fully qualified type names must be used for parameter types. The definitions of the functions used in the premises, e.g., *normalize*, are listed in the report (Paper VI).

The recursive rule first finds the sequence of refined concept identifiers that, as discussed in section 3, preserves the depth-first traversal order of the refinement hierarchy. Then, in the second premise, associated function signatures, $\overline{sig}$, are extracted from the concept *cid*, the root of the current lookup. In the third premise, a sequence of associated function signatures, $\overline{sig}_i$, is extracted from each of the concepts, cid_i, found in the first step. The sequences of function signatures are concatenated in the same order as their enclosing concepts in the sequence $\overline{cid_i}^i$, preserving the depth-first traversal order of discovery. In premise 4, the signatures $\overline{sig}_i$ from the refined concepts are appended to the sequence of signatures for the root of the lookup, $\overline{sig}$. When looked up in their local scopes function signatures are normalized but in premise 5 they are normalized again in the context of the concept in which the search was initiated. This step is performed to put the signatures in the same "perspective," identifying types that should be the same. Finally, in premise 6, the function *rdup* (defined elsewhere) removes duplicate signatures from the sequence $\overline{sig}'$, keeping the signatures encountered first and removing the subsequent duplicates, keeping only a single representative for each function.

$$\frac{1.\,\mathit{find\text{-}scope}\,(C, cid, id_\tau) = \tau}{\mathit{find\text{-}rec}\,(C, cid, id_\tau) = \tau} \qquad \frac{\begin{array}{ll} 1.\,\mathit{find\text{-}scope}\,(C, cid, id_\tau) = \mathit{None} & \\ 2.\,C \vdash cid \prec_1 \overline{cid_i}^i & 3.\,\overline{\mathit{find\text{-}rec}\,(C, cid_i, id_\tau) = \tau?_i}^i \\ 4.\,\mathit{rm?}\,(\overline{\tau?_i}^i) = & \end{array}}{\mathit{find\text{-}rec}\,(C, cid, id_\tau) = \mathit{None}}$$

Figure 2.4: Selected rules for recursive type lookup

Type lookup is similar to function lookup except that only one type is returned rather than a set, as in the case of functions. On the other hand, there are more cases to handle since types can be hidden. All together, the recursive part of type lookup comprises 6 rules, of which 2 are shown in figure 2.4. The first rule is the base case of the recursive search: if the scope identified by *cid* contains an associated type τ named id_τ, the search is terminated and $cid :: id_\tau$ is the result of this part of the lookup. There may be other concepts that *cid* refines but hiding rules (clause 3.3.10 para. 1) require that they not be searched. The first premise of the second rule ensures that no type was found in the current concept. Then, the second premise extracts the directly refined concepts ($\prec_1$). If the transitive refinement ($\prec$) was applied instead, the base rule could not stop the search from progressing into scopes in which potentially matching types should be hidden by an earlier result. In the third premise, the refined scopes are searched recursively, until the first rule is matched or the list of refinements extracted in the second premise is empty. For each of the searched scopes, cid_i, a type option, $\tau?_i$, is returned. Finally, the fourth premise states that after removing all *None* results, which signify unsuccessful lookups in refined scopes, the sequence of results is empty. In such case, the result of the whole lookup is *None* since no type named id_τ was found.

Given the definition of name lookup, we next discuss the two parts of separate type checking: checking of constrained templates and of implementation binding.

4.1.2 Type Checking of Constrained Templates

$$\frac{1.\, tl = \texttt{template}\,\texttt{requires}\,\overline{cid_i}^{\,i}\; tn\,\{e\} \qquad 2.\, \overline{cid_i \text{ defined in } C}^{\,i} \qquad 3.\, tl \text{ not defined in } T \qquad 4.\, C;\emptyset;\overline{cid_i}^{\,i} \vdash e :\ \tau \qquad 5.\, C;M;tl\,T \vdash P \Downarrow C';M';T'}{C;M;T \vdash tl\,P \Downarrow C';M';T'}$$

Figure 2.5: Type checking of constrained templates

Processing of constrained templates is the first step of separate type checking. The rule in figure 2.5 shows how a template definition tl, in a program $tl\,P$, is checked in the context $C;M;T$, producing a new context $C';M';T'$. Checking of constrained templates boils down to type checking of the expression e contained in the template tl in the context given by the requirement clause of the template; premise 4 states that the expression e is of the type τ (written $e :\ \tau$) in the context of all concepts C (necessary to traverse the refinement hierarchy), an empty variable typing environment $\emptyset$, and the requirements $\overline{cid_i}^{\,i}$ of the template tl. The other premises, in order, introduce the full definition of tl, ensure that every concept referred to in the requirements is defined, assert that a template with the same name is not already defined, and that processing the rest of the program P with the currently processed template inserted into context $(tl\,T)$ produces the new context $C';M';T'$.

$$\frac{1.\, \overline{\mathit{find\text{-}rec}\,(C, cid_i, f) = \overline{sig_i}}^{\,i} \qquad 2.\, \mathit{distinct}\,(\overline{\overline{sig_i}}^{\,i}) \qquad 3.\, \overline{C;\Gamma;\overline{cid_i}^{\,i} \vdash e_k :\ \tau_k}^{\,k} \qquad 4.\, \mathit{overload\text{-}res}\,(f\,\overline{\tau_k}^{\,k}, \overline{\overline{sig_i}}^{\,i}) = sig \qquad 5.\, \mathit{returns}\,(sig) = \tau}{C;\Gamma;\overline{cid_i}^{\,i} \vdash f\,(\overline{e_k}^{\,k}) :\ \tau}$$

$$\frac{1.\, \overline{\mathit{find\text{-}rec}\,(C, cid_i, \tau') = \tau?_i}^{\,i} \qquad 2.\, rm?\,(\overline{\tau?_i}^{\,i}) = \overline{\tau} \qquad 3.\, \overline{\tau} \approx \tau}{C;\Gamma;\overline{cid_i}^{\,i} \vdash \texttt{obj}\,\tau' :\ \tau}$$

Figure 2.6: Selected rules for typing expressions in constrained contexts

An expression is typed in the context consisting of available concepts C, a variable typing environment Γ, and a list of concept identifiers $\overline{cid}$ that represents the requirements of a constrained template. Type checking of expressions includes some expected rules, for example, the type of a variable is looked up in a local environment. The difference to the usual type checking is in name lookup, which is performed in the concepts that the constrained template requires. Figure 2.6 shows typing of function application and of object creation, which require name lookup.

Type checking a function application $f\,(\overline{e_k}^{\,k})$ consists of 5 steps. First, *find-rec* looks up f in each of the requirements and the lookup gives a sequence of function signatures representing an overload set from each of the requirements. The second step ensures that function signatures found through name lookup are distinct and that there are no ambiguities in the overload set. In the third premise, the arguments with which the function is called are typed themselves. Then, in premise 4, overload resolution is invoked to choose one signature from the overload set given the function name and the types of the arguments with which the function is called. Finally, in the fifth premise, the return type of the function is extracted. Given all these premises, function application is typed with the return type of the function selected by overload resolution.

Type checking of object creation starts with name lookup, in each of the requirements, for the type with which the object is created. Each lookup returns a type option,

which is *None* if name lookup failed. In premise 2, the *None* results are removed, yielding a list of types that were successfully looked up. Premise 3 states that only one distinct type was found by requiring set equality ($\approx$) between the sequence of results $\overline{\tau}$ and a sequence of a single type τ. In such case, the created object can be typed with type τ.

In the following subsection, we discuss the next step of separate type checking as outlined in section 2, namely implementation binding.

4.1.3 Implementation Binding, Compatibility, and Propagation

$$\frac{\begin{array}{l}1.\, icm = (cm, cid?) \qquad 2.\, cm = \texttt{concept\_map}\, cid\, \{\, \overline{tydef}\, \overline{fdef}\, \} \qquad 3.\, cm \text{ not defined in } M \\ 4.\, tydefs\text{-}check\,(C, cid, \overline{tydef}) \qquad 5.\, tydefs_{=\!=}\,(C, M, cid, \overline{tydef}) \\ 6.\, fdefs\text{-}check\,(C, cid, \overline{fdef}) \qquad 7.\, fdefs_{=\!=}\,(C, M, cid, cid?, \overline{fdef}) \\ 8.\, icms\,(C, M, cid, cid?, \overline{tydef}, \overline{fdef}) = \overline{icm_i}^{\,i} \qquad 9.\, C; icm\, M; T \vdash \overline{icm_i}^{\,i}\, P \Downarrow C'; M'; T'\end{array}}{C; M; T \vdash icm\, P \Downarrow C'; M'; T'}$$

Figure 2.7: Checking and generation of implicit concept maps

The two main tasks when processing a concept map are to check whether the definitions that the map provides are compatible with the definitions in concept maps defined earlier and to implicitly define concept maps for refined concepts as necessary (clause 14.9.3.2). The program instantiation rule for processing concept maps is listed in figure 2.7. We first give an overview of the rule and then discuss it in detail in the remainder of the section.

The first premise simply creates an alias for the implicit concept map *icm* that is being processed, which is a tuple of a concept map *cm* and an optional concept identifier *cid*; the second premise creates an alias for the *cm* component of *icm*. The third premise establishes that a concept map for the same concept instance has not yet been defined. Premises 4 through 7 state that type and function definitions are well-formed, well-typed, and compatible ($=\!=$) with definitions provided in existing concept maps. Premise 8 gives the set of implicit concept maps that are generated by the concepts directly refining the currently processed concept map *cm*. And finally, premise 9 finds the new environment $C'; M'; T'$ resulting from processing the newly generated implicit concept maps and the rest of the program in the old environment extended by the current concept map. Each of the generated implicit concept maps is then processed in the same manner as the map that initiated their generation. This recursive processing ensures that implicit concept maps are propagated throughout the refinement hierarchy, traversed depth first.

$$\frac{1.\, \forall\, \tau f\, (\overline{var_i : \tau_i}^{\,i})\, \{\, e \,\} \in \overline{fdef} \bullet C; [\overline{var_i : \tau_i}^{\,i}]; cid \vdash e : \tau \qquad 2.\, distinct\,(\overline{sig}\,(\overline{fdef}))}{fdefs\text{-}check\,(C, cid, \overline{fdef})}$$

$$\frac{\begin{array}{l}1.\, \overline{ty^a}\,(C, cid) = \overline{ty^a}^{\prec} \qquad 2.\, \overline{id_\tau}\,(\overline{tydef}) \approx \overline{id_\tau}\,(\overline{ty^a}^{\prec}) \\ 3.\, \forall\, tydef \in \overline{tydef} \bullet tydef\,\surd \qquad 4.\, distinct\,(\overline{id_\tau}\,(\overline{tydef}))\end{array}}{tydefs\text{-}check\,(C, cid, \overline{tydef})}$$

Figure 2.8: Function and type definitions check in concept maps

Figure 2.8 shows how function and type definitions are checked in the premises 4 and 6 in figure 2.7. For functions, the first condition is that the body of every function definition ($\bullet$ separates the quantifier from the condition) must type-check in the

context of the concepts C in the environment, the function parameters $[\overline{var_i : \tau_i}^i]$, and the concept identifier *cid*, designating the concept map to which the function definition belongs. Type checking proceeds as described previously (figure 2.6).

Checking of type definitions differs from checking function definitions. The first premise extracts associated types, $\overline{ty^a}^{\prec}$, from all refined concepts (indicated by $\prec$ in $\overline{ty^a}^{\prec}$) and the second premise requires that type definitions in $\overline{tydef}$ provide a definition for every type in $\overline{ty^a}$. The equality in the second premise is a set equality because different concepts refined by *cid* may require an associated type with the same name. Premise 3 requires that every type definition is well-formed (see Zalewski (2008) for details) and premise 4 requires that there is only one definition per associated type name.

$$\frac{\begin{array}{ll}1.\,\overline{f^a}\,(C, cid) = \overline{f^a}^{\prec} & 2.\,\overline{sig}\,(C, M, cid?, cid) = \overline{sig}^{\prec} \\ 3.\,distinct\,(\overline{sig}^{\prec}) & \\ 4.\,normalize\,(C, cid, \overline{sig}\,(\overline{fdef})) = \overline{sig} & \\ 5.\,\overline{sig} \approx \overline{f^a}^{\prec} \setminus \overline{sig}^{\prec} & \end{array}}{fdefs_{=}\,(C, M, cid, cid?, \overline{fdef})}$$

$$\frac{\begin{array}{ll}1.\,\overline{ty^a}\,(C, cid) = \overline{ty^a} & 2.\,C \vdash cid \prec_1 \overline{cid}^{\prec_1} \\ 3.\,\overline{tydef}\,(M, \overline{cid}^{\prec_1}) = \overline{tydef}^{\prec_1} & 4.\,\overline{ty^a} \vdash \overline{tydef} = \overline{tydef}^{\prec_1}\end{array}}{tydefs_{=}\,(C, M, cid, \overline{tydef})}$$

Figure 2.9: Function and type definitions compatibility in concept maps

The next step is to check that the newly introduced implementations are *compatible* with the existing ones (premises 5 and 7 in figure 2.7). The rules for compatibility checking are listed in figure 2.9. For functions, first all associated function requirements are extracted from refinements, including the associated functions from the currently considered concept. The second premise finds the signatures of the existing implementations, omitting concept maps that are implicitly defined by the same parent (based on the *cid*? argument to $\overline{sig}$). The omission is necessary to properly handle diamonds in the refinement hierarchy; an example of such hierarchy is shown in the following code snippet:

```
1 concept A<typename T>                 { void f(); }
2 concept B1<typename T> : A<T>         { }
3 concept B2<typename T> : A<T>         { }
4 concept C<typename T>  : B1<T>, B2<T> { }
5
6 concept_map C<int> { void f() { return; } }
```

The definition of `void f()` is propagated from `C` to its refined concepts, and maps for `A`, `B1`, and `B2` are automatically created. The propagation proceeds from left to right in the refinement clauses, generating concept maps for `B1`, `A`, and `B2` in that order. When the map for `B2` is checked, the map for `A` is already in the environment and contains the definition for `void f()` propagated from `C`. The concept map for `B2` contains the same definition but since the two maps originate from `C` it is clear that they must be the same. The third premise checks for duplicate definitions. The fourth premise normalizes signatures for function definitions in the current concept map, ensuring consistent naming. Finally, the fifth premise requires that the set of function signatures for the function definitions in the current concept map, $\overline{f^a}^{\prec\prime}$, is equal to the difference between the set of required signatures $\overline{f^a}^{\prec}$ and the set of signatures of required functions $\overline{sig}^{\prec}$.

$$\frac{1.\ \overline{tydef}' = \{\, tydef \in \overline{tydef} \mid \neg\, id_\tau\,(tydef) \in \overline{id_\tau}\,(\overline{ty^a})\,\} \quad 2.\ \forall\, tydef \in \overline{tydef}' \bullet \mathit{find\text{-}defs}\,(id_\tau\,(tydef), \overline{tydef}^{\prec_1}) \approx tydef}{\overline{ty^a} \vdash \overline{tydef} \rightarrowtail \overline{tydef}^{\prec_1}}$$

Figure 2.10: Type definitions one-level compatibility

Compatibility checking of associated types, given by the second rule in figure 2.9, is different because associated types can be defined many times (clause 14.9.3.2 para. 4), as long as each definition names the same type, and they can be hidden by other associated types while functions cannot. First, a sequence of associated type requirements is extracted from the current concept. Next, in premise 2, the sequence of directly refined concepts is obtained and then, in premise 3, all the type definitions contained in the corresponding concept maps are extracted. Finally, in the fourth premise, the definitions are checked for compatibility, according to the rule in figure 2.10. Compatibility between type definitions $\overline{tydef}$, from the concept map designated by *cid*, and type definitions $\overline{tydef}^{\prec_1}$, from the concept maps for concepts directly refining *cid*, are compared in the context of associated types $\overline{ty^a}$ that are explicitly required in the concept *cid*. These associated types hide others with the same names in the refined concepts and, consequently, their definitions do not need to be compared with the ones in the refined concept maps; premise 1 in figure 2.10 accordingly eliminates hidden definitions from the compatibility check. All remaining definitions, $\overline{tydef}'$, must be identical to those in the directly refined concept maps. Accordingly (premise 2), the set of definitions in $\overline{tydef}^{\prec_1}$, that define a type with the same name, $id_\tau\,(tydef)$, as that defined by a *tydef* in $\overline{tydef}'$, must consist of only one element, *tydef* itself. Because this compatibility check is performed at every step of refinement and because every concept map must contain definitions for associated types from its own and from all concepts it refines, type definitions for associated types with the same name are guaranteed to be the same unless one definition hides another.

$$\frac{\begin{array}{lll} 1.\ cid? = cid_{impl} & 2.\ C \vdash cid \prec_1 \overline{cid}^{\prec_1} & 3.\ \overline{cid_i}^{\,i} = \{\, cid \in \overline{cid}^{\prec_1} \mid cm\,(M, cid) = None\,\} \\ 4.\ \overline{fdef}\,(C, M, cid_{impl}, cid) = \overline{fdef}^{\prec} & 5.\ \overline{\overline{ty^a}\,(C, cid_i) = \overline{ty^a}^{\prec}{}_i}^{\,i} & 6.\ \overline{\overline{f^a}\,(C, cid_i) = \overline{f^a}^{\prec}{}_i}^{\,i} \\ 7.\ normalize\,(C, cid, \overline{fdef}) = \overline{fdef}' & 8.\ \overline{\overline{tydef}_i = \{\, tydef \in \overline{tydef} \mid id_\tau\,(tydef) \in \overline{id_\tau}\,(\overline{ty^a}^{\prec}{}_i)\,\}}^{\,i} & \\ 9.\ \overline{\overline{fdef}_i = \{\, fdef \in \overline{fdef}'\, \overline{fdef}^{\prec} \mid sig\,(fdef) \in \overline{f^a}^{\prec}{}_i\,\}}^{\,i} & & \\ 10.\ \overline{cm_i = \texttt{concept\_map}\, cid_i\, \{\, \overline{tydef}_i \overline{fdef}_i\,\}}^{\,i} & & \end{array}}{icms\,(C, M, cid, cid?, \overline{tydef}, \overline{fdef}) = \overline{(cm_i, cid_{impl})}^{\,i}}$$

Figure 2.11: Generation of implicit concept maps

After the current concept map is checked for compatibility, concept maps for refined concepts are generated if they do not already exist (premise 8 in figure 2.7). Figure 2.11 shows how implicit concept maps are generated from a concept map that itself has been implicitly generated. Another rule, not listed in the figure, defines how concept maps are generated from an explicit concept map; these two rules differ only in the first premise. The rule in the figure handles concept maps that are implicitly generated by requiring the optional concept identifier of the implicit "parent" to be set to cid_{impl}. The identifier of the parent is then passed on to further generated concept maps, marking all generated maps with the explicit map that triggered the generation. For an explicit concept map, the first premise reads $cid? = None$, meaning that the concept

map currently considered was not generated from another concept map but that it was explicitly defined. Then, the concept identifier of the current map, *cid*, is passed to the generated concept maps as the "parent." In premise 2, the sequence of directly refined concepts is extracted and then, in premise 3, restricted to only the concepts for which no concept map is found in the environment M. Next, a sequence of all associated function implementations is extracted from all concept maps that are defined for any of the concepts refined by the current concept represented by *cid*. In premises 5 and 6, the sequences of required associated types and functions are extracted for each of the directly refined concepts in $\overline{cid_i}^i$. In premise 7, function definitions taken from the concept map initiating implicit declarations are normalized for consistent naming. In premise 8, a sequence of type definitions $\overline{tydef}_i$ is constructed for each of the implicit concept maps from the definitions $\overline{tydef}$ in the parent concept map. This is possible because, as we described earlier, the parent concept map contains type definitions for all associated types from all concepts it refines. Premise 9 defines how associated function definitions are generated. Premise 10 puts all the parts together in a sequence of implicitly defined concept maps to be returned.

In the next section we illustrate the rules by applying them to particular examples. Then we define template instantiation and type checking of instantiated templates, and using these definitions, we informally argue why separate type checking safety (theorem 1) holds.

5 Illustration and an Argument for Correctness

Up to now, we have introduced the semantics for separate type checking of constrained templates and implementations used to instantiate templates against the common interface provided by concepts. In this section, we provide an interpretation of separate type checking in a broader view, we illustrate the working of the rules on some simple examples, and we provide an educated argument of correctness, which forms a skeleton of a proof for theorem 1.

Traditionally, type checking is understood as a syntax-driven process where different constructs of a language are assigned different types to prevent incorrect programs, summarized by the famous quote "well-typed programs do not go wrong" (Pierce, 2002); this property is usually called *safety*. Which incorrect programs are prevented, depends on a particular type system but the notion of safety almost always means that a well-typed program cannot result in a run-time error or, from the view of type checking, reach a "stuck state," according to some *dynamic semantics*. The dynamic semantics determines the behavior of a program at "runtime" and it usually does not cover nonsensical cases, such as adding non-numbers. The safety is guaranteed in two steps, commonly known as *preservation* and *progress*. Progress means that a well-typed term is not stuck, that it either belongs to some predefined class of final values or that it can be further evaluated according to the dynamic semantics. Preservation means that every step of the evaluation of a well-typed term produces another well-typed term. In practice, type systems often provide other, higher-level guarantees and in most languages preservation actually guarantees that evaluating a well-typed term always produces a term of the same type.

In a language such as C++, the type checking process is complicated through the introduction of templates. Templates introduce a certain dynamic semantics into the compilation process, giving rise to "meta-programming" in C++ world. A template is first defined with parameters, and it is then only partially checked; as explained in sec-

tion 2, it is checked again when instantiated with particular arguments. A template and its arguments may type check perfectly separately, but fail to type check when combined through instantiation. Even worse, an instantiation of a template may succeed in the type checker but with undesired semantic effects at run time. The instantiation process, therefore, is a sort of dynamic semantics for C++ templates, the semantics that determines how fully-fledged programs are constructed from their parameterized skeletons.

Separate type checking guarantees that instantiation always[1] succeeds by providing a common interface between templates and their arguments in the form of *concepts*, which can be thought of as types of types (Dos Reis and Stroustrup, 2006). Separate type checking safety, then, is the guarantee that instantiation of templates will succeed, *given* that the templates check against the context of their concept constraints and the implementations fulfill all the demands of the appropriate concepts; if either templates or implementations are incorrect, errors are caught *before* instantiation. This greatly simplifies generic programming in C++, which, without concepts, often produces famously unreadable error messages during template instantiation.

The dynamic semantics of template instantiation is rather limited, boiling down to name lookup and substitution, given the modular nature of concept refinement hierarchies. Hence, there is not a clear progress and preservation property. Instead, the guarantee is that substitution errors in instantiation are prevented by forcing a common interface between template creators and clients. The goal of this section is to show by example and argue informally, that separate type checking, as given by our semantics, provides the safety of instantiation.

To argue for the correctness of the separate type checking safety theorem, later in this section, we provide a notion of instantiation and a definition of type checking of instantiated templates. Given these definitions, the claim of the theorem may be argued for case by case, showing that for every different case of concept hierarchies, constrained templates, and concept maps, an instantiated template will always type check after instantiation, given that it type checked against its constraints and that concept maps for the concepts in the constraints are provided. First, however, we give some extended examples of how the rules of our semantics work in practice.

5.1 Examples of Separate Type Checking

The examples in this section approximately follow the syntax given in figures 2.1 and 2.2. As defined there, concepts, templates, and concept maps do not have arguments, as if every parameterized construct had a single parameter only and this same type argument was used everywhere; this is sufficient since it is the lookup of associated types and functions that is our main concern.

In some of the examples we "bend" the syntax a bit by, for example, having two expressions in a template to show two different errors or repeating the same concept map with a different content to demonstrate different constellations that may occur.

5.1.1 A simple example

The simplest example of separate type checking involves a single concept with one requirement, a constrained template that makes use of the requirement, and a concept map that provides a definition for the requirement:

[1] Except for some caveat cases (Järvi et al., 2006).

```
1 concept Simple { typename t; }
2 template requires Simple test { obj t }
3 concept_map Simple { typedef int t; }
```

Separate type checking consists of three different checks: a check of the concept, `Simple`; a check of the concept-constrained template `test`, whether it refers to no other entities than the ones provided by `Simple`; and a check of the associated concept map, whether it provides the necessary implementations. In illustration, we simulate in the following how each of the check proceeds. In the interest of saving space, we refer to some rules just by their names; their full definition can be found in the technical report (Paper VI).

The definition of the concept `Simple`, to begin with, passes checking trivially, since the only requirement is that the concept has not yet been defined; the corresponding rule is P_INST_CONCEPT.

Next, the template `test` is checked, to ensure that all names used in its body are provided by the constraining concept `Simple`. The three most important rules for that check are the template instantiation rule, P_INST_TEMPLATE, the rule for type checking object creation, T_OBJ, and the rule for non-recursive type name lookup, FTREC_DIRECT. We briefly walk through each rule:

1. The template `test` is instantiated according to the template instantiation rule:

$$\frac{1.\, tl = \texttt{template}\,\texttt{requires}\,\overline{cid_i}^{\,i}\; tn\,\{\,e\,\} \qquad 2.\, \overline{cid_i \text{ defined in } C}^{\,i}}{\begin{array}{l} 3.\, tl \text{ not defined in } T \qquad 4.\, C;\emptyset;\overline{cid_i}^{\,i} \vdash e : \tau \\ 5.\, C;M;tl\,T \vdash P \Downarrow C';M';T' \end{array}}{C;M;T \vdash tl\,P \Downarrow C';M';T'}$$

   First, in premise 2, a check is performed whether the requirements, in this case the `Simple` concept, have been defined; the concept is simply looked up by its name in the concepts environment M. Next, premise 3 checks that a template `test` has not yet been defined. Premise 4 is the one in which most of the checks are performed. Given the current concepts environment M, which contains only the `Simple` concept, the empty set of variables $\emptyset$ (since there are no arguments in a template), and the sequence of requirements, which contains only the `Simple` concept, the expression `obj` τ contained in the template `test` is type checked.

2. Type checking is performed according to the rule for typing of object creation:

$$\frac{1.\, \overline{\mathit{find\text{-}rec}\,(C, cid_i, \tau') = \tau?_i}^{\,i} \qquad 2.\, rm?\,(\overline{\tau?_i}^{\,i}) = \overline{\tau} \qquad 3.\, \overline{\tau} \approx \tau}{C;\Gamma;\overline{cid_i}^{\,i} \vdash \texttt{obj}\,\tau' : \tau}$$

   First, the type of the created object must be looked up in the requirements. In this case, the name `t` is looked up in the concept `Simple` according to the rule FTREC_DIRECT for a lookup that succeeds without recursion into the refinement structure:

$$\frac{1.\, \mathit{find\text{-}scope}\,(C, cid, id_\tau) = \tau}{\mathit{find\text{-}rec}\,(C, cid, id_\tau) = \tau}$$

   The rule for *find-scope* asserts that, in this particular example, the τ found is `Simple::t`. Checking of the expression can then proceed to the second premise

that removes results of failed type lookups. In this example, there is only one successful lookup result and the premise has no effect. The final premise states that the set of discovered type specifiers must be a singleton set, which is true since the type specifier `Simple::t` is the only discovered type specifier. Since all premises are met, the typing rule successfully assigns type `Simple::t` to the expression in the template, since all names referred to are found in the context provided by the concept `Simple`. The constrained-template check, thus, succeeds.

Finally, the concept map for `Simple` is checked. In our example, the concept map must provide a (type) definition for `t`. The rule listing all the checks applied, is P_INST_CMAP_IMP:

$$\frac{\begin{array}{ll} 1.\ icm = (cm, cid?) & 2.\ cm = \texttt{concept\_map}\, cid\, \{\overline{tydef}\,\overline{fdef}\,\} \\ 3.\ cm \text{ not defined in } M & 4.\ tydefs\text{-}check\,(C, cid, \overline{tydef}) \\ 5.\ tydefs_{=}\,(C, M, cid, \overline{tydef}) & 6.\ fdefs\text{-}check\,(C, cid, \overline{fdef}) \\ 7.\ fdefs_{=}\,(C, M, cid, cid?, \overline{fdef}) & 8.\ icms\,(C, M, cid, cid?, \overline{tydef}, \overline{fdef}) = \overline{icm_i}^{\,i} \\ 9.\ C; icm\,M; T \vdash \overline{icm_i}^{\,i}\, P \Downarrow C'; M'; T' & \end{array}}{C; M; T \vdash icm\, P \Downarrow C'; M'; T'}$$

This is the rule for checking implicit concept maps. Note that, initially, we are in the case of *explicit* concept maps, but the rule for checking explicitly defined maps simply forwards the check to the rule for implicit maps with *None* for generating parent. The first two premises set up the variables, the third premise is true since no concept map has been defined yet for the `Simple` concept. The fourth premise checks that definitions are provided only for type names required by the concept for which the map is defined, that the definitions are well-formed, which in our system means that one of the built-in types is used for the definition, and that the definitions are unique; all of these conditions are met in the concept map for the `Simple` concept.

This completes the separate type checking process, and the template `test` now can safely be instantiated according to the separate type checking safety theorem.

5.1.2 Function and type lookup, correct negatives

The next code listing consists of a (non-refining) concept with two templates and two concept maps. The example illustrates the combination of function lookup with type lookup. It also shows two erroneous cases: a function template that is not properly concept-constrained, and a type for which no proper concept map is defined. In the first case, it is the checking of constrained templates part of separate type checking that correctly fails, and, in the second case, it is checking of concept maps.

```
1  concept A {
2    typename t;
3    int f(t);
4  }
5
6  template requires A test_ok {
7    f(obj t)
8  }
9
10 template requires A test_error {
11   f()                        // error, no \texttt{f()}
12   obj tt                     // error, no type \texttt{tt}
13 }
14
15 concept_map A {              // This concept map checks successfully
```

```
16   typedef int t;
17   int f(v : t) { f(v) }
18 }
19
20 concept_map A {              // This concept map contains errors
21   typedef t t;                // t is not a built-in type
22   int f(v : t) { obj bool }  // the definition does not type
23 }
```

We first simulate the checking of the two templates `test_ok` and `test_error`. Both of the templates require the concept `A`, but the first template contains an expression that passes type checking, while the second template contains two examples of an expression that does not check. Both templates are checked according to the rule P_INST_TMPL, and, as in the previous example, the non-trivial part of the check is the type checking of the template expressions, performed in premise 4 of that rule.

1. The type checking of the expression `f(obj t)`, in the first template, begins with the rule for checking function application, T_APP. In the first premise, the functions named `f` are found in the template context, in this case the concept `A`; the lookup results in one function signature `int f(A::t)`, with argument type `A::t` rather than just `t`, to keep track of which `t` the function applies to if there is more than one. The second premise asserts that the set of discovered function signatures does not contain duplicates, preventing ambiguities if identical signatures are found. The third premise types the arguments of the function, `obj t`, proceeding in exactly the same way as in the first, simple example. In premise 4, overload resolution is performed, matching the types of the parameters with the types of the arguments; here, the overload set contains one signature `int f(A::t)` and the function is applied to the argument of type `A::t` (the type of `obj t`), giving `int f(A::t)` as the result of the resolution. The last premise extracts the return type of the chosen function and assigns it to the whole function application. With that, the template `test_ok` checks.

2. The second template, `test_error`, contains two incorrect expressions—concept `A` provides neither a function of the signature matching the function call `f()` nor a type `tt`. We discuss each error separately and show how the check fails. The function call `f()` first finds the same overload set as in the previous template, there are no arguments to type. But overload resolution does not find any function signature matching the function call and, thus, premise 4 gets "stuck," preventing the conclusion from being reached. Being stuck corresponds to an error in our semantics; there is no explicit error handling. The second expression fails on the third premise of the rule T_OBJ, since the lookup of type `tt` returns an empty sequence of results that is not set equal to a singleton set.

   In either case, the check correctly fails, thus one of the parts of separate type checking does not hold and, as a consequence, the safety guarantee cannot be given—concept checking has prevented an instantiation error.

Next, the code snippet contains two example concept maps, one that checks—since it properly implements concept `A`—and one that does not, since it incorrectly implements `A`, with an ill-formed type definition and a function that does not type check.

The first non-trivial check is performed in premise 4 of P_INST_CMAP_IMPL, which checks the correctness of type definitions, and it succeeds as in the earlier example. This example also contains a function definition, which is checked for correctness in premises 5 and 6. The function definitions check, performed according to the rule

FDEFS_CHECK, first types the body of the function definition, `f(v)`, according to the function application rule and checks that the return type is the same as the declared return type of the function; here, the recursive application `f(v)` has the correct type. Next, the uniqueness of the definitions is checked. The next steps of P_INST_CMAP_IMPL check the compatibility of the function definitions with the existing definitions and generate implicit concept maps; these steps are trivial in this case since there are no refinements.

The second concept map does not pass the same checks. The type definition `typedef t t;` fails the well-formedness check in premise 3 of TYPEDECLS_CHECK that limits the definition types to built-in types, which `t` is not (it is a type identifier only). The function definition `int f(v : t){ obj bool }` fails because the function body is of type `bool` but the return type of the function is `int`. Consequently, as in the case with the template `test_error`, the type checking safety guarantee correctly cannot be given, but in this case, a different premise is violated.

5.1.3 Refinement

The following examples illustrate the working of the rules in more complicated cases involving refinement. The following snippet of code introduces three simple concepts, each containing one requirement for an associated type named `t`.

```
concept A { typename t; }
concept B { typename t; }
concept C { typename t; }
```

The next snippet introduces a refinement hierarchy of concepts, built on the top of the three simple ones. The hierarchy illustrates simple refinement, as in the concept `E`; a more complicated case of the concept `F`, which adds its own requirements shadowing the requirements in the concept `C`; and, finally, the complex refinements of the concepts `G` and `H`, which gather requirements from concepts `E` and `F`, with the concept `H` adding its own requirement for the associated type `t`.

```
concept E refines A, B { }
concept F refines C { typename t;  int f(t); }
concept G refines E, F { }
concept H refines E, F { typename t; }

template requires G test1 {
  f(obj t)  // correct, f(A::t)
}

template requires H test2 {
  f(obj t)  // error, f(H::t) not available
            // have f(C::t)
}
```

The two constrained templates `test1` and `test2` contain an identical expression, which passes the check in one but not in the other, due to the differences in the name lookup in the requirements. The relevant rules here are the rule P_INST_CONCEPT for typing of function applications, the rule FIND_FUN_REC for recursive lookup of functions, and the normalization rule N_FDECLS_REC for returning correctly qualified argument types. We look separately at the processing of each of the two templates.

1. The typing of the expression `f(obj t)` in the first template begins with typing of the function application, in the rule T_APP. The first step of finding the available signatures differs from the previous examples: even though the function `int f(t)` is defined in the concept `C`, the lookup returns `int f(A::t)` as the only

available signature. The lookup of functions named `f`, in the first premise of the rule T_APP, is given by the rule FIND_FUN_REC. First, the set of refined concepts of `G` is computed, and then, in premises 2 and 3, the scopes of `G` and its refined concepts are searched for functions named `f`. The declaration of function `f`, `int f(t);`, is found in the scope of `C` and then it is *normalized* in the context of `C` (premise 5 of the rule FIND_FUN_SCOPE), yielding a signature `int f(C::t)`. The normalization process (rule N_FDECLS_REC) looks up every type in the signature of the function in the normalization context, marking the types by the scopes in which they were found; since the concept `C` contains a requirement for type `t`, the result of normalization in the context of `C` is a qualified type `C::t` (the result of the lookup of the unqualified name `t` in the scope of `C`).

The next step of function application typing in the rule T_APP normalizes the signature again but this time in the context of the concept `G`. The first normalization records the result of type name lookup in the scope of `C`, where the function signature was found, and the second normalizations checks to which equivalence class the type belongs to in the context of `G`. In this particular example, three types `t` are reachable from `G` through the refinement hierarchy: `A::t`, `B::t`, and `C::t`. Accordingly, the second normalization returns `A::t` as the unique representative of the equivalence class to which `C::t` belongs in the context of `G`. Going back to the function application check, the overload set now contains `int f(A::t)`.

Next, the argument `obj t` is typed (rule T_OBJ) by looking up `t` in the scope of `G`. Again, type lookup rule picks the unique representative `A::t` from the three visible `t`s, resulting in a successful completion of the function application check as the signature `int f(A::t)` in the overload set matches the type of the argument `obj t`. This completes the checking of template `test1`, assuring that the expression in the template can be checked, provided the interface represented by the concept `G`.

2. The second template, `test2`, is similar to `test1` but is constrained by the concept `H` instead the concept `G`. The concept `H` introduces its own type `t` that shadows the other `t`s: the equivalence class for the lookup of `t` in the context of `H` is only `H::t`, in difference from the lookup in the concept `G`. Consequently, the second template fails since the type of the expression `obj t` is `H::t` (`t` looked up in `H`) and the function available in the overload set has the signature `int f(C::t)` (the second normalization stage does not change the signature as in the first template because `C::t` does not belong to a larger equivalence class). Separate type checking of the second template fails, because the expression `f(obj t)` is incorrect in the context of the constraining concept `G`.

5.1.4 Implicit and explicit concept maps, compatibility

The examples in this subsection illustrate handling of implicit vs. explicit concept maps and, at the same time, of compatibility checks. We demonstrate in particular how implicit concept maps are created, which extend the concept-map environment. Assume the previous refinement hierarchy with the concept `E` changed as follows:

```
1 concept E refines A, B { int f(t); }
```

We note in passing that this change makes the template in the previous example 5.1.3 invalid because two different signatures `int f(A::t)` are now discovered, one in the

concept E and one in the concept F—this situation is not allowed by premise 2 in rule T_APP. In the current example, however, we focus on the different ways in which concept maps can be defined for the concept hierarchy from the previous example, with the modified concept E. The following snippet shows the maps for the concepts E and G defined explicitly, and the maps for concepts A, B, F, and C defined implicitly.

```
1 concept_map E {  // implicit maps for A and B
2   typedef int t;
3   int f(var : t) { obj int }
4 }
5
6 concept_map G { typedef int t; }  // implicit maps for F and C
7 // typedef compatible
8 // definition of \texttt{f(t)} comes from the concept map for \texttt{E} and
9 // is propagated to \texttt{F} and \texttt{C}
```

We first discuss the check of the concept map for E, then the one for G.

1. The check of the concept map for the concept E proceeds similarly to the one for previous concept maps, according to the rule P_INST_CMAP_IMPL. In difference to the previous examples, however, there are refined concepts with missing concept maps and these are generated in premise 8, according to the rule IMPLICIT_CMAPS_NONE (rather than according to IMPLICIT_CMAPS_SOME, since the map for E is explicit). The first premise of the rule asserts that the generating parent concept map is not itself an implicit map, which is true in the case of the concept map for E. In premise 2, the set consisting of A and B, the concepts directly refined by E, is extracted. Premise 3 limits that set to concepts that have no concept map defined yet, which is true for both A and B. In premise 4, function definitions from maps for refined concepts are extracted, but since none of the concepts refined by E has a map defined yet, the sequence of definitions is empty.

   With premise 5, synthesis of the implicit concept maps starts. In this premise, the associated types required by each of the refined concepts are extracted, giving a singleton set containing t for both concepts A and B refined by E. A similar extraction of the associated functions required by A and B, in premise 6, results in an empty sequence for each of the two concepts. Before moving on further, the function definitions of the generating concept map, E in this case, are normalized to assure that the meaning of type names is captured in the context of E: the definition `int f(var : t){ obj int }` is normalized to `int f(var : A::t){ obj int }`. Premises 8 and 9 pick the definitions required for the concepts A and B from the definitions available in the map for concept E (including the empty sequence of function definitions, $\overline{fdef}^{\prec}$, from the existing maps for the refined concepts). Since the order of the refinement must be preserved, a concept map for A is created first and the map for B is created second with a single definition `typedef int t;` in both of the maps. Back in the rule P_INST_CMAP_IMPL, these maps are prepended to the rest of the program in premise 9 and processed accordingly. Including the processing of implicit concept maps, processing of the concept map for the concept E, on lines 1–4, thus results in 3 new concept maps in the environment.

2. Processing of the concept map for G, on line 6, involves compatibility checking and sibling-to-sibling propagation of function definitions. Premises 1 through 4 of the rule P_INST_CMAP_IMPL check as in the other concept maps in the previous examples. In premise 5, however, the type definitions are checked against

the existing definitions in the concept map for the concept `E`, according to the rule TYPEDECLS_COMPAT. The first three premises set up the check by extracting the sequence of required associated types for `G`, the sequence of concepts directly refined by `G` (the concepts `E` and `F`), and the sequence of all type definitions from the directly refined concepts (`typedef int t;` from `E`). The last premise invokes the actual compatibility check, defined in ASSTS_COMPAT. The first premise of this check eliminates type definitions for any associated type directly required by the concept `E`, to avoid forcing compatibility in the case that the requirements in the current concept shadow the requirements in the refined concepts. The second premise requires that any definition of non-shadowed type requirements in the map for `E` match the definitions for the refined concepts; this is true since both the maps for `G` and `E` contain the same definition `typedef int t;`. The compatibility check completes successfully, guaranteeing that types that are treated as equivalent in constrained templates have identical definitions in the corresponding concept maps.

Since the map for `G` does not contain any function definitions, the next two steps of concept map checking in P_INST_CMAP_IMPL succeed trivially. In premise 8, concept maps for the concepts `F` and `C` must be generated, concept maps for the concepts `E`, `A`, `B` are already in the environment. The generation of concept maps proceeds in a similar fashion as in the map for the concept `E`, with the exception that the concept map for `G` does not contain the definition of a function `int f(t)` required by the concept `F`. The missing definition is taken from the concept map for `E`, as specified in the rule IMPLICIT_CMAPS_NONE. In premise 4 of this rule, function definitions are extracted from the existing maps for the refined concepts of `G`. The concept map for `E` contains a definition `int f(var : t){ obj int }`, which is transformed by the normalization process in the rule DFDECLS_NONE (premises 6 and 7) to `int f(var : A::t){ obj int }`. The normalized definition is inserted into the implicitly generated map for `F`. The definition for the associated type `t` is taken directly from the map for `G`, with the earlier check in premise 4 of the rule P_INST_CMAP_IMPL guaranteeing that the definition is available. Thus generated concept map for `F` is inserted into the program, in premise 9 of P_INST_CMAP_IMPL. When it is checked itself, all the definitions inserted from the previously defined concept maps are checked again, this time in the context of `F`. The definition `int f(var : A::t){ obj int }` pushed from the map for `G` fails type checking because `A::t` cannot be looked up in the context of the concept `F`, which does not refine the concept `A` (premise 2 of the rule FTREC_REC_SCOPE does not hold). In principle, such situation must be rejected since the source of the propagated function definition is not known and it is not possible to decide if a type name can be renamed to match the local context.

5.2 Correctness Argument

Theorem 1 states that assuming a program *separately type checks* (assumption 1), a constrained template is defined in the program (assumptions 2 and 3), and a concept map is defined for each of the requirements of the template (assumption 4), the template can be safely instantiated. Consequently, the goal of the correctness argument is to analyze the possible cases of instantiation, showing that indeed no instantiation errors occur.

The argument involves two parts. First, we need to establish the ingredients of the

hypothesis, template instantiation and type checking of instantiated templates. Given a definition of instantiation and of type checking of instantiated templates, we can, case by case, work our way backwards, showing that anything that could go wrong has been eliminated earlier, when the program was separately type checked.

5.2.1 Template Instantiation

Template instantiation puts the parameterized code of templates together with the template arguments, producing an executable piece of code. For constrained templates, this means plugging into the templates the implementations given in concept maps. The following definition gives the meaning of instantiation in our semantics.

Definition 3 (Template Instantiation)**.** Assume an environment $C; M; T$ and a template $\texttt{template requires}\,\overline{cid_i}^{\,i}\ tn\,\{\,e\,\}$. Then a template is instantiated as follows:

1. Substitute types in the template body by their bindings in the environment C: every occurrence of a type identifier in e is replaced by the result of its lookup in the requirements $\overline{cid_i}^{\,i}$, i.e., e' is the expression e with every type identifier τ replaced by τ' such that $\mathit{find\text{-}rec}\,(C, cid, \tau) = \tau'$ (FTREC_ family of rules).

2. Bind types and functions to their implementations provided in the concept maps environment, M.

   (a) For each requirement cid_i extract its corresponding, associated function definitions, i.e., the set of function definitions, $\overline{fdef}_i$, such that the following holds $\overline{fdef}\,(C, M, None, cid) = \overline{fdef}_i$ (rule DFDEFS_NONE). Then normalize each extracted definition in the context of the concept cid_i (normalization replaces every type name occurrence by the unique representative of the equivalence class to which it belongs in the context of cid_i).

   (b) For each requirement cid_i extract its corresponding, associated type definitions.

      i. Find all concepts $\overline{cid_k}^{\,k}$ refined by cid_i such that $C \vdash cid \prec \overline{cid_k}^{\,k}$.

      ii. For each of these concepts, extract its associated type definitions, $\overline{tydef}_k$, such that $\overline{tydef}\,(M, cid_k) = \overline{tydef}_k$.

      iii. For each associated type definition, $\texttt{typedef}\,\tau\,id_\tau \in \overline{tydef}_k$, generate a tuple of a qualified type name and the corresponding implementation $(cid_k :: id_\tau, \tau)$.

   (c) Concatenate the extracted definitions into a sequence of associated function definitions, $\overline{fdef}$, and a sequence of associated type definitions, $\overline{\tau_{defs}}$.

The resulting instantiated template is of the form $\overline{\tau_{defs}}\,\overline{fdef}\ tn\,\{\,e'\,\}$.

In summary, an instantiated template, $\overline{\tau_{defs}}\,\overline{fdef}\ tn\,\{\,e'\,\}$, loses the constraints of the constrained template from which it is instantiated and instead gets the implementations from the concept maps corresponding to the requirements. The implementations give the template its own "mini-environment," represented by the sequences $\overline{fdef}$ and $\overline{\tau_{defs}}$. In addition, the template body of the constrained template, e, is "flattened out" into the instantiated body, e', by substituting type names in e with the results of their lookups in the environment C. One can consider such template as "an executable" piece of code, assuming the environment is well-formed (this condition is the subject of the rest of this section). The following example illustrates the process of instantiation.

```
concept A {typename t;}
concept B refines A {int f(t)}
concept_map B { typedef int t; int f(t) { obj int } }
template requires B test { f(t) }
instantiate test;
```

There are two concepts, `A` and `B`, a concept map for `B`, and a template `test`, instantiated in the last line of the program. The instantiated template contains two type definitions, `(A::t, int)` and `(B::t, int)` (the definition in `A` is propagated from `B`), and one function definition `int f(A::t){ obj int }`. The expression of the instantiated template is `f(A::t)`. The instantiated template has the form:

```
(int f(A::t) {obj int}) ((A::t, int), (B::t, int)) test { f(A::t) }.
```

The first parentheses demarcate the sequence of function definitions, the second parentheses surround the sequence of type definitions, next comes the template name, and, finally, the substituted expression from the constrained template.

5.2.2 Type Checking of Instantiated Templates

To establish correctness of instantiation one needs a correctness criterion. Theorem 1 establishes type checking of an instantiated template as the criterion for correctness: the goal of separate type checking is to guarantee that when constrained templates and implementations come together, there will be no type errors. This correctness criterion alleviates the problems with templates in the original C++ language: in C++ without concepts, incorrect instantiation causes type errors in the instantiated templates, but concept constraints make these errors impossible, guaranteeing separate type checking safety. The following definition specifies type checking of constrained templates.

Definition 4 (Type checking of instantiated templates)**.** Given an instantiated template $\overline{\tau_{defs}}\,\overline{fdef}\,tn\{e'\}$, it is type checked in the same manner as expressions in constrained templates (the T_ family of rules) with the following three exceptions:

1. In rules T_OBJ and T_APP, types are looked up in the implementation environment, $\overline{\tau_{defs}}$, instead of in the requirements of a constrained template.
2. In rule T_APP, functions are looked up in the implementation environment, $\overline{fdef}$, and instead of premise 2, which checks for uniqueness ($distinct\,(\overline{\overline{sig_i}}^{\,i})$), duplicates are removed from the overload set ($rdup\,(\overline{\overline{sig_i}}^{\,i})$).
3. In rule T_APP, an additional premise is made that the function definition found by overload resolution must type check, i.e., given the parameters of the function, its body types to the return type of the function (as in FDEFS_CHECK, premise 1). Again, the check is performed in the implementation environment of the instantiated template, rather than in the concept hierarchy environment.

The type checking relation for instantiated templates has the form $\overline{\tau_{defs}}\,;\overline{fdef}\,;\Gamma\vdash e:\ \tau$ instead of $C;\Gamma;\overline{cid}\vdash e:\ \tau$ for type checking in constrained context.

Definition 4, thus, states that to type check an instantiated template, its expression must be checked against the available implementations. The type checking process is based on the rules for type checking in constrained templates (the T_ family of rules), differing in name lookup and in typing of function applications. Instead of looking types up in template requirements, in an instantiated template, type names can

be looked up only in the sequence of type definitions (definition 4, item 1). Since the sequence consists of pairs of a type name and its implementation, it can be treated as a map from type names to implementations (due to the way it is created, all mapped type names are fully qualified, e.g., `A::t`). As for typing of function applications, the sequence of available function definitions constitutes the overload set in which function applications are to be checked. In difference to type checking of constrained templates, however, the overload set may contain multiple definitions of associated functions, due to propagation of definitions in implicit concept maps (definition 4, item 2). In general, multiple function definitions with the same signature are considered ambiguous in C++. Yet, in the case of type checking of instantiated templates, we know that any two identical functions have the same source of origin, i.e., they are copies propagated from the same original definition, and, thus, we do not consider this situation ambiguous. The following lemma states the assumption of unambiguous function definitions.

Lemma 1 (Non-ambiguous duplicate function definitions)**.**
Assuming a separately type checked program, duplicate function definitions in an instantiated template have the same origin if the function is called in the template's expression, i.e., they were all copied from the same function definition, or the set of definitions contains only the original definition and its copies.

This lemma is not strictly necessary to argue for the correctness of theorem 1, but it makes our choice of removing duplicates plausible. The argument for correctness of lemma 1 takes the following shape:

> **Non-ambiguous duplicate function definitions.**
>
> **Given:** A separately type checked program P_{user} such that $P_{user} \Downarrow C; M; T$, a template $\overline{\tau_{defs}}\,\overline{fdef}\; tn\,\{\, e \,\}$ instantiated in the program P_{user}, and a signature $\tau f\,(\overline{\tau_i}^{\,i})$ of a function called in e.
>
> **Show:** Duplicate function definitions in $\overline{fdef}$ necessarily have the same origin if they are called in the expression e.
>
> **Argument:** The argument depends the on the assumption in the lemma that the function call in the instantiated template is generated from the corresponding call in a constrained template—this guarantees that the call was type checked in the constrained template by assumption of separate type checking. Since the instantiated template is created from a constrained one and function definitions are extracted from the concept maps for template requirements, it is sufficient to show that said concept maps do not contain definitions of functions $\tau f\,(\overline{\tau_i}^{\,i})$ having different origins. We make a distinction between duplicate definitions originating from multiple requirements and those originating from a single requirement. We first show that the case of multiple requirements origins does not occur and then we consider the case of single requirement origin.
>
> **Case analysis:**
>
> 1. (Multiple requirements origin) For multiple requirements, expression typing in constrained templates requires the discovery of the function in only one of the requirements (P_INST_TMPL, premise 4). Consequently, the signature $\tau f\,(\overline{\tau_i}^{\,i})$ comes from only one of requirements (specifically, the requirement and its refined concepts). It is hence sufficient to consider only the next case of single requirement origin.

2. (Single requirement origin) In this case, it is sufficient to show that in the concept maps for the refinement hierarchy of the originating requirement, there is either a unique definition for the function in question or all definitions have been propagated from the same source. The argument follows the structure of function definition checking in concept maps, which proceeds differently in explicit and implicit maps.
   (a) Explicit case.
      i. The function definition is provided in the concept map. Then, there is no conflicting definition in the same map (premise 2 of FDEFS_CHECK) and there are no other definitions in refined concept maps (FDEFS_COMPAT, premise 5).
      ii. The function definition is provided in a refined concept map. Then, there are no duplicate definitions (FDEFS_COMPAT, premise 3) and the same definition is copied to implicitly generated concept maps, if any (IMPLICIT_CMAPS_SOME, recursively).
   (b) Implicit case. Existing definitions are allowed only if they originate from the same source (FDEFS_COMPAT, premise 2 and DFDECLS_SOME, premise 5).

5.2.3 Safety

Given the notions of instantiation and type checking of instantiated templates, the argument for the *safety* of separate type checking can be given. To recall theorem 1, instantiation of a separately checked template and, later, its type checking must succeed. Thus the argument is split into two parts. First we argue that the instantiation process succeeds and, then, that an instantiated template type checks.

Instantiation. By analyzing the two cases in definition 3, it is clear that template instantiation can only fail in two ways: either some of the concepts used in the template requirements do not have concept maps defined, or, during substitution of type names with the results of their lookup, some of the lookups do not succeed. Yet, given a separately type checked program, none of these two conditions is possible.

Instantiation safety.

Given: The assumptions of theorem 1: a correct program, a constrained template, and a concept map for each of the concepts named by refinement (note that the theorem does not guarantee concept maps for concepts refined by requirements). In this argument, we reuse the names introduced in the theorem.

Show:

1. A concept map is defined in M for each requirement cid_i and for each concept refined by one of the requirements, i.e., for each concept in the sequence $\overline{cid} = \overline{\overline{cid}_i}^{\,i}$ such that $\overline{C \vdash cid_i \prec \overline{cid}_i}^{\,i}$.
2. Every type name in the body of a constrained template, e, can be successfully looked up given the requirements $\overline{cid}$ and the concept environment C.

Argument: To show that all necessary concept maps exist, it is sufficient to show that maps for all concepts refined by requirements exist in the environment (qua assumptions, concept maps for requirements themselves are defined). From the concept map checking rule (P_INST_CMAP_IMPL) it follows, due to implicit map creation (the IMPLICIT_CMAPS_ family of rules), that concept maps for refined concepts are defined, either explicitly or implicitly; the first claim thus holds. For the condition of safe type lookup, it needs to be shown that recursive type lookup in the concepts environment C succeeds given the context of requirements $\overline{cid}$. From constrained template checking (P_INST_TMPL), it follows that the template expression checks in the context of requirements (premise 4). Then, by type checking in constrained templates (T_OBJ and T_APP), every type name has been successfully looked up when the template was processed, and this lookup must succeed in the instantiation process as well.

It is important to note that the argument for instantiation safety makes no guarantees that type checking of instantiated templates automatically succeeds; there is a possibility that an instantiated template is nonsensical. Yet, the next part of the safety argument shows that type checking of an instantiated template must surely succeed given the assumptions of theorem 1.

Type checking of instantiated templates. From definition 4, the definition of type checking of instantiated templates, the conditions for successful type check can be summarized as follows: if a type name is looked up, there must be a single corresponding definition (T_OBJ), and if a function call is encountered and its arguments type check, there must be a matching function definition to call in the instantiated template function definitions, and the definition found must type check. Next, we argue why these conditions are always met.

Separate type checking safety.

Given: The assumptions of theorem 1: a correct program, a constrained template, and a concept map for each of the concepts named by refinement (in this argument, we reuse the names introduced in the theorem). In addition, a successfully instantiated template $\overline{\tau_{defs}}\,\overline{fdef}\;tn\,\{\,e\,\}$.

Show: It must be shown that in each case of the rules for type checking of instantiated templates the assumptions guarantee success. Object creation and function application are the only two non-trivial cases:

1. Object creation, `obj` τ, types if the type name τ can be successfully looked up in the implementation environment of the instantiated template.
   (a) Show that τ has only one definition in $\overline{\tau_{defs}}$ and is defined to be a built-in type (i.e., a concrete, implementation type).
2. By analysis of function call typing rules, a call $f\,(\overline{e_i}^{\,i})$, types if the following conditions can be shown to hold.
   (a) Every argument expression, e_i, type checks.
   (b) Overload resolution for the function call finds an appropriate function definition, $fdef$, in the sequence of available definitions, $\overline{fdef}$.

(c) The function definition *fdef*, returned by the overload resolution, type checks.

Argument: For each of the above cases, type checking of the constrained templates and checking of the corresponding concept maps guarantee that all necessary definitions exist and are correct.

Case analysis:

1. Type checking of object creation, `obj` τ, in the instantiated template succeeds because the expression type checks in the corresponding constrained template, type lookup in the constrained template resolves to the same unique type as type lookup in instantiation, and because concept map checking guarantees that the appropriate type definition is available in the implementation environment of the instantiated template.
   (a) The expression `obj` τ' in the constrained template, from which the expression in the instantiated template is generated, is typed in the context of the constrained template requirements (rule P_INST_TMPL, premise 4). From constrained context typing, it follows that the type identifier τ' successfully resolves to a unique associated type name (T_OBJ, premise 3).
   (b) Since the lookup used in instantiation is performed in the same context as in the constrained template, the type name τ in the instantiated template is the same as the result of type lookup in the constrained template.
   (c) Next, it follows by constrained context typing (performed in T_OBJ, premise 1) that lookup always returns a qualified type name of the form $cid' :: id_\tau$ because the only successful base case of recursive type lookup gives a qualified type name (the base case is defined in FTREC_DIRECT). By the previous step, we know that $\tau = cid' :: id_\tau$ and by constrained context typing and type lookup we know that cid' must be refined by one of the requirements. We have shown before (see instantiation safety) that a concept map for cid' must exist and by concept checking rules and type lookup rules we know that the map must contain a correct definition for id_τ. By instantiation rules, we know that this definition is inserted into the implementation environment of the instantiated template and thus object creation must type in the instantiated template.
2. As said before, three conditions must be met for a function call $f\,(\overline{e_i}^{\,i})$ to type.
   (a) To show that every argument expression e_i types, one has to consider two non-trivial cases of typing in instantiated templates. Each argument expression can itself be a function application or object creation. For object creation, the previous argument applies and for function call, the argument is applied recursively.
   (b) Overload resolution for the function call finds an appropriate function definition, *fdef*, in the sequence of available definitions, $\overline{fdef}$. The line of argumentation here is similar to the argument

for lemma 1. Following that argument and the definition of type checking of instantiated templates (definition 4), shows that the necessary function definition exists.

(c) The function definition *fdef*, returned by overload resolution, type checks. This is true, since every definition in $\overline{fdef}$ is extracted from a checked concept map, the definition must check (rule FDEFS_CHECK) and since it was normalized (definition 3), each type referred to in the definition has a corresponding implementation in the instantiated template.

This argument shows that separate type checking guarantees safe and correct instantiation, the property we term *separate type checking safety*. Separate type checking provides a mechanism to insert a common interface between the developer and the user of a generic library, guaranteeing that the implementations by the user and the templates by the developer can be safely put together. Furthermore, concepts and concept maps allow one to structure and modularize the expression of the interface in concept hierarchies, and to correspondingly modularize implementations in concept maps. Instantiation connects both structures, giving an executable piece of code that is correct.

6 Interpretation of the Informal Definition

While our semantic definition is as faithful as possible to the concept wording (Gregor et al., 2008e), we have encountered informal definitions that were either ambiguous or seemed to be inconsistent with the understood design goal. In this section we outline the important differences between our definition and the wording.

6.1 Name Lookup

Clause 3.3.10 para. 1 states that a name can be hidden by a declaration in a refining concept. While it is not stated explicitly which names can be hidden, the previous publications on concepts and the authors' experience suggest that associated type names can be hidden by associated type names in refining concepts. On the other hand, the formulation of name lookup in concepts, specifically clause 14.9.3.1 para. 4, 4th bullet, states that associated type names are collected from the scopes of *all* refining concepts and that for the name lookup to succeed they all must be the same type, as if there was no hiding. Our semantics assumes hiding, as prescribed in clause 3.3.10 para. 1, and terminates the search through the refined scopes once a matching type is found: the first rule in figure 2.4 does not continue if a type was found while the second rule continues the search since the first premise assures that no matching type was found in the currently searched scope.

Furthermore, clause 14.9.3.1 para. 4 states that if there is more than one associated type with the same name in a refinement hierarchy, these types must be explicitly declared to be the same—through `SameType` constraints (a built-in, compiler-supported concept)—for name lookup to succeed; if these constraints are missing then name lookup fails with ambiguity. Our semantics implicitly assumes these constraints; the motivation for that assumption is given in the next subsection.

6.2 Implementation Binding

Compatibility and definition propagation. Clause 14.9.3.2 para. 4 states that a definition in a concept map for a refining concept is compatible with a definition in a refined concept only in three cases:

- the definition in the refining concept map is explicit and the definition in the refined map is implicit,
- the definition in the refined concept map is explicit and the definition in the refining concept map is implicit,
- the definitions satisfy an associated type requirement and both definitions explicitly name the same type.

When formalizing this definition we have discovered that the definition in the wording may allow cases that are not compatible and that it makes defining certain concept maps impossible. Consequently, our semantic rules differ from the wording.

The first compatibility difference is that we restrict associated types definition to the third case given above. That is, the second rule in figure 2.9 requires, in the third premise, that every concept map provide definitions for *every* associated type requirement in the concept hierarchy. Since every concept map explicitly defines all required associated types, the compatibility check boils down to the third rule, which requires that the definitions for the same associated type requirement name the same type. Note that while the wording does not take hiding into account, our definition in figure 2.10 eliminates hidden types from compatibility check.

The compatibility rules in the wording, specifically the second rule, allow the situation where concept maps for refined concepts provide differing definitions for associated functions with identical signatures. This compatibility breach, coupled with the name lookup rules given in clause 14.9.3.1 and reflected in our semantic rules listed in figure 2.3, means that one definition is chosen out of potentially conflicting associated function definitions, according to the depth-first traversal discovery rules discussed in section 4. Our semantic definition allows *only one* explicit definition for an associated function in refined concept maps, as seen in figure 2.9, first rule, premise 3. Consequently, there cannot be conflicting definitions. In addition, while the wording allows only one level of definition propagation, from an explicit definition to an implicit one, our rules allow definitions to be propagated further. In figure 2.11, associated function definitions are taken from the sequence $\overline{fdef}\,\overline{fdef}^{\prec}$ meaning that definitions will be propagated downwards but also sideways in the refinement hierarchy. The wording allows propagating downwards, from an explicit definition in a refining concept map to an implicit definition in a refined concept map, while our definition allows taking both implicit and explicit definitions in refined concept maps and propagating them to other refined concept maps. If, in figure 2.11, we changed the sequence of available definitions to $\overline{fdef}$, our semantics would match the semantics in the wording. However, the semantics in the wording may be too restrictive; in the following code listing we give an example of a situation where concept maps cannot be defined in the semantics given by the wording.

```
1 concept A<typename T>              { void f(); }
2 concept B<typename T>              { void f(); }
3 concept C<typename T>              { void f(); }
4 concept E<typename T> : A<T>, B<T> { }
5 concept F<typename T> : B<T>, C<T> { }
6
```

```
concept_map A<int> { void f() {} }
concept_map B<int> { void f() {} }
concept_map C<int> { void f() {} }
concept_map E<int> { }                // error
concept_map F<int> { }                // error

concept_map E<float> { void f() {} }
concept_map F<float> { void f() {} }  // error
```

Given the concept hierarchy in the example there is no way to define a concept map for all of the concepts. The concept maps for `int` are defined starting with the refined concepts `A`, `B`, and `C`. The concept maps for the refining concepts have no explicit definitions of `void f()` and must "pull" the implementations from the refined concept maps. Yet, for both concept maps `E<int>` and `F<int>` there are conflicting definitions in the refined maps. In the maps for `float`, the first concept map is defined and it implicitly generates maps for `A<float>` and `B<float>`. When the map for `F<float>` is defined, an explicit definition of `void f()` must be given so that the missing map for `C<float>` can be generated but that definition conflicts with the one in `B<float>` generated previously. If our "sibling-to-sibling" semantics was allowed, the map for `F<float>` could be defined without an explicit definition of `void f()`, first pulling the definition from `B<float>` and then pushing it to `C<float>`.

Our choice of compatibility semantics is not meant as a final one. Instead, we indicate that there may be a problem and we show one example solution.

Substitution of associated type implementations. According to the wording in clause 14.9.2.2 para. 2, associated types are substituted with their assigned implementations in the signatures of associated functions within concept maps; for example, `void f(A<T>::t)` becomes `void f(int)` in contexts where `A<T>::t` has been implemented as `int`. In our semantics, this substitution does not take place; this can be seen in premise 4 of the first rule in figure 2.9 where signatures of function definitions must be exactly the same as the signatures of associated functions, without substituting the actual type implementations, and in premise 1, where signatures of the associated functions are normalized, i.e., parameter types are fully qualified (the extraction of associated function signatures is not shown in any of the figures but is included in the companion technical report, Paper VI). These function definitions are directly stored in concept maps, which can be seen in premise 2 of the rule in figure 2.7 for explicit concept maps, and premise 9 of the rule in figure 2.11 for implicit concept maps. The following code listing shows an example of a situation where our semantics allows code that is incorrect according to the wording and rejects code that the wording deems correct.

```
concept A<typename T>        { typename t;  void f(t); }
concept B<typename T> : A<T> { typename t;  void f(t); }

concept_map B<int> {
  typedef int t;
  void f(int) { return; }
}
concept_map B<float> {
  typedef int t;
  void f(A<int>::t) { return; }
  void f(B<int>::t) { return; }
}
```

The concept map for `B<int>` is allowed by the wording but not by our semantics. The difference between two associated functions is lost and they are "munged" into a single

`void f(int)` function. The second map for `B<float>` is allowed by our semantics but not by the wording. According to the wording, both `A<int>::t` and `B<int>::t` become aliases to `int` due to the definition on line 9 and the two function signatures on lines 10–11 are identical, causing an error. In our semantics, the substitution of associated type implementations does not take place and, although both associated types have been assigned the same implementation, the functions remain different, insulating templates constrained with the concept `B` from the fact that both associated types are assigned the same implementation.

6.3 Type Checking of Concept-Constrained Templates

According to the wording, clause 14.10.2 para. 18, in a situation where different requirements introduce conflicting associated functions, no effort should be made to check whether the conflicting functions are actually the same. The following listing gives an example of a situation in which name lookup finds the same function through each of the requirements but the call to the function results in an error:

```
1 concept Child<typename T> { void f(); }
2 concept Parent1<typename T> : L<T> { }
3 concept Parent2<typename T> : L<T> { }
4
5 template<typename T> requires Parent1<T>, Parent2<T>
6 void test1() { f(); }  // error
```

As a consequence of performing type lookup in each of the requirements separately and choosing the type found first in depth-first search, sometimes intuitively correct code may cause errors; a small example is shown in the following code listing[2]:

```
1 concept A1<typename T>               { typename t; }
2 concept A2<typename T>               { typename t; }
3 concept P1<typename T> : A1<T>, A2<T> { }
4 concept P2<typename T> : A2<T>, A1<T> { }
5 concept P3<typename T> : A2<T>, A1<T> { }
6
7 template<typename T>  requires P1<T>, P2<T>  struct Test1 {
8   typedef t test;  // error: A1<T>::t or A2<T>::t?
9 };
10 template<typename T>  requires P2<T>, P3<T>  struct Test2 {
11   typedef t test;  // ok: A2<T>::t
12 };
```

In `Test1`, name lookup for `t` finds `A1<T>::t` in `P1<T>::t` and `A2<T>::t` in `P2<T>::t`. Since the two types are different, there is ambiguity. In `Test2`, lookup of `t` in both of the requirements results in `A1<T>::t`, and the sequence of final results contains two types that are the same. The first rule in figure 2.6 makes it clear that lookup is performed in each requirement separately and that the knowledge about identity of associated types is lost between requirements. In this case our semantics agrees with the wording but a potential problem is clearly exposed.

7 Related Work

Linguistic support for concepts in C++ has a long history in the C++ standardization process, dating back to Stroustrup (2003) and Siek et al. (2005b). More recently, concepts have been presented to a broader forum by Dos Reis and Stroustrup (2006) and

[2]The example would require same-type constraints in the concepts `P1`, `P2`, and `P3` but in our semantics the constraints are implied.

soon after by Gregor et al. (2006b). All concept proposals share the goal of separate type checking safety and Dos Reis and Stroustrup indeed provide a sketch of a type checking safety theorem as a future work item. Our theorem 1 is a development of this idea, supported by our semantic definition—in that sense, our contributions are a direct continuation of the previous work.

The tradition of C++ concepts is rooted in the formal setting of algebraic specification (Kapur and Musser, 1992) but, nevertheless, the true utility of concepts was first manifested in the design documentation of the Standard Template Library (Austern, 1998; Stepanov and Lee, 1994 (revised in October 1995 as tech. rep. HPL-95-11). As the prominence of concepts rose in C++ generic libraries, there was renewed interest in formalizing concepts. Willcock et al. (2004) proposed a formal definition of concepts based on their algebraic specification roots, on various counterparts in other languages, mostly in ML (Milner et al., 1990b), and on the ongoing experience of concept use in the C++ world. Their formalization had the ambitious goal of capturing the very notion of concepts in a programming language. Siek and Lumsdaine (2005), on the other hand, incorporated concepts into System F (Girard, 1972), providing strong formal foundation to separate type checking with concepts and outlining the important features that a concept system should include. Finally, in our own work (Zalewski and Schupp, 2007), we bridged the gap between concepts as proposed for C++ and their roots in algebraic specification—we have described how concepts may be interpreted as specifications in the context of institutions (Goguen and Burstall, 1992) (Haveraaen (2007) also considers concepts as institutions, specifically in the context of testing). In the current paper, we complement the existing body of work with a C++ specific formalization that formalizes the design of concepts as proposed for the next version of C++. Our formalization has the practical goals of enabling precise analysis of the current design and providing a foundation for the future compiler implementations, but also the more theoretically inclined goal of initiating the work on formal verification of crucial parts of C++ concepts design.

Concepts are the basis of practical analysis and programming techniques. Currently, these applications of concepts are developed in terms of concrete C++ syntax and tools—our analysis provides a foundation to describe and develop such applications. In our previous work (Zalewski and Schupp, 2006), we have demonstrated a change impact analysis for conceptual specification of generic libraries; Tang and Järvi (2007) propose concept-based optimizations. Concepts have also been used to compose generic libraries. Järvi et al. (2007) use concepts to compose GUI controls from different libraries and to compose graph and imaging libraries; Breuer et al. (2006) compose a parallel graph and eigensolver libraries to obtain an efficient parallel graph eigensolver. Furthermore, Gregor and Lumsdaine (2005) develop principles and patterns for using generic libraries—their development is based on concepts.

Our formalization shares the motivation and some techniques with formalizations of other parts of C++. Wasserrab et al. (2006) have formalized type safety of multiple inheritance and provided a machine-checked proof in support of their hypothesis. Many issues covered by their formalization are related to lookup of names; in particular, they rely on a slightly modified sub-object model introduced by Rossie and Friedman (1995). Siek and Taha (2006) have formalized the template instantiation process and operational semantics for a subset of the C++ language, providing a proof of type safety. Their formulation of the template instantiation process served as the basis for our program instantiation relation. Dos Reis and Stroustrup (2005) have provided a formal framework for a large part of the C++ language, concentrating on the features necessary to concept checking. Their formalization is broad, covering a large part of

the C++ language, but is not as well developed as others and, to our knowledge, has not been used in the current process of specifying concepts. A final note of caution: Järvi et al. (2006) have identified cases where separate type-checking safety cannot hold, yet those violations stem from the interaction of the (safe) concept language with unsafe features *outside* (e.g., partial ordering of function templates) and cannot be prevented without changing, and breaking, legacy C++.

Finally, our formalization has been strongly inspired by the Ott tool (Sewell et al., 2007). Ott lowers the cost of formally defining a language and enables a smooth progression in formality, from LaTeX typeset mathematics to a theorem prover definition. Our semantic definition borrows many techniques from the formalizations of a fragment of OCaml by Owens (2008) and of Java modules semantics by Strniša et al. (2007), which are defined using Ott.

8 Conclusions and Future Work

While concepts are considered one of the most important features to be introduced in the next version of C++, the current design process relies on informal design documentation supported by only a partial implementation and little experience. In this paper we have provided a formal semantic definition of the crucial parts of concept design, covering name lookup in concepts, implementation binding in concept maps, and type checking in concept-constrained templates.

We first give, in section 2, an informal introduction to concepts. We illustrate separate type checking with several examples, such as listing 2.1 that shows the difference between concept-constrained and unconstrained templates, and with more specific examples, such as listing 2.5 that shows implementation binding for associated functions.

Next, in section 3, we introduce the abstract syntax of a small language, X++, that contains the crucial features of the C++ concept system: concepts, concept maps, and concept-constrained templates. X++ includes language features necessary to discuss concepts, such as overload resolution, but excludes all other features that are orthogonal to concepts.

The semantic definition of concepts, which is the crux of our contribution, is presented in section 4. There, we formulate the separate type-checking safety theorem (theorem 1). In the rest of the section we present the semantic judgments that form the basis of the theorem. The presentation of the semantics is divided into three parts: name lookup, type checking of constrained templates, and implementation binding.

The working of semantic rules is demonstrated in section section 5; the examples walk the reader step by step through rule applications explaining how separate type checking is performed. In the same section, we argue for the correctness of separate type checking safety (theorem 1). We first introduce a notion of instantiation and, then, a notion of type checking of instantiated templates. Type checking of instantiated templates serves as a correctness criterion against which the safety of instantiation is checked. We provide informal but rigorous argument for correctness by case analysis of semantic rules.

Finally, in section 6, we discuss how our formal definition corresponds to the informal definition in the wording (Gregor et al., 2008e). Two differences stem from our attempt to fix apparent errors in the wording. First, type lookup rules in the wording do not respect hiding rules established elsewhere in the wording; in our semantic definition the hiding rules are honored. Second, the wording admits a situation where concept maps for refined concepts can provide conflicting implementations for associ-

ated entities; our semantics does not allow such situations. Furthermore, our semantics differs from the wording where intuitively valid code is not admitted by the wording. In summary, the differences are that our semantics allows "sibling-to-sibling" propagation of associated function implementations and it does not perform substitution of associated types with their implementations in concept maps. Furthermore, our semantics exposes a possible problem with type checking of concept-constrained templates where types and functions that are intuitively the same are considered different.

Our formalization is useful in the analysis of the current concept design put in front of the C++ standardization committee. A formal definition of concept semantics, such as ours, is a great aid in understanding and discussing the design, and can serve as the specification for implementations. Also, a formal semantics makes it easier to design program analyses involving concepts; instead of having to understand hundreds of pages of C++ standard and complex compiler implementations, an analysis can be prototyped at the abstract level of semantic entities and relations.

To facilitate such design and prototyping, we plan to make our semantics "executable" through a context-sensitive term-rewriting system. In particular we plan to investigate the PLT Redex rewriting system (Matthews et al., 2004). Using a context-sensitive rewriting system may require restatement of our semantic definition with evaluation contexts, in the style of Wright and Felleisen (1994). Ott currently supports evaluation context semantics but it does not generate output for a rewriting system.

To further enhance usefulness of our semantics for the development of tools we plan to extend the coverage of C++ with concepts. First, we plan to extend our semantics with concept parameters. Directly related features that we plan to include in the next version of semantics are template concept maps and same type constraints on template parameters. We anticipate that adding parameters and the related features may as much as double the size of the definition.

Paper II

Change Impact Analysis for Generic Libraries

Comparing to the published version of this paper, the related work section is extended.

Change Impact Analysis for Generic Libraries

Marcin Zalewski and Sibylle Schupp

Abstract

Since the Standard Template Library (STL), generic libraries in C++ rely on *concepts* to precisely specify the requirements of generic algorithms (function templates) on their parameters (template arguments). Modifying the definition of a concept even slightly, can have a potentially large impact on the (interfaces of the) entire library. In particular the non-local effects of a change, however, make its impact difficult to determine by hand. In this paper we propose a *conceptual* change impact analysis (CCIA), which determines the impact of changes of the conceptual specification of a generic library. The analysis is organized in a pipe-and-filter manner, where the first stage finds any kind of impact, the second stage various specific kinds of impact. Both stages describe reachability algorithms, which operate on a *conceptual* dependence graph. In a case study, we apply CCIA to a new proposal for STL iterator concepts, which is under review by the C++ standardization committee. The analysis shows a number of unexpected incompatibilities and, for certain STL algorithms, a loss of genericity.

1 Introduction

Arguably one of the best known generic libraries is the Standard Template Library of C++ (STL) (Musser et al., 2001; Stepanov and Lee, 1994 (revised in October 1995 as tech. rep. HPL-95-11). From a practical point of view, STL provides a collection of generic containers and generic algorithms to operate on containers. However, it is the introduction of so-called *concepts*, underlying the organization of algorithms and containers, that is the outstanding contribution of STL. What characterizes "STL-style programming", or generic programming, is therefore not just the exploitation of C++ templates but, more importantly, the development of concepts and conceptual specifications for a particular application domain. Although concepts can be formalized (type-) theoretically and in a language-independent way (Kapur and Musser, 1992; Siek and Lumsdaine, 2005; Willcock et al., 2004), most developers are familiar with them as a way to constrain C++ templates: instead of designing an algorithm merely in terms of (universally quantified) template parameters, the designer can use a concept to attach additional requirements on the types that instantiate a template. These requirements can be syntactic, semantic, or behavioral, but they always are abstractions from types. In the following, we denote by *conceptual specification* the concept taxonomy and the concept-constrained interfaces that define the specification of a generic library.

Although concepts on the whole ease the maintenance of a library, their very nature introduces some complications one has to be aware of when a particular concept is modified. For one, each concept describes a generic specification of a family of types, which is met by many concrete types at the implementation level of a library. A modification in a concept definition can therefore affect many types and algorithms with parameters that are constrained by the modified concept. Second, in contrast to abstract classes in object-oriented programming, a concept does not aim at capturing the complete interface of a type. Instead, it defines a coherent set of requirements, which a concrete type must meet at the minimum. As a consequence, modifications in a concept definition might go unnoticed for a long time just because all concrete types that instantiate the affected template, incidentally meet both the old and new concept specification. In fact, our initial interest in change-impact questions comes from a practical problem. When algorithms were changed to use STL allocators, the allocator

concept introduced an additional requirement "DefaultConstructible." Yet, all types we used with the modified algorithms happened to be default-constructible; only later, and then unexpectedly, code broke when used with types that did not provide a default constructor. Finally, algorithms might be affected by a change in a concept even if none of their parameters is directly constrained by that concept. The non-local impact is propagated by the *refinement relation* between concepts, where one concept includes the requirements of one or more other concepts.

In this paper we propose a *conceptual* change impact analysis (CCIA), which determines the impact of changes of the conceptual specification of a generic library. The analysis is organized in a pipe-and-filter manner, where the first stage finds, for any part of a conceptual specification, all changes it is impacted by. Subsequent optional filters on the second stage further refine the output in various ways, to detect specific kinds of impact. Presently, we provide two filter algorithms: one to detect the impact of changes on the compatibility of different versions of concept specifications, and one to detect the impact of changes to the degree of genericity of an algorithm, that is, the number of types with which it can be instantiated. It is possible to extend the analysis by additional filters and stages.

We applied CCIA to one of the most fundamental changes to STL-like libraries that is currently under consideration by the C++ standardization committee: the proposed change of iterator concepts (Siek et al., 2004). As our analysis can show, this proposal unintentionally introduces a number of incompatibilities (between old and new iterator concepts) and renders several STL algorithms less generic than before.

Recent discussions indicate that, "in some form or other" (see Reis and Stroustrup, 2006), C++ will be extended by direct support for concepts (Gregor et al., 2006a; Siek et al., 2005a; Stroustrup and Reis, 2005); indeed, our work is partly motivated by this discussion, as we are interested in providing good tools for concepts once they have become first-class citizens. Currently, however, concepts are described in an informal way, using structured natural language. For CCIA, the current state of affairs implies that the analysis cannot yet be completely automated: the task of transforming an informal conceptual specification into a machine-readable format must be done manually. The change impact analysis itself, though, is automated.

The outline of the paper is as follow: section 2 compares CCIA with other change impact analyses, sections 3 and 4 provide background on concepts in C++ and give a complete analysis example. The analysis itself is presented in section 5 and demonstrated in a case study in section 6. Sections 7 and 8 provide conclusions and an outlook on further work.

2 Related Work

An important classification criterion for change impact analyses are the sources for non-locality of change impact. In object-oriented programming, for example, non-locality of impact is introduced by inheritance, dynamic binding, aggregation, and polymorphism (Kung et al., 1994). In imperative programs, the impact of changes may be propagated through side effects such as assignment to, and use of variables (Tonella, 2003). In the case of generic libraries, there are two major sources of non-locality: the separation between concepts at the specification level and types and type parameters at the implementation level; and the refinement relation between concepts, through which changes propagate.

When the sources of non-locality correspond to programming constructs, the im-

pact of changes is propagated by relations established at the source code level of a program. A large class of analyses therefore builds on top of program dependence graphs or system dependence graphs (Horwitz and Reps, 1992). Analyses that operate not exclusively at source code level, on the other hand, but are concerned with models, ontologies, or specifications (Briand et al., 2003; Heflin and Hendler, 2000), construct their own, tailored representation; in our case, the *concept dependence graph*, which represents the concept specification, the (concept-constrained) parameters of generic algorithms, and the changes in both. Since our analysis applies to generic libraries, where the non-locality of impact is due to the separation of concepts and types, its closest counterparts are the class-level CIAs by Rajlich (1997), although we do not use his snapshot model for change propagation. Inspired by the notion of "atomic changes" (Ryder and Tip, 2001), our representation of change implies that all changes take place simultaneously. Following these ideas is Chianti (Ren et al., 2004, 2005), a tool to select regression tests that have to be repeated after a change is made to a Java program. Chianti detects a set of interdependent atomic changes roughly on the method level, including changes to dynamic lookup. The changes are then compared to call graphs of test programs, and affected tests are determined. For every test program, Chianti can detect a subset of affecting atomic changes using the interdependence relation computed earlier.

Our analysis assumes concepts and constrained templates but the bulk of existing generic C++ libraries are written without explicit support for concepts. In such libraries templates may be considered *fragile*, since a change in unrelated code may cause a template to change. A similar situation occurs in aspect-oriented programming where pointcuts may match unexpected pieces of user code. Stoerzer and Graf (2005) propose pointcut analysis to compute how a change in the program affects joinpoints that a pointcut matches. Unconstrained templates could be considered in a similar manner, where dependent expressions in a template correspond to pointcuts, and matched implementations correspond to joinpoints.

Another axis of classification describes the kinds of impact that the analysis identifies. Most analyses are devised for a particular purpose, for example, to identify which regression tests to rerun or which documentation to update (Bates and Horwitz, 1993; Fyson and Boldyreff, 1998). Fewer analyses (most notably Han's Han's, 1997) can be customized by the client. Our analysis falls in between: the analysis can be used to detect different kinds of impacts but is not yet organized in a freely customizable framework. The analysis itself is an instance of a reachability analysis, that is, it traverses the underlying dependence graph appropriately. Like in most change impact analyses, reachability, thus impact, is purely syntactically defined. One could envision a semantic analysis, which checks, e.g., whether two syntactically different sets of requirements are semantically equivalent. Yet, for the current prototype, we consider the gain too low compared to the effort that a semantic analysis incurs, in particular since most applications of CCIA are expected to be "processed" by a human client.

In applications to software evolution or early phases of the software life cycle, change impact analysis essentially requires the identification or classification of the computed effects. In applications to later phases of the software cycle, the identification of change impact often marks the first step only, since these effects must be communicated to other tools or analyses. Our CCIA falls in the first category as it is typical for CIAs based on specifications or requirements documents (Antoniol et al., 2002; O'Neal, 2001; Turver and Munro, 1994). In the latter category, in particular the number of applications to regression testing stands out (Orso et al., 2003; Rothermel and Harrold, 1997).

Description The `BidirectionalIterator` concept defines types that support traversal over linear sequences of values, including support for multi-pass algorithms, and stepping backwards through a sequence (unlike `ForwardIterators`). A type that is a model of `BidirectionalIterator` may be either mutable or immutable, as defined in the `TrivialIterator` requirements.

Refinement of `ForwardIterator`

AssociatedTypes The same as for `ForwardIterator`

Notation Let `X` be a type that is a model of `BidirectionalIterator`; `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	Expr.	Return Type
Predecrement	`--i`	`X&`
Postdecrement	`i--`	`X`

Expression Semantics

Expr.	Precond.	Semantics	Postcond.
`--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`.
`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 `BidirectionalIterator` are guaranteed to be amortized constant time.

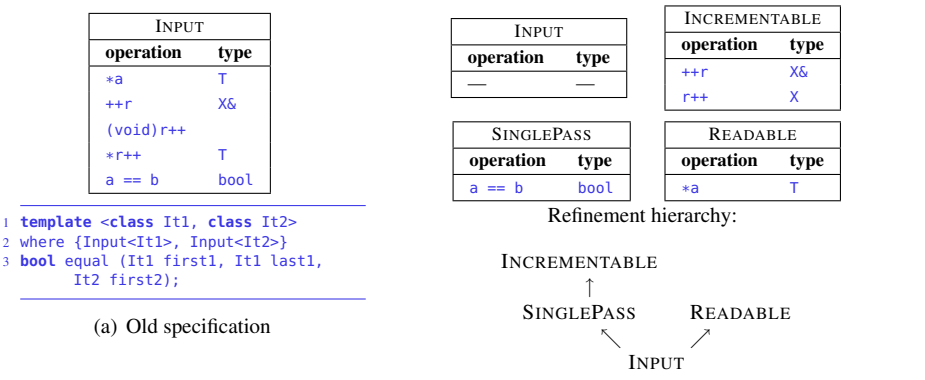
Figure 3.1: Definition of the `BidirectionalIterator` concept of STL (SGISTL)

Finally, in some analyses, only the final impact is of interest, not the single propagation steps. In the delta debugging technique (Zeller, 2002), for example, where program failure is of interest, the changes causing program failure are detected by means of an efficient search, without further consideration of the ways the change has been propagated. CCIA, in contrast, can be used by concept developers who want to understand how a change has caused a specific impact.

3 Concepts in C++

Presently, concepts in C++ are specified in a de-facto standardized documentation. In illustration, figure 3.1 shows the description of the STL `BidirectionalIterator` concept adopted from the documentation of STL (SGISTL). Of particular interest for the paper are the rows "Refinement of" and "Associated Types", and the table "Valid Expressions." The refinement relation specifies that for a type to model (i.e., fulfill all requirements of) the `BidirectionalIterator` concept, it must also model the `ForwardIterator` concept. Associated types are types that must be defined, for example, so that the signatures of the required expressions are well-defined. The table of valid expressions, finally, defines all operations a modeling type has to provide. Two additional requirements, the "Complexity Guarantees" and the "Expressions Semantics" tables, are not included in our impact analysis. The "Expressions Semantics" table, however, is used to precisely encode the informal concept specifications in a machine-usable form (see section 6.2).

As we pointed out already, the informal nature of concept descriptions prevents full automation of the analysis: representing concepts in an effective format currently needs to be done by hand.




```
template <class It1, class It2>
where {Input<It1>, Input<It2>}
bool equal (It1 first1, It1 last1,
      It2 first2);
```

(a) Old specification

```
template <class It1, class It2>
where {SinglePass<It1>, Readable<It1>,
       Incrementable<It2>, Readable<It2>}
bool equal (It1 first1, It1 last1, It2 first2);
```

(b) New specification

Figure 3.2: An example of a change in the conceptual specification of a library

4 Example

Instead of plunging directly into technical details, we introduce our analysis by means of an example. Figure 3.2 shows the original and the modified conceptual specifications of a simple library. The original conceptual specification of the library consists of one concept and one algorithm, `Input` and "`equal`", respectively. The algorithm "`equal`" has two type (or template) parameters, "`It1`" and "`It2`", both constrained by the concept `Input`. In the new version of the specification, the `Input` concept is modified, 3 new concepts along with a refinement hierarchy are introduced, and the constraints on the type parameters of "`equal`" are rewritten. For backward-compatibility, the `Input` concept is part of the new specification but it only refines the newly introduced concepts and does not have any requirements of its own.

The first step of the analysis, performed manually, is to encode the original and the modified versions of the specification and to compare their differences. The comparison involves identifying entities and relations that exist in the old but not in the new version of the specification, and vice versa. As a result of the first step, for example, the concept `Incrementable` is marked as *added* since it exists in the new but not in the old version. All further steps execute automatically.

Next, a directed dependence graph is constructed, which represents the two versions of the specification. The vertices correspond to concepts, requirements, and type parameters, while the edges represent the direction of change impact propagation implied by three kinds of relations: concepts including requirements, concepts refining concepts, and concepts constraining type parameters. A dependence graph for the specification in figure 3.2 is shown in figure 3.3. The graph consists of two type-parameter vertices, four concept vertices of which three are marked as added, and five requirement vertices of which two are marked deleted, one is marked added, and the remaining three are unchanged between the two versions of the library. In the example, all the relations between concepts, requirements, and type parameters have been changed.

The impact analysis proceeds in stages, composed in a pipe-and-filter manner. In

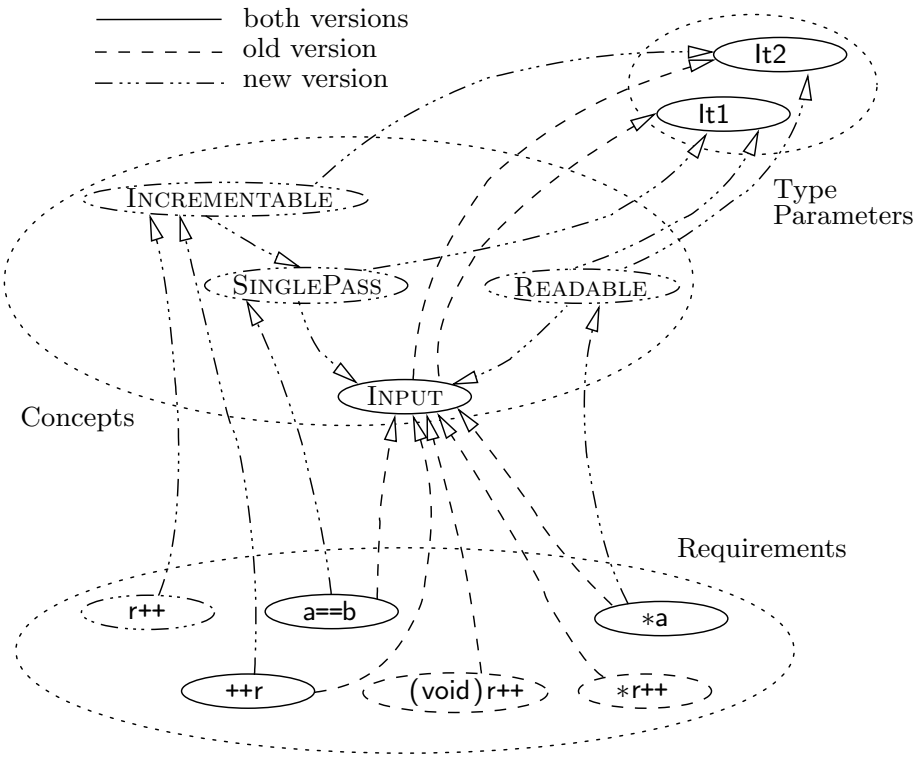


Figure 3.3: The dependence graph constructed for the specification from figure 3.2

the first stage, the impact of the changes is propagated along the edges of the graph to identify the type parameters and concepts that may have been affected. The edges which propagate the impact are marked accordingly for use in subsequent analysis steps.

If one is interested merely in a list of concepts and type parameters impacted by changes, one can terminate the analysis at this point. Otherwise, one of the two currently available filters can be used to determine, for example, whether the concept `Input` represents the same set of requirements as before or whether rewriting the constraints on the parameters of the algorithm "`equal`" increases its genericity. Provided these filters are plugged-in, the analysis yields that the new concept `Input` is neither forward- nor backward-compatible with the old concept of the same name: two requirements are deleted ("`*r++`" and "`(void)r++`"), but also a requirement ("`r++`") is added. Furthermore, the analysis detects that for both type parameters of the algorithm "`equal`" the requirements "`(void)r++`" and "`*r++`" were removed and "`r++`" was added; for the parameter "`It2`" additionally the requirement "`a == b`" is removed. Since requirements on the type parameters are both added and dropped, the analysis can conclude that the algorithm "`equal`" became neither strictly more nor strictly less generic through the change. Both filters reuse the results of the first stage and only operate on relevant (impacted) parts of the dependence graph.

5 Conceptual Change Impact Analysis

As the example in the previous section shows, the three major steps of CCIA comprise the construction of the dependence graph, the first stage of general impact propagation, and subsequent filtering for specific impact. The first stage is a forward-reachability problem that determines *any* impact of *any* changes. The second stage, a backward-reachability problem, varies between different applications. The current prototype provides two filters, which capture two frequent questions about conceptual changes: whether concepts are compatible between different versions of conceptual specification and whether the requirements on a type parameter of a generic algorithm changed. This section details each of the three steps.

5.1 Conceptual Dependence Graph

An intermediate representation of six constructs suffices to capture the conceptual specification of a library: 3 *entities* and 3 *relations*, directly corresponding to the relations that concepts establish (see section 3):

- *Type Parameters*: Type parameters of generic algorithms
- *Concepts*: Sets of requirements
- *Requirements*: Operations, associated types, and any other valid expressions (see figure 3.1)
- *Constrains-relations*: Relations between type parameters and concepts
- *Refines-relations*: Relations between concepts
- *Requires-relations*: Relations between concepts and requirements

For example, the concept `SinglePass` (see figure 3.3) constitutes one requires-relations to the requirement "`a == b`" and one refines-relation to the concept `Incrementable`. The type parameter "`It1`" (see figure 3.3) constitutes one constrains-relation to the `Input` concept in the original library and two constrains-relations to the `Readable` and `SinglePass` concepts in the modified library.

The six constructs of the intermediate representation naturally map to vertices in a graph, where edges capture the dependencies between them. In this graph, any entity or relation that exists in the new version but not in the old one is marked as *added*, and any entity or relation that exists in the old version but not in the new one is marked as *deleted*; any entity or relation that exists in the old version and is modified in the new version is represented by a deletion followed by an addition. In the current prototype, we perform these annotations manually; if concepts were first-class citizens, the annotation could be easily automated by applying a diff-like algorithm to a compiler-provided representation.

We note that the graph in figure 3.3 is a simplified view of a dependence graph insofar it represents only single-parameter concepts and requirements. Since concepts and requirements can have multiple parameters, every relation and entity, except for type-parameter entities, explicitly represents its parameters. Figure 3.4 shows an example of a constrains-relation edge between a two-parameter concept `C` and two type parameters "`P1`" and "`P2`". The relations and entities in figure 3.3 are represented similarly but always happen to have only one parameter vertex.

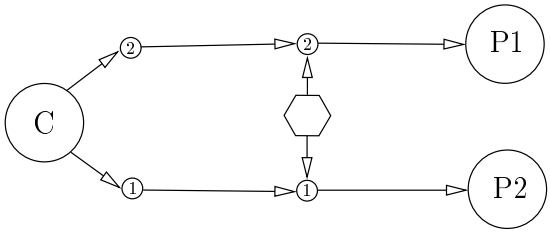


Figure 3.4: A detailed view of a constrains-relation edge.

5.2 Impact Propagation

After the graph is constructed, the impact of changes is propagated: any vertex marked as added or deleted has an impact on any of the vertices reachable from it. It is sufficient, however, to only consider relation vertices: a deletion of an entity must be accompanied by deletion of the corresponding relations and addition of an entity does not have a non-local impact unless some relations are added. Depending on the particular kind of impact sought after, addition or deletion of a relation may not be significant. Yet, in general, all vertices reachable from a relation marked as added or deleted are impacted in *some* way.

The algorithm for propagating the impact of changes is rather straightforward. In short, it is a depth-first search where the root vertex of the search is a relation vertex marked as either added or deleted and all the traversed edges are marked as change-propagating. The search stops on deleted edges if the root vertex of the search is marked added, and on added edges if the root vertex is marked deleted. This condition is necessary to avoid false impact propagation; for example, if a path from an added vertex v to a vertex u contains a deleted edge e, u should not be affected by the added vertex. An algorithm that finds all impacting edges for a given vertex y is similar, but the direction of the search is reversed. In the example from section 4 every relation is marked as added or deleted and thus all edges reachable from relation vertices are marked as propagating. In practice, however, such all-encompassing change of a library will rarely occur and only some edges will be marked as change-propagating.

5.3 Filtering for Specific Impact

The impact propagation algorithm identifies *any* impact. Yet, a more specific kind of impact is often of interest. In this subsection, we present two algorithms, *Constraints Change* and *Concept Compatibility*, to detect specific impact. The first algorithm checks whether requirements on algorithm parameters are changed, the second one determines the compatibility of concepts between different versions of a conceptual specification. Both algorithms operate on the level of syntactic requirements. As said earlier, these filter algorithms are based on the result of the preceding impact propagation algorithm and further refine its results. Separating the general impact algorithm from the filters for specific impact, makes the analysis both extensible and scalable: a potentially unlimited number of specific kinds of impact and accordingly specializing filters can be admitted. At the same time, each of these filtering algorithms can restrict itself to impacted vertices. Where only general impact is of interest, filters may be omitted altogether. We now turn to the discussion of the two algorithms.

Informally speaking, algorithm *Constraints Change* computes the change in syntactic requirements on type parameters. The algorithm consists of 2 main steps. First, for all type parameters that are reached by some change (that is, targets of change-propagating edges), the set of reaching changes has to be found. This is done by a search through the reversed dependence graph, traversing only change-propagating edges. Second, for every change that reaches a type parameter, the specific impact on the type parameter has to be computed. For example, if the change reaching some type parameter is a deleted constrains relation, all syntactic requirements implied by the constraining concept have to be marked as removed for that type parameter. A high-level definition of the algorithm follows:

Algorithm *Constraints Change.*

Input: $\mathscr{G}$, a dependence graph where all change-propagating edges are marked; T, a type parameter vertex.

Output: R, a set of tuples (p,S) such that p is a path in $\mathscr{G}$ from T to a modified vertex q and S is a set of paths from q to the added or deleted requirements on T that result from the change in q.

Local: *reaching_paths*, a container of the results of [Find changes].

Notation and subroutines:

path: A path in a reversed dependence graph.

forward or cross edge: An edge (u,v) where v is colored black and not an ancestor of u in a search tree.

last(): Given a path, extracts the last vertex.

significant_vertex(): Given a parameter vertex, finds the main vertex of the corresponding relation or entity.

A1. [Filter and revert $\mathscr{G}$.] $\mathscr{G}'$ is $\mathscr{G}$ with all edges reversed and non-propagating edges removed.

A2. [Find changes.] Run depth-first search on $\mathscr{G}'$ with T as the root vertex:

A2.1. If a vertex marked as added or deleted is discovered, record current path in *reaching_paths*, mark vertex black, and backtrack depth-first search.

A2.2. If a forward or a cross edge is detected in the depth-first search, recursively run [Find changes] for the target of that edge. Merge all detected paths with the current path and record it in *reaching_paths*.

A3. [Process Changes.] For each path p in *reaching_paths* where s is *significant_vertex*(*last*(p)):

A3.1. If s is a constrains-relation, flatten the requirements of the constraining concept. Record the tuple (p, {*paths from the flattened concept to requirements*}) in R.

A3.2. If s is a refines-relation, flatten the requirements of the refined concept. Record the tuple (p, {*paths from the flattened concept to requirements*}) in R.

A3.3. If s is a requires-relation, record the tuple (p, {*one-vertex path of last*(p)}) in R.

■

In the example from section 4, every reaching change identified in step A2 is an added or deleted constrains-relation. Consequently, in step A3, only A3.1 applies.

The second algorithm, *Concept Compatibility*, is an extension of the first algorithm, *Constraints Change*. For every concept for which compatibility is tested, temporary

type-parameter vertices are created. Specifically, a type-parameter vertex and a corresponding constrains-relation edge are created for every parameter of the tested concept that is a target of a change-propagating edge. The *Constraints Change* algorithm is then performed on each of the temporary type-parameter vertices. To decide whether compatibility holds, finally, the requirements that hold for a type parameter in the new and the old versions of the library need to be compared. In the current prototype, we simply check whether every requirement removed for a type parameter in one way was then added in another way (backward-compatibility) and whether every added requirement existed previously (forward-compatibility). Such simple comparison could find false positives, for example, if a deleted requirement continues to be associated with a type parameter through another, unchanged path in the graph. False positives can be eliminated in a third, forward pass that checks for any added or deleted requirement whether other paths exist that neutralize the effect of addition or deletion, respectively. We have not yet implemented this pass, but the algorithm *Constraints Change* is prepared insofar it already associates changes and paths.

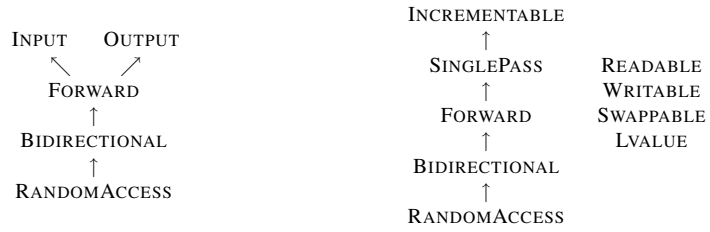
6 Case Study

For a case study, we apply CCIA to a proposed change of the iterator concept taxonomy of STL. The background of this proposal is a discussion that started in 2001, when Siek pointed out that the currently standardized iterator concepts entangle the concerns of range traversal and data access (Siek, 2001). Because of such unnecessarily strong coupling between value-access and traversal requirements, some generic algorithms are under-generalized and some iterator types incorrectly categorized with respect to their traversal protocol (see, e.g., Sutter's paper, 1999). Since 2001, altogether 5 new concept specifications for iterators have been submitted for consideration by the C++ standardization committee, and the discussion still continues.

Since iterators are in the very core of STL, any new specification will impact virtually every user of STL and many other libraries. Since these changes are subtle and their impact, because of non-local effects, non-trivial to detect, the proposed changes make for a good case study of a change impact analysis. In this study, we concentrate on the two kinds of intended impact that the proposal sets out to make, namely to reduce conceptual requirements on the parameters of STL algorithms, while at the same time ensuring backward- and forward-compatibility between old and new iterator concept taxonomies. In the following, we denote by *new* the latest version of the proposed concepts (Siek et al., 2004) and by *old* their current specification from the C++ standard (ISO, 2003a).

6.1 Iterator Concept Taxonomies

In the old iterator taxonomy, each concept includes requirements related to range traversal as well as value access. In the new taxonomy, in contrast, these requirements are divided into two groups: traversal concepts on the one hand and value-access concepts on the other hand. New and old iterator taxonomies are depicted in figure 3.5 (the `Iterator` suffix has been omitted in all concept names); concept refinement is represented by the usual arrows. As the figure shows, the old taxonomy consists of five concepts total, while the new taxonomy has nine concepts: five traversal concepts, corresponding to the traversal requirements of the old taxonomy, and, separately, four value-access concepts that have been factored out from the old concepts.



(a) Old, standardized taxonomy

(b) New, proposed taxonomy

Figure 3.5: Old and new iterator taxonomies

INPUTITERATOR	
operation	**type**
`X u(a);`	`X`
`u = a;`	`X&`
`a == b`	convertible to `bool`
`a != b`	convertible to `bool`
`*a`	convertible to `T`
`a->m`	
`++r`	`X&`
`(void)r++`	
`*r++`	convertible to `T`

SINGLEPASSITERATOR	
operation	**type**
`++r`	`X&`
`r++`	`X`
`a == b`	convertible to `bool`
`a != b`	convertible to `bool`

READABLEITERATOR	
operation	**type**
`X u(a);`	`X`
`u = a;`	`X&`
`*a`	convertible to `T`
`a->m`	

T denotes the value type, *X* the type of the modeling iterator.

Figure 3.6: "Valid Expression" tables of the `InputIterator` concept (ISO, 2003a, Table 73) and its (incompatible) refactoring into the concepts `SinglePassIterator` and `ReadableIterator` of the new taxonomy (Siek et al., 2004)

The new concepts were specified with the intention of avoiding compatibility problems. In illustration, figure 3.6 shows the old `InputIterator` concept, which is refactored into two concepts, `SinglePassIterator` and `ReadableIterator` in the new taxonomy. As CCIA will reveal, however, the new and old `InputIterator` concepts are, in fact, not fully compatible. The proposal also aims at increasing the genericity of a number of STL algorithms and, to that end, provides rules for rewriting the current constraints on type parameters by the appropriate constraints through the new concepts. For instance, one of the rules specifies that for the STL algorithms `equal` and `mismatch`, all parameters constrained by (the second occurrence of) the concept `Input` are now to be constrained by the concepts `Incrementable` and `Readable`. CCIA will confirm that (under additional conditions, see section 6.4) the application of this rewrite rule will dispose of 4 requirements, thus makes both algorithms more generic.

6.2 Experiment Setup

Before the analysis can be conducted, the conceptual specifications have to be encoded in a form from which the dependence graph (see section 5.1) can be constructed. Unfortunately, this encoding cannot be automated. As explained, the iterator concepts are specified informally, using natural language, with parts of the requirements formatted as tables (see, e.g., figure 3.1). Further syntactic and semantic requirements are dispersed throughout the documentation—manual encoding is thus unavoidable.

The conditional specifications that are often used in valid expression tables repre-

Old concept	Corresponding new concepts
INPUT	READABLE, SINGLEPASS
OUTPUT	WRITABLE, INCREMENTABLE
FORWARD	READABLE, READABLELVALUE, FORWARD
MUTABLEFORWARD	READABLE, READABLELVALUE, FORWARD, BASICWRITABLE, WRITABLELVALUE
BIDIRECTIONAL	READABLE, READABLELVALUE, BIDIRECTIONAL
MUTABLEBIDIRECTIONAL	READABLE, READABLELVALUE, BIDIRECTIONAL, BASICWRITABLE, WRITABLELVALUE
RANDOMACCESS	READABLERANDOMACCESS, READABLE, READABLELVALUE, RANDOMACCESS
MUTABLERANDOMACCESS	READABLERANDOMACCESS, READABLE, READABLELVALUE, RANDOMACCESS, WRITABLERANDOMACCESS, WRITABLE, WRITABLELVALUE

Table 3.2: Correspondences between the old and the new iterator concepts

sent an additional complication; they have to be encoded as two different requirements in two different concepts, each corresponding to one branch of the condition (Gregor et al., 2005, App. A). For example, the return type of the dereference operator "`*a`" in `ForwardIterator` is stated as "`T` if `X` is mutable, otherwise `const T&`" (where `T` denotes the value type, `X` the type of the modeling iterator) (ISO, 2003a, Table 75). To represent this requirement we had to introduce the `MutableForwardIterator` concept corresponding to the "is mutable" condition. Other conditional specifications in disguise have the form of optional (type) qualification or different return types of overloaded expressions. We forgo the further discussion of the details of the encoding process as they are not crucial to the case study. Table 3.2 shows the intended compatibility between old and new concepts including the concepts added during the encoding process. Using this table, compatibility can be decided row-wise: a concept in the old taxonomy is forward-compatible if any type modeling the concept also models the new concepts in the same row. Conversely, a concept of the new taxonomy is backward-compatible if every modeling type also models the old concept in the same row.

The setup of the case study is now easy to explain. To check the compatibility between the old and the new concepts, we proceed essentially as already illustrated in section 5.3: based on the expected compatibilities defined in table 3.2, we redefine all old concepts in terms of their counterparts in the new proposal, that is, we mark all requires- and refines-relations from the old specification as *deleted* and then *add* refines-relations from every old concept to the corresponding new one(s). From there, we build the conceptual dependence graph and then run algorithm *Concept Compatibility*. To calculate the changes in the requirements of STL algorithms, we encode the rewrite rules from the iterator proposal (see section 6.1 for an example) and apply *Constraints Change* to the type parameters of the STL algorithms. The following two subsections discuss the results of the two CCIA algorithms. An example of the command-line output of CCIA can be found in figure 3.7.

6.3 Compatibility

Surprisingly, the analysis yields the new and the old iterators not compatible. More specifically, none of the 8 old concepts and their corresponding new concepts (as in table 3.2) is backward- or forward-compatible. Even if incompatibilities propagated through the refinement hierarchy are ignored, there are only 3 concepts that introduce no incompatibilities on their own: `ForwardIterator` and `BidirectionalIterator` (yet, see the discussion below), and the `MutableBidirectionalIterator` concept that we had to introduce (section 6.2). These 3 concepts will be automatically both back-

Full:

```
InputIteratorModel --> constrains --> Iter (of InputIterator) --> refines -->  (DELETED)
     T (of CopyConstructible) --> requires --> T (of T t; T(t);)
     T (of CopyConstructible) --> requires --> T (of const T u; T(u);)
InputIteratorModel --> constrains --> Iter (of InputIterator) --> refines -->  (ADDED)
     Iter (of ReadableIterator) --> requires --> T (of typename T::value_type;)
     Iter (of ReadableIterator) --> requires --> T (of T::value_type v; T p; v = *p;)
```

Compatibility-summary:

```
InputIteratorModel NOT COMPATIBLE
ForwardIteratorModel NOT COMPATIBLE
```

Compatibility-incompatible:

```
InputIteratorModel
     Requirement "T t; T q = t++;" added 1 times, deleted 0 times. (FORWARD INCOMPATIBLE)
     Requirement "typename T::difference_type;" added 0 times, deleted 1 times. (BACKWARD INCOMPATIBLE)
```

Compatibility-short:

```
OutputIteratorModelIter
     Requirement "T t; T u; T& q = (t = u);" added 1 times, deleted 1 times.
     Requirement "T t; const T v; T& q = (t = v);" added 1 times, deleted 1 times.
```

Genericity-change:

```
find_first_of::ForwardIterator2  --- Genericity not increased.
```

Figure 3.7: Examples of five different kinds of output from the analysis

	Con.	Requirement	
1	**O**	`typename Iter::value_type;`	b
2	**O**	`Iter r; Iter q = r++;`	f
3	**O**	`typename Iter::difference_type;`	b
4	**O**	`typename Iter::pointer;`	b
5	**O**	`typename Iter::reference;`	b
6	**O**	`Iter r; const Iter& q = r++;`	b
7	**O**	`Iter r; V o; *r++ = o;`	~~b~~
8	**I**	`Iter r; Iter q = r++;`	f
9	**I**	`typename Iter::difference_type;`	b
10	**I**	`typename Iter::pointer;`	b
11	**I**	`typename Iter::reference;`	b
12	**I**	`Iter r; r++;`	~~b~~
13	**I**	`Iter r; Iter::value_type q = *r++;`	~~b~~
14	**F**	`Iter r; const Iter::value_type& q = *r++;`	~~b~~
15	**MF**	`Iter r; Iter::value_type& o = *r;`	f
16	**MF**	`Iter r; Iter::value_type o; *r++ = o;`	~~b~~
17	**B**	`Iter r; Iter::value_type q = *r--;`	~~b~~
18	**RA**	`Iter r; Iter::value_type q = r[n];`	f
19	**RA**	`Iter r; const Iter::value_type& q = r[n];`	b
20	**MRA**	`Iter r; Iter::value_type v; r[n] = v;`	f

where **O**=OUTPUT, **I**=INPUT, **F**=FORWARD, **MF**=MUTABLEFORWARD, **B**=BIDIRECTIONAL, **RA**=RANDOMACCESS, **MRA**=MUTABLERANDOMACCESS

Table 3.3: The requirements that cause forward-incompatibility (f) or backward-incompatibility (b). False positives are indicated by ~~stricken~~ text.

ward- and forward-compatible provided their refined concepts have been made compatible.

Table 3.3 details the incompatibilities. Following the refinement hierarchy, the table lists for each concept exactly the incompatibilities this concept introduces, the incompatibilities that are only propagated through refinement are omitted. Each row, thus, corresponds to one incompatibility; the kind of incompatibility is indicated in the last column. For instance, line 1 of the table indicates that the associated type "`value_type`" of the old `OutputIterator` concept is missing in the corresponding new concepts (`WritableIterator` and `IncrementableIterator`), which breaks backward-compatibility. A further 6 incompatibilities of the `OutputIterator` concept are given in lines 2-7. They all propagate to all refining concepts—that is, all other concepts except the `InputIterator` concept—but are not listed again in the table.

algorithms	Del.
reverse_copy, find_end, adjacent_find, search, search_n, rotate_copy, lower_bound, upper_bound, equal_range, binary_search, min_element, max_element	1
find_first_of	3, 4
copy_backwards	1, 0
equal, mismatch, transform	4

Table 3.4: STL algorithms with increased genericity, grouped by the number of requirements removed per parameter (second column); backward-compatibility is provided.

It is important to note that some incompatibilities detected by our analysis are in fact wrong, albeit in a subtle way that shows a general limitation of our approach. We indicate all faults on part of the analysis by stricken text in the last column of Table 3.3. In all cases, the reason of the false positive is that the analysis does not take the semantics of the requirements into account. As a purely syntactical analysis, it cannot recognize mere splitting or merging of C++ expressions. For example, the requirement "`*r++`" of the `ForwardIterator` concept is decomposed into two requirements, "`r++`" and "`*r`", which are then associated to different new concepts (false positive on line 14). False positives were identified manually after the analysis has completed.

6.4 Constraints Change of Parameters

Every backward-incompatibility from table 3.3 introduces an additional requirement on an algorithm parameter, which it originally did not have to meet. Since it is of interest to understand nevertheless how much the genericity *could* increase, we enforced total compatibility for the purpose of the experiment. For every backward- and forward-incompatibility in table 3.4 we accordingly added and removed the corresponding requirement in the old concepts.

Table 3.4 shows for which algorithms their genericity increases provided the intended compatibility between the new and the old iterators holds. The algorithms are grouped by the number of requirements removed for each of their type parameters. From the 42 STL algorithms that are affected by the changes to the iterator concepts, 17 became more generic.

7 Conclusions

Generic libraries in C++ use so-called *concepts* to constrain type parameters in templates; conversely, concepts control the degree of genericity of an algorithm or a class. The task of a generic library designer, therefore, includes the development of a concept taxonomy for the particular application domain of the library. Despite their importance for specification and implementation, however, very little mechanical support exists for concepts. In this paper, we introduce the first conceptual change impact analysis, CCIA. The analysis is a reachability analysis, comprising a forward-reachability stage and a backward-reachability stage. For extensibility and scalability, the two stages are organized in a pipe-and-filter manner.

We applied CCIA to an officially submitted proposal to change iterator concepts, one of the most fundamental constructs in STL and other generic libraries. The analysis shows that, unexpectedly, the two iterator concept taxonomies are neither forward- nor backward-compatible and lists the parts of the specification that cause incompatibility. Its results can help library designers to avoid unintended effects of a change and, in

general, provides a base for assessing its impact.

8 Future Work

As the two next steps, we want to fully automate our analysis and make it more customizable. To develop CCIA into a full-fledged tool, we plan to integrate it in the ConceptGCC compiler (Gregor, 2008a; Gregor and Siek, 2005), which is an experimental implementation of concepts for C++ (Gregor et al., 2005). Full integration in a real compiler requires extending our analysis with support for all linguistic features of concepts, like nested requirements, for example.

To make our analysis more customizable, we want to identify basic "queries", similar to the queries in PathInspector™(GrammaTech, 2008). These queries should allow constructing new filters by simple combining basic queries appropriately. We expect already the existing filters, *Constraints Change* and *Concept Compatibility*, to be decomposable into smaller, reusable computation units.

Acknowledgments

We thank the reviewers of ICSM that greatly improved the presentation of the paper. Further thanks go to the participants of the Library-Centric Software Design (LCSD) workshop at OOPSLA, where an early version of the case study was presented; Frank Tip deserves a special mention. We also thank the authors of the Boost Graph Library (BGL), which saved us substantial implementation work. Douglas Gregor provided input for the necessary mapping from the semi-formal specifications in the C++ standard to a machine-usable format.

Paper III

C++ Concepts as Institutions
A Specification View on Concepts

C++ Concepts as Institutions
A Specification View on Concepts

Marcin Zalewski and Sibylle Schupp

Abstract

With the recent developments in the C++ language, concepts are mostly discussed as a form of constrained polymorphism. Yet, concepts also allow for an alternative, implementation-independent view that comes from their origin in (algebraic) specification languages. In this paper, we return to this specification view on concepts and formalize C++ concepts as *institutions*, a well-established notion for precise specifications of software components. We argue that institutions form a suitable theoretical framework for software systems like libraries where the different parts establish relations that are captured by different logics, or no formal logic at all. Assuming the C++ concept descriptions, concept maps, and axioms as in the draft currently accepted by the C++ standardization committee, we show that concept descriptions and axioms form an institution (with equational logic) but also, and perhaps surprisingly, that concept descriptions and concept maps form an institution (with no formal logic).

1 Introduction

In generic programming (Musser and Stepanov, 1998), concept taxonomies are a way of describing abstractions of a certain domain. Concept taxonomies have been used in the context of generic programming as early as 1981 (Kapur et al., 1981a,b). Yet, concepts entered the mainstream of software library development with the introduction of the Standard Template Library (STL) (Stepanov and Lee, 1994 (revised in October 1995 as tech. rep. HPL-95-11). STL is based on a concept taxonomy of basic data structures (most notably, the iterator concepts) and provides fundamental algorithms to operate on these data structures. Since STL introduced concept taxonomies, concepts have pervaded generic library development; the Boost Graph Library (BGL) (Siek et al., 2002) is one of the prominent examples of a modern library based on a taxonomy of concepts.

Concepts have a dual nature in the development of generic libraries. On the one hand, they are used to describe the constraints on parameters of polymorphic algorithms and data structures. On the other hand, they capture the structure of domains and describe mathematical objects. In fact, the very duality of concepts serves as the bridge between an abstract domain and the more mundane world of particular implementations.

C++ concepts, a recently proposed feature for the C++ language (Gregor et al., 2006b; Dos Reis and Stroustrup, 2006), represent the implementation-oriented aspect of concepts. There, concepts are directly a feature of the type system, used to constrain polymorphism; concepts are similar to type classes in Haskell (Wadler and Blott, 1989), signatures in ML (Milner et al., 1990b), or concepts in F^G(Siek and Lumsdaine, 2005). As such constraints, concepts make it safer to use generic libraries. From the point of view of an average user, their possibly most visible effect is the elimination of the infamously complicated error messages during C++ template instantiation. Yet, from the point of view of a library developer, a concept taxonomy also has the important task of describing the domain on which the library operates. The well-known iterator concepts from STL, for example, capture an important *idea* and are useful not only for the purpose of programming but also for the purpose of comprehending of the domain they describe.

Concepts are, thus, a way of capturing ideas and then connecting these abstract ideas to concrete programs, which makes them, in effect, a *specification* of software. Once developers express a concept, such as a particular iterator concept, they also specify, at least implicitly and informally, a class of concrete implementations. In that view, C++ concepts serve as a practical implementation of concepts as specifications for C++ programs.

In this paper, we formalize the intuition of concepts as specification, using the notion of *institutions* (Goguen and Burstall, 1992). Institutions have been introduced in the context of software specification as a way of dealing with multiple ways of describing how one specifies "things," or *models* in other words, and what these models are. An institution abstracts from the details of an implementation and allows one to capture the idea of a logic, by providing a way of separately describing models, semantics, and the relation between models and semantics.

We first provide a general institution of concepts, denoted by $\mathbf{C}$, which abstracts from any particular logic. This general institution formalizes the idea of concepts as specifications of classes of models and allows one to plug-in different notions of satisfaction, including C++ *concept maps* (see section 3.1), as well as different logics. For example, C++ concepts include *axioms*, which are more or less equivalent to a (conditional) equational logic. Correspondingly, we can define a concept institution $\mathbf{C}_{\mathbf{EQ}}$ for equational logic. While in many situations equational logic may suffice, one often needs a different logic, for example higher-order logic or modal logic. The power of institutions lies in the ease with which one can change logics.

Our definition of concepts as institution is a formal development that spells out and clarifies many intuitive notions. As any formalism, it might open doors to new theoretical considerations but can also rather directly guide the design of various practical kinds of software analysis. In fact, our motivation for developing the presented formalism comes from our work on change impact analysis (Zalewski and Schupp, 2006) for generic libraries and the need for a formal framework to precisely define both the analysis and the parts of library to be analyzed, and to design the data structures and algorithms needed in that analysis.

The paper is organized as follows. In section 3 we provide the necessary background on the formalism of institutions, and the syntax and semantics of C++ concepts. In section 4.1, we define concept signatures and concept models, two constituents of the notion of a (concept) institution. The institutions themselves are presented in section 4.2, first the general concept institution, $\mathbf{C}$, without any particular logic, then the concept institution $\mathbf{C}_{\mathbf{EQ}}$ for equational logic, which also serves the purpose to illustrate how particular logics can be treated. A discussion of related and future work concludes the paper.

2 Motivation

Generally, any formalization of concepts can claim to improve the overall understanding of concepts and to provide new vocabulary for their precise formulation. Concepts as institutions, however, give two benefits that other formalizations do not provide: for one, they define a framework through which the different formal logics that will operate on concepts in the form of various tools, can, informally speaking, share the C++-related part of concepts. Second, they provide the means to capture the difference between two versions of a library. Both benefits stem directly from the notion of an institution, which has been introduced as a formalization for encapsulating the details

of a logic and its parts in a minimal set of axioms: the encapsulation allows for easy use of different logics, and the choice of the institution axioms allows for *structured specifications* (Tarlecki, 2003) that can constructively describe how a software module has been built from other modules or modified over time.

Concepts as institutions therefore aim at two kinds of software analysis: verifying and validating analyses on the one hand, which may specify the logic best for their task without worrying about C++ details, and analyses for change impact, evolution, or refactoring on the other hand, which need to express, track, or roll-back the "diff" of two libraries. Obviously, the two kinds of analyses are quite different: verification and validation check for certain properties of the interface, implementation, or usage of parts of a concept-based library, while refactoring and evolution look at entire concept hierarchies. As it so happens, however, concepts as institutions serve both needs. In the remainder of the section, we further elaborate, how.

Consider as a motivation the interface of the STL algorithm `copy`:

```
1 template<typename OutputIterator,
2           typename InputIterator>
3 OutputIterator copy(InputIterator first,
4                      InputIterator last,
5                      OutputIterator result);
```

One of the first notions every STL user learns is that of a *valid range* of a pair of iterators: for example in the `copy` algorithm, `first` and `last` must form a valid range or else the algorithm no longer works as expected. Whether formal or not, thus, logic properties play an important role in STL-like libraries already today: library designers exploit the properties of ordered sets, reachable iterators, or persistent containers to ensure correctness or to permit certain optimizations, and library users have to obey these properties whenever they manifest themselves in the interfaces and APIs as pre-conditions of a method or an algorithm.

C++ with concepts does not force but enables the concept designer to formalize those properties through a feature called *axiom*. Thus, tools for verifying or testing generic libraries (Haveraaen, 2007; Wang and Musser, 1997) or statically checking their proper use (Gregor, 2004), no longer need to rely on self-made specifications but can instead pick up the properties of a concept as the concept author has stated them. Yet the question is, on which logic to build those tools.

Alas, no one logic fits all purposes. Most axioms in the concepts specified to date (Gottschling, 2006; Gregor and Lumsdaine, 2008c) suggest a variant of an equational logic since an optimizing compiler can then simplify concept-based expressions using rewrite rules. Proving the correctness of a valid range, on the other hand, requires a non-standard logic (Musser and Wang, 1995). Neither of the two logics, next, is suitable to establish safety properties, including fairness or the guaranteed return of resources; those typically require temporal logic. And none of these three logics can replace, for example, separation logic, which allows reasoning about heap-based properties of a concept. Altogether, different needs have led to the well-known phenomenon of "population explosion" of logics (Tarlecki, 2003).

At the same time, only few logics are directly ready for use with C++ concepts: most of them have been developed for a formal language and need to first integrate with C++ before they become applicable. The first benefit of institutions is that such integration becomes simple: in concepts as institutions one specifies once, and in an abstract way, *concept parameters*, *concept refinement*, *inline requirements*, and all other C++-specific ingredients of concepts (see section 3.1), and entirely encapsulates the

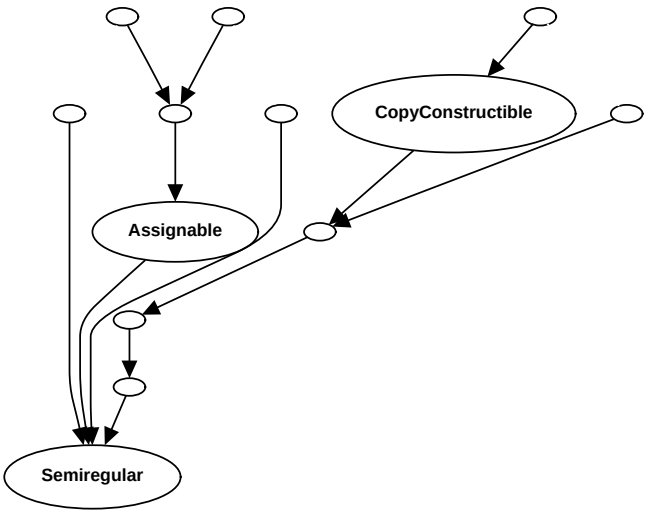


Figure 4.1: CASL development graph example for selected concepts from Concept-GCC (Gregor, 2008a). Edges represent the structuring constructs that compose concept parts into named concepts. The empty nodes signify the anonymous concept fragments from which the concepts are constructed.

logic in the *satisfaction condition*, models, and sentences (see definition 7). Defining a new logic for concept-based software thus comes down to defining just the three parts of a logic; the (category-theoretical) machinery of institutions then propagates the logic automatically to the C++ portion of concepts. Although we do not provide in this paper the actual abstract specification of the C++ constructs of concepts, we provide the first step by showing how to embed C++ concepts in the sphere of institutions. The specifications themselves are part of on-going work.

The second benefit of an institution-based formalization is that it allows for *structured specifications*. In structured specification, the constituents of a C++ concept— parameters, refinements, and requirements we mentioned already— are defined abstractly and in way so that a particular concept can be described through the mappings that compose its parts. Because of the constructive semantics, structured specification lends itself to program analyses that explore in some way the concept hierarchy of generic libraries. Such analyses may in principle be based on the semantics of concepts as defined for the C++ language standard. Yet, the description of C++ concepts there is compiler-oriented and, as such, does not "concentrate" the semantics of structuring concepts in a single, easily extracted form. To give an example, we refer to the notion of parameterization: in C++, it is not trivial to express the idea of parameterization. Using institutions, however, parameterization can be elegantly described in terms of the axioms of institution only without having to consider language details. For that reason, structured specifications are also of interest to analyses that are not concerned with the details of the language features but with the conceptual structure of a generic library. Analyses for library evolution, refactoring, and change impact fall in this category.

Lastly, we remark that bridging the gap to structured specification and algebraic specification in general makes available the theory and tools developed there, and therefore has very practical advantages for research on concepts and concept tools. The Common Algebraic Specification Language (CASL) (Bidoit and Mosses, 2004;

concept:
    concept-header concept-body
concept-header:
    `auto`$_{opt}$ `concept` *identifier* `<`*template-parameter-list*`>`
    refinement-clause$_{opt}$
concept-body:
    `{`*concept-member-spec*$_{opt}$`}`
concept-member-spec:
    associated-function concept-member-spec$_{opt}$
    associated-parameter concept-member-spec$_{opt}$
    associated-requirements concept-member-spec$_{opt}$
    axiom-definition concept-member-spec$_{opt}$

Figure 4.2: Grammar of concepts (adapted from (Gregor and Stroustrup, 2007))

Mosses, 2004), for example, uses the notion of institution to describe the semantics of various structuring constructs, and many of them can be reused for the structuring constructs that occur in C++ concepts. In algebraic specification, there is also a number of tools available that we otherwise would have to write ourselves. As a small example, we show in figure 4.1 the *development graph* constructed by the Heterogeneous Tool Set (Hets) (Mossakowski et al., 2007) for selected concepts that we have taken from the library of ConceptGCC and then translated to CASL.

3 Background

In this section we introduce the notations used later in the paper, and summarize and illustrate the main terminology of C++ concepts on the one hand, the framework of institutions on the other hand.

3.1 C++ Concepts

Templates are the C++ feature on which C++ generic libraries rely the most. A template is, informally, a type or function in which some "holes" are left open and later filled in with particular types, yielding an *instance* of a template. Type checking of templates is divided into two phases: during the first phase, the syntax and the parts that do not depend on *template parameters* are (type-) checked. In the second phase, after template arguments have been provided to fill in the "holes", the remaining parts of the template are checked that depend on the template arguments. Even though templates have proven useful in generic libraries, the two-staged type checking process can result in the well-known, infamous error messages that come from template internals, often referring to code that is not written by the template users.

Although concepts as "sets of requirements" have an existence also outside of C++ (see section 5), they are now mostly known as a C++ feature, in particular after the C++ evolution group has accepted "the concept proposal" (Gregor and Stroustrup, 2007) and this flavor of concepts is likely to be included in the next incarnation of the language. In C++, concepts contain syntactic requirements and can explicitly state semantic requirements in the form of axioms (conditional equations). The main purpose of concepts is to serve as an interface specification for templates.

concept-map-definition:
`concept_map` *concept-id* `{` *concept-map-member-specification*$_{opt}$ `}`
concept-map-member-specification:
simple-declaration concept-map-member-specification$_{opt}$
function-definition concept-map-member-specification$_{opt}$
template-declaration concept-map-member-specification$_{opt}$

Figure 4.3: Grammar of concept maps (Gregor and Stroustrup, 2007, adapted from)

template-declaration:
`template <`*template-parameter-list*`>` *requires-clause*$_{opt}$
declaration

Figure 4.4: Simplified grammar of constrained templates (Gregor and Stroustrup, 2007, adapted from)

The grammar of a concept declaration is shown in figure 4.2. A concept consists of a *concept header* and a *concept body*. A *concept header* specifies the name of the concept, the list of parameters, and a refinement clause; optionally, a concept can be declared as `auto`. The `auto` designation is related to concept maps and is discussed later in this section. *Concept parameters* allow one to specify which C++ entities (in most cases: tuples of types) the concept will be connected to, and the (optional) *refinement clause* allows one to extend existing concepts. The body of the concept consists of specific requirements for *associated types*, *associated functions*, *associated requirements*, and *axioms*. The first two kinds of requirements, *associated types* and *associated functions*, specify the syntactic entities required by a concept, namely, C++ types and functions, including free, member, and template functions. An *associated requirement* is similar to the *refinement clause* and allows one to include requirements of other concepts. In difference to refinement, inclusion makes it possible to put requirements on the associated types of a concept, while refinements can only refer to the parameters. Since refinement and inclusion are similar structuring constructs, sometimes either one of them may be used; guidelines exist for the appropriate selection (Gottschling, 2006).

Once a concept has been defined, the programmer can then state which types fulfill the requirements of a concept using one or more *concept map*s. The grammar of a concept map is given in figure 4.3. A concept map provides an argument for each parameter of a concept, establishing a *concept-id* (that is, a concept with particular arguments) as shown in the grammar. In the body of the concept map, the programmer can specify how a particular type fulfills the requirements of a concept. Note that there is no need to provide proofs for axioms; a concept map simply states that a tuple of types satisfies the semantic requirements represented by axioms. If a concept is marked as `auto`, then the compiler attempts to automatically generate concept maps for that concept (on demand, see the discussion of constrained templates below) if a trivial correspondence exists between the syntax provided by certain types and the syntax required by the concept. Finally, a *template concept map* can cover an unbounded family of types where some arguments of the concept are parameters of the template concept map or depend on them.

Given a set of concepts and concept maps, one can write *constrained templates* as outlined in figure 4.4. A constrained template is just like a usual C++ template with the addition of a *requires clause*. A *requires clause* specifies which concepts must be

met by the potential arguments of the template. In the body of the template, only those types and functions may be used that are guaranteed by the concepts in the *requires clause*. When an argument is supplied to the template, the compiler checks whether the necessary concept maps are defined and issues an error if a concept map is missing for any of the concepts in the *requires clause*. As a consequence, the user rarely (for the cases in which errors may still occur, see Järvi et al., 2006) gets errors from within the template: once a template argument fulfills the requirements of a template it is guaranteed, in most cases, that the body will type check.

Based on concepts from STL (Gregor and Lumsdaine, 2008c), the following examples illustrate how concepts are used.

Example 1 (Concept and concept map).
One of the basic concepts, LESSTHANCOMPARABLE, can be defined as follows:

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

The concept has two parameters but, since those usually are the same, uses the parameter `T` as the default value for the parameter `U`. The only requirement stated is that there must exist an `operator<` to compare two values. The concept is declared as `auto` and the compiler will generate concept maps for any type or two types that have an `operator<`. Below, we show an example of an explicit concept map for a case where a type has not implemented the comparison `operator<`.

```
struct mytype {
  int i;
  mytype(int i_in) : i(i_in) {}
};

concept_map LessThanComparable<mytype> {
  bool operator<(const mytype& a,
                 const mytype& b) {
    return a.i < b.i;
  }
}
```

Once the concept is defined, one can use it to define a generic `min` algorithm:

```
template<typename T>
requires LessThanComparable<T>
inline const T& min(const T& a, const T& b) {
  if (b < a) return b;
  return a;
}
```

This template type-checks because the `operator<` used in the body is provided by the LESSTHANCOMPARABLE concept in the *requires clause*. Now, the `min` template can be used as follows:

```
min(1, 2);  // ok, auto concept map for int ex.
min(mytype(1), mytype(2));  // ok, explicit
                            // concept map
```

Next, we illustrate *template concept maps*, which establish a relation between a concept and an unbounded family of types.

Example 2 (Template concept map).
Suppose, one wanted to declare that all `vector`s are containers. The following template concept map establishes the relation:

```
1 template<typename T>
2 concept_map Container<vector<T> > { }
```

In this concept map, no definitions are provided, and the compiler automatically matches the operations from the CONTAINER concept (defined elsewhere) to the corresponding functions in the class template `vector`.

As a second example, the following concept map declares that any reverse iterator adaptor that adapts a BIDIRECTIONALITERATOR is a BIDIRECTIONALITERATOR itself:

```
1 template<typename Iterator>
2 requires BidirectionalIterator<Iterator>
3 concept_map BidirectionalIterator<reverse_iterator<Iterator> > { };
```

3.2 Categories and Functors

In this section, we define the two basic terms from category theory that are necessary to introduce institutions, namely categories and functors. For a more thorough treatment of category theory, we refer the reader to any of the text books on that subject (see, e.g., Goldblatt, 1984). We also provide examples of categories, including the **Set** and **CAT** categories used in the definition of institutions.

The basic premise of category theory is that we often are not interested in the internals of the objects we study but in their relationships. Thus, a category only states that objects have to exist but does not require an internal structure of the object be provided. The relations between objects are represented by *arrows*. The definition of category is given below (taken from Goldblatt, 1984).

Definition 5 (Category)**.** A category **C** comprises

1. a collection of things called **C**-arrows;
2. a collection of things called **C**-objects;
3. operations assigning to each **C**-arrow f a **C**-object $dom(f)$ (the "domain" of f) and a **C**-object $cod(f)$ (the "codomain" of f). If $a = dom(f)$ and $b = cod(f)$, we display this as
$$f\colon a \to b;$$
4. an operation assigning to each pair (g, f) of **C**-arrows with $dom(g) = cod(f)$, a **C**-arrow $g \circ f$, *the composite of* f *and* g, with
$$dom(g \circ f) = dom(f) \text{ and } cod(g \circ f) = cod(g),$$
i.e.,
$$g \circ f\colon dom(f) \to cod(g),$$
and such that the following conditions are satisfied:

Associative Law: Given arrows $f\colon a \to b$, $g\colon b \to c$, and $h\colon c \to d$ then

$$h \circ (g \circ f) = (h \circ g) \circ f.$$

Identity Law: For every object a there exists an arrow $1_a : a \to a$, such that

$$f \circ 1_a = f = 1_b \circ f \quad \forall \text{ arrows} f : a \to b.$$

Next, we give two example of categories used later in the definition of institutions, the category **Set** and the dual category.

Example 3 (The category Set).
The category **Set** is formed by taking all sets as objects and all functions between sets as arrows. The associativity law is satisfied by the usual function composition.

Example 4 (The dual category).
Given a category **C**. Reversing the arrows in **C** (and keeping the objects) constitutes a new category, the dual category of **C**, denoted by $\mathbf{C}^{\mathbf{op}}$.

Given two categories, one can map the objects and arrows of one category to the objects and arrows of the other. If the mapping preserves the structure of the "source" category, it is called a functor. The definition (from Goldblatt, 1984) of a functor follows.

Definition 6 (Functor)**.** A *functor* **F** from a category **C** to a category **D** is a function that assigns

1. to each **C**-object a, a **D**-object $\mathbf{F}(a)$;
2. to each **C**-arrow $f\colon a \to b$, a **D**-arrow $\mathbf{F}\colon \mathbf{F}(a) \to \mathbf{F}(b)$ such that
   (i) $\mathbf{F}(1_a) = 1_{\mathbf{F}(a)}$, for all **C**-objects a, i.e., the identity arrow on a is assigned the identity on $\mathbf{F}(a)$,
   (ii) $\mathbf{F}(g \circ f) = \mathbf{F}(g) \circ F(f)$, whenever $g \circ f$ is defined.

Example 5 (Functor).
Standard examples are the identity functor and the power set functor.

(a) The *identity* functor $1_{\mathbf{C}}\colon \mathbf{C} \to \mathbf{C}$ has $1_{\mathbf{C}}(a) = a$, $1_{\mathbf{C}}(f) = f$.

(b) The *power set functor*: $\mathbf{P}\colon$ **Set** $\to$ **Set** maps each set A to its powerset $\mathscr{P}$, and each function $f\colon A \to B$ to the function $\mathscr{P}\colon \mathscr{P}(A) \to \mathscr{P}(B)$ that assigns to each $X \subseteq A$ its f-image $f(X) \subseteq B$.

Given a definition of a functor, one can describe the category of all categories, **CAT**.

Example 6 (Category of categories).
The category **CAT** consists of categories[1] as objects and functors as arrows.

[1] For a discussion of foundational issues see Lawvere (1966).

3.3 Institutions

The notion of an *institution* captures the essence of a logic: it defines *signatures*, which represent the available symbols, *sentences* for a particular signature, which represent propositions, and *models*, which represent the "real world." Furthermore, an institution defines a *satisfaction relation* between the sentences of a logic and the models so that satisfaction does not change under renaming—in other words, truth must be invariant under change of notation. Common examples of institutions include equational logic, first-order logic, and higher-order logic. Yet, an institution may also define a non-standard logic, for example based on a programming language semantics, as long as the axioms of an institution are satisfied.

Institutions allow one to formalize a specification framework without having to hard-code a logic for describing the semantics of models. The choice of the logic may be left open, or even entirely excluded if the assumption is made that the logic chosen later will actually form an institution.

We refer the reader to Goguen and Burstall's original paper on institutions (Goguen and Burstall, 1992) and the follow-up papers (Goguen and Roşu, 2002; Sannella and Tarlecki, 1988) for an in-depth treatment of institutions. We limit the presentation here (based on Tarlecki, 2003) to the basic definitions and some examples. By $|\_|$, we denote the set of objects of a category and by $\_^{op}$ the dual category as defined in example 4.

Definition 7 (Institution)**.** An *institution* consists of

1. a category **Sig** of *signatures* and *signature morphisms* as arrows;
2. a functor $\mathbf{Sen}\colon \mathbf{Sig} \to \mathbf{Set}$, assigning
   - to each signature $\Sigma \in |\mathbf{Sig}|$ a set of Σ-*sentences*, and
   - to each signature morphism $\sigma\colon \Sigma \to \Sigma'$ a sentence translation function $\sigma\colon \mathbf{Sen}(\Sigma) \to \mathbf{Sen}(\Sigma')$ written as $\mathbf{Sen}(\sigma)$;
3. a functor $\mathbf{Mod}\colon \mathbf{Sig}^{op} \to \mathbf{CAT}$, assigning
   - to each signature $\Sigma \in |\mathbf{Sig}|$ a category of Σ-*models*, and
   - to each signature morphism $\sigma\colon \Sigma \to \Sigma'$ a *reduct functor* $\mathbf{Mod}(\sigma)$ between categories of models, written as $\_|_\sigma\colon \mathbf{Mod}_{\Sigma'} \to \mathbf{Mod}_\Sigma$;
4. for each $\Sigma \in |\mathbf{Sig}|$ a Σ-*satisfaction relation*

   $$\models_\Sigma \subseteq |\mathbf{Mod}(\Sigma)| \times \mathbf{Sen}(\Sigma)$$

   such that the following condition is satisfied:

   Satisfaction Condition:

   $$M'|_\sigma \models_\Sigma \varphi \iff M' \models_{\Sigma'} \sigma(\varphi)$$

   where $\sigma\colon \Sigma \to \Sigma'$ is a signature morphism, $M' \in |\mathbf{Mod}(\Sigma')|$ is a Σ'-model, and $\varphi \in \mathbf{Sen}(\Sigma)$ is a Σ-sentence.

Goguen and Burstall (1992) provide a short and intuitively understandable explanation of the parts of institutions that we paraphrase here. The essence of the notion of an institution is that a change of signature (by a signature morphism) induces "consistent"

changes in sentences and models—the statement of the Satisfaction Condition. This statement mirrors the basic and familiar fact that the truth of a sentence (in logic) is independent of the symbols chosen to represent its functions and relations, as summed up in the slogan: "Truth is invariant under change of notation." It is important to understand that sentences translate in the same direction as the change of notation, whereas models translate in the opposite direction (the $\mathbf{Sig}^{op}$ makes the reduct functor **Mod** covariant).

The requirements for an institution are very mild and the classical logics can be viewed as institutions. Following, we sketch how the institution of equational logic may be defined, and enumerate other logics that have been described as an institution.

Example 7 (The institution EQ).
Equational logic forms an institution, denoted **EQ**, with an appropriate definition of the category **Sig**, the functors **Sen** and **Mod**, and the satisfaction relation. We outline their definition and refer to the literature (Tarlecki, 1999) for further details.

Algebraic signatures consist of sorts and operations that can be roughly thought of as types and functions in a programming language. Sentences are equations, with expressions constructed of variables and applications of the operations in a given signature. Models are algebras, which assign carrier sets to each sort and a function to each operation in a signature in such way that the functions match the signatures of the operations. Satisfaction relation, finally, states that an algebra satisfies an equation if, and only if, both expressions of the equation evaluate to the same value under the assignment of carrier sets and functions of the given algebra for any sort-correct assignment of values to the variables. In a programming language analogy, algebras can be approximately compared to function implementations and values, while signature components, sorts and operations, can be compared to types and function signatures.

1. The category $\mathbf{Sig_{EQ}}$ is the category **AlgSig** of algebraic signatures and their (algebra) morphisms (see (Tarlecki, 1999, sect. 2.5)).
2. The functor $\mathbf{Sen_{EQ}}\colon \mathbf{AlgSig} \to \mathbf{Set}$ gives (see (Tarlecki, 1999, sect. 2.6)):
   - the set of Σ-equations for each $\Sigma \in |\mathbf{AlgSig}|$; and
   - the σ-translation function taking Σ-equations to Σ'-equations for every signature morphism $\sigma\colon \Sigma \to \Sigma'$.
3. The functor $\mathbf{Mod_{EQ}} = \mathbf{Alg}\colon \mathbf{AlgSig}^{op} \to \mathbf{CAT}$ gives:
   - the category $\mathbf{Alg}(\Sigma)$ of Σ-algebras; and Σ-homomorphisms for each $\Sigma \in$ **AlgSig** (see (Tarlecki, 1999, sect. 2.3)), and
   - the reduct functor $\_|_\sigma\colon \mathbf{Alg}(\Sigma') \to \mathbf{Alg}(\Sigma)$ mapping Σ'-algebras and Σ'-homomorphisms to Σ-algebras and Σ-homomorphisms for each signature morphism $\sigma\colon \Sigma \to \Sigma'$ (see (Tarlecki, 1999, sect. 2.5)).
4. For each $\Sigma \in |\mathbf{AlgSig}|$, the satisfaction relation $\models_{\mathbf{EQ},\Sigma} \subseteq |\mathbf{Alg}(\Sigma)| \times \mathbf{Sen_{EQ}}(\Sigma)$ is the usual relation of satisfaction of a Σ-equation by a Σ-algebra (see (Tarlecki, 1999, sect. 2.6)).

Example 8 (Standard logics as institutions).
Examples of logics that have been described as an institution include **EQ** (equational logic), **FOEQ** (first-order logic with predicates and equality), **PEQ**, **PFOEQ** (as before but with partial operations), **HOL** (higher-order logic), and **CASL**, the logic of the specification language CASL (Bidoit and Mosses, 2004; Mosses, 2004).

Further examples of institutions include modal logics, programming language semantics, and many-valued logics (Tarlecki, 2003).

4 Concepts as Specifications

In this section we provide the definition of an institution of concepts—the core contribution of that paper. As we have argued already, the institution-based formalization allows one to view concepts as a specification language where a concept describes a certain class (or set[2]) of implementations. In this view, a concept defines a signature and a set of sentences over that signature that define the semantic properties of the modeling implementations.

Just as it is proposed for C++, we can define the concepts at specification level so, that they can be used to express the syntactic requirements for the modeling implementations but leave the set of semantic sentences implicit. Also as in C++, this set of implicit sentences may be extended through mechanisms with which properties can be stated explicitly. In C++, *axioms* represent such mechanism insofar they allow developers to state properties explicitly, in a logic closely corresponding to conditional equational logic, and we can extend the general definition of a concept institution accordingly. C++ axioms are meant to enable semantic optimizations by a compiler and do not claim completeness in a logic sense. It is a major advantage of institutions that one does not have to claim completeness either—using institutions, one can mirror both implicit sentences and equations but may also switch to another logic for properties that cannot be expressed through equations. The satisfaction relation, lastly, may also be defined in multiple ways: in C++ terms, satisfaction is left to be defined by the programmer, using concept maps. Were one to use a verifying compiler, however, the equations expressed by axioms would be checked against the modeling implementations. Thus, institutions make the power of concepts explicit: every concept can be considered as equipped with semantics, might it be formalized or not. What matters is, that concepts provide a practical way of incorporating semantic statements into real programs.

As currently stated, our definition of concept institutions does not reflect how a concept specification has been constructed. While C++ concepts are parameterized and can be structured through the mechanisms of refinement and inclusion (see section 3.1, *refinement-clause* and *associated-requirements*), the corresponding institution we define does not include any of those constructs—it provides the basic building blocks only. Yet, it is possible to embed those structuring constructs in the framework of institutions: refinements and inclusion can be flattened, and parameterization can be modeled by the notion of a *view* from one specification to another. The library interconnect language (LIL) (Goguen, 1986; Goguen and Levitt, 1983), LILEANNA (Goguen and Tracz, 2000), and the CASL language (Bidoit and Mosses, 2004; Mosses, 2004) are examples of specification languages that provide structuring constructs that work with any institution and can be applied to our theory almost directly.

To describe concepts as institution, one needs to introduce the notions of concept signatures and models, which we do in section 4.1. In section 4.2, we first define a general institution of concepts, then an institution of concepts with equational logic. The latter also illustrates how one can extend the general institution.

[2] Again, we avoid the discussion of foundational issues and refer to the literature instead (Lawvere, 1966).

4.1 Signatures and Models

The purpose of a concept signature is to describe the *syntax* that a concept provides. It is important to distinguish between the syntax of the concept specification language and the C++ syntax, although for practical reasons, the specification syntax in C++ concepts is a subset of the existing C++ syntax. As argued in the introduction to this section, not all constructs of C++ concepts have counterparts in the concept signature, because the structuring constructs of refinement and inclusion can be ignored. For the purpose of a signature, thus, it suffices to consider that parts of a concept with which one can request particular pieces of syntax: the declarations of associated parameters and associated functions (*associated-function* and *associated-parameter*). The parameters of C++ concepts from the concept header (see figure 4.2) are also included in the concept signature; the formal definition of parameterization can be provided separately from the definition of signatures (e.g., as in CASL).

An *associated parameter* requirement in C++ simply states that a type with some particular name is available. An *associated function* requirement can be of three different kinds: a simple function declaration, a function definition, or a function template declaration. In more detail, one can require declarations or definitions of functions, function templates, member functions, and member function templates. In addition, function parameters can be required to be references or `const` references (see Gregor and Stroustrup, 2007 for the full grammar). To simplify the definition of a concept signature, we only consider simple function declarations and do not take into account `const` qualifiers or references; those restrictions are merely technical and could be lifted by adding special, compound sorts that record, in C++-speak, declaration-specifiers and other technical properties of types. Function templates on the other hand would require formalizing polymorphic functions, currently a task still left to do.

In this section, we first define concept signatures and show that they form a category, **CSig**. Then we define concept models and show that they, too, form a category, **CMod**.

4.1.1 Concept Signatures

We recall the definition of an indexed set.

Definition 8 (Indexed set)**.** An *indexed set* $\mathscr{I}$ is a mapping from a set of indices $\mathscr{I}_I$ to a set of carrier sets $\mathscr{I}_C$.

Notation: We denote the set of indices of an indexed set $\mathscr{I}$ by $\langle\mathscr{I}\rangle$. Given an index $i \in \langle\mathscr{I}\rangle$ we write $\mathscr{I}_i$ for the corresponding carrier set $\mathscr{I}(i)$ and $\|\mathscr{I}\|$ for the union of all carrier sets.

Concept signatures (and signature fragments) are defined as in multi-sorted algebras. They consist of *sorts* and *operations*[3].

Definition 9 (Concept signature fragment and concept signature)**.** (a) A *concept signature fragment* Δ is a tuple (S, O) where S is a finite set of sorts and O is a set of operation names indexed by operation profiles, written w, s or ws, such that w is a (possibly empty) sequence of sorts and s is a sort.
(b) A *concept signature* Σ is a concept signature fragment (S, O) where every operation profile ws fulfills the following conditions:

[3] For clear distinction, we call the elements of concept signatures *operations* and the C++ functions simply *functions*.

- $w \in S^*$,
- $s \in S$.

Notation: $S(\Sigma)$ ($S(\Delta)$ for signature fragments) is the set of sorts S and $O(\Sigma)$ ($O(\Delta)$ for signature fragments) is the set of operations O indexed by operation profiles.

Intuitively, the sorts of a concept signature correspond to C++ types and the operations of a signature correspond to C++ functions. Later on, this connection will be made explicit in definitions of concept models.

Example 9 (Concept signature).
Suppose a concept `Iter`, defined as follows:

```
concept Iter<T> {
  typename value_type;
  void operator++(T);
  value_type operator*(T);
}
```

The signature Σ for the concept `Iter` consists of the set of sorts

$$S(\Sigma) = \{value\_type, T, void\}$$

and the two operations

$$O(\Sigma)_{T,void} = \{++\}, O(\Sigma)_{T,value\_type} = \{*\}.$$

The signature has three sorts, corresponding to the C++ types used in the `Iter` concept. Note that the parameter type `T` has a corresponding sort in the signature and that the `void` type has a corresponding sort *void* as well. Instead of including sorts corresponding to the built-in C++ types, one could define a signature Σ_{C++} that would provide all sorts and operations corresponding to types and functions defined in the C++ standard and define all other signatures as extensions of Σ_{C++}.

Signature morphisms map sorts to sorts, operation names to operation names, and, accordingly to the sort mapping, operation profiles to operation profiles. They are defined in the standard way.

Definition 10 (Signature morphism)**.** Let Σ, Σ' be concept signatures. A *signature morphism* $\sigma\colon \Sigma \to \Sigma'$ is a pair of functions (σ^S, σ^O) where:

$$\sigma^S\colon S(\Sigma) \to S(\Sigma')$$

and

$$\sigma^O\colon \langle O(\Sigma)\rangle \to (\|O(\Sigma)\| \to \|O(\Sigma')\|)$$

such that

$$\sigma^O_{ws}\colon O(\Sigma)_{ws} \to O(\Sigma')_{\sigma^S(ws)} \qquad \forall ws \in \langle O(\Sigma)\rangle$$

with $\sigma^S(ws) = \sigma^S(s_i)_i, \sigma^S(s)$ for $s_i \in w$ (w can be empty).

We note the following simple properties of signature morphisms.

Proposition 1.
Let $\Sigma, \Sigma', \Sigma''$ be signatures and let $\sigma\colon \Sigma \to \Sigma'$, $\tau\colon \Sigma' \to \Sigma''$, and $\rho\colon \Sigma \to \Sigma''$ be signature morphisms.

(a) *We define the composition* $\rho := \tau \circ \sigma$ *as a pair of mappings* (ρ^S, ρ^O) *with*

$$\rho^S := \tau^S \circ \sigma^S$$

and

$$\rho^O := \{ws \mapsto \tau^O_{\sigma^S(ws)} \circ \sigma^O_{ws} \mid ws \in \langle O(\Sigma)\rangle\}.$$

The composition ρ *is a signature morphism* $\rho : \Sigma \to \Sigma''$.

(b) *There exists an identity signature morphism* $1_\Sigma : \Sigma \to \Sigma$.

(c) $\rho \circ (\tau \circ \sigma) = (\rho \circ \tau) \circ \sigma$.

(d) $\sigma \circ 1_\Sigma = \sigma$ *and* $1_{\Sigma'} \circ \sigma = \sigma$.

Proof.

(a) It suffices to show that $\rho^S : S(\Sigma) \to S(\Sigma''')$ and that $\rho^O : \langle O(\Sigma)\rangle \to (\|O(\Sigma)\| \to \|O(\Sigma'')\|)$ such that $\forall ws \in \langle O(\Sigma)\rangle, \rho^O_{ws} : O(\Sigma)_{ws} \to O(\Sigma'')_{\rho^S(ws)}$. By definition of function composition, the requirement on ρ^S follows directly. The rest follows from the properties of the function composition $\tau^O_{\sigma^S(ws)} \circ \sigma^O_{ws}$.

(b) The identity morphism $1_\Sigma : \Sigma \to \Sigma$ can be constructed as the pair of the identity function on the set of sorts, $1_S : S \to S$, and the function that maps every operation profile to the identity map on operation names $\{ws \mapsto 1_{O(\Sigma)_{ws}} \mid ws \in \langle O(\Sigma)\rangle\}$.

(c) For the sort mapping, associativity follows from the associativity of functions. For $\rho \circ (\tau \circ \sigma)$, the operation mapping $(\rho \circ (\tau \circ \sigma))^O$ yields $\{ws \mapsto \rho^O_{(\tau\circ\sigma)^S(ws)} \circ (\tau^O_{\sigma^S(ws)} \circ \sigma^S_{ws}) \mid ws \in \langle O(\Sigma)\rangle\}$. Analogously, $((\rho \circ \tau) \circ \sigma)^O$ yields $\{(\rho^O_{(\tau\circ\sigma)^S(ws)} \circ \tau^O_{\sigma^S(ws)}) \circ \sigma^S_{ws} \mid ws \in \langle O(\Sigma)\rangle\}$. The proposition now follows from the associativity of function composition.

(d) By definition of the identity morphisms and the composition of signature morphisms. □

We can now show that concept signatures form a category.

Proposition 2.
The class of concept signatures as objects together with signature morphisms as arrows form a category, denoted **CSig**.

Proof. Show that for each object (signature), there is an identity arrow (morphism), that arrows can be composed, and that composition is associative. Clear by proposition 1. □

4.1.2 Concept Models

Every concept signature introduces syntax that in turn determines the set of possible C++ implementations, or, in more general terms, the *models* that are admissible under a signature; in this subsection, we formally introduce the notion of a model. A simple way to define the models of a concept signature is to require a set of C++ types and functions, named as the sorts and operations in the signature, so that C++ functions match the profiles of the operations of the signature. Such definition, however, binds

C++ names unnecessarily closely to the signature symbols. To avoid such close coupling, we define a model as a mapping from signature symbols to particular C++ types and functions.

First we introduce the sets of all C++ types and functions:

- C++–*Types*,
- C++–*Funs*,

where the set C++–*Funs* comprises free functions, member functions, and operators. We write *args(f)* to refer to the tuple of argument types and *ret(f)* to the return type of f.

Definition 11 (Σ-model)**.** Let $\Sigma = (S, O)$ be a signature. A Σ-*model* M is a tuple (T, F) where:

$$T\colon S(\Sigma) \to \textit{C++–Types},$$
$$F\colon \langle O(\Sigma)\rangle \to (\|O(\Sigma)\| \to \textit{C++–Funs})$$

such that for all $ws \in \langle O(\Sigma)\rangle, f \in O(\Sigma)_{ws}$,

$$\mathit{args}(F_{ws}(f)) = T(w) \text{ and } \mathit{ret}(F_{ws}(f)) = T(s).$$

A Σ-model maps every operation $o \in O(\Sigma)_{ws}$ to a C++ function $F_{ws}(o)$ with (argument and return) types that match exactly the image of the profile of o. Note that it would suffice to require that the images of the sorts in the profile of o can be coerced to the corresponding types of $F_{ws}(o)$, since the C++ rules for overload resolution would still select the same $F_{ws}(o)$; the reader may therefore read the equality of types modulo coercion. Also note that models are defined only for signatures and not for signature fragments, i.e., the sorts used in the operation profiles must be in the set of sorts of a signature $S(\Sigma)$.

The sorts to types and the operations to functions mappings of a model are total functions. The totality of the mappings mirrors the design of C++ concepts where a concept map has to provide a concrete type or function for every symbol in the signature of the concept. In the C++ concept language, the mapping from concept symbols to types and functions may be left out in some situations, but then the compiler generates the missing pieces and the mapping is still total.

Example 10 (Σ-model).
Let $\Sigma = (S, O)$ be a signature with

$$S = \{\mathit{value\_type}, \mathit{void}, T\}$$

and

$$O = \{\{{+}{+}\}_{(T,\mathit{void})}, \{*\}_{(T,\mathit{value\_type})}\}).$$

In the following, we use a notation where C++ functions are subscripted with their argument and return types.

(a) Let $M = (T, F)$ where:

$$\begin{aligned} T =& \{\mathit{value\_type} \mapsto \texttt{int}, \mathit{void} \mapsto \texttt{void}, T \mapsto \texttt{int}*\},\\ F =& \{(T, \mathit{void}) \mapsto \{{+}{+} \mapsto {+}{+}_{(\texttt{int}*,\texttt{void})}\},\\ & \quad (T, \mathit{value\_type}) \mapsto \{* \mapsto *_{(\texttt{int}*,\texttt{int})}\}\}. \end{aligned}$$

Then M is a Σ-model. The model M maps the sort *void* to the C++ type `void` although the formulation of Σ-models does not enforce such mapping. Using *structured specifications*, it is possible, however, to fix the mapping of those sorts that correspond to built-in types. The model furthermore maps the operations $++$ and $*$ to the C++ functions `++` and `*`, which, again, is not enforced—the signature operations and C++ functions are entities of different domains, and the direct correspondence between the signature operation names and C++ function names is coincidental.

(b) Let $M_1 = (T_1, F_1)$ where:

$$\begin{aligned} T_1 =& \{\mathit{value\_type} \mapsto \texttt{mytype}, \mathit{void} \mapsto \texttt{void}, T \mapsto \texttt{iterator}\}, \\ F_1 =& \{(T, \mathit{void}) \mapsto \{++ \mapsto ++_{(\texttt{iterator},\texttt{void})}\}, \\ & (T, \mathit{value\_type}) \mapsto \{* \mapsto *_{(\texttt{iterator},\texttt{mytype})}\}\}. \end{aligned}$$

Then M_1 is a Σ-model.

(c) Let $M_2 = (T, F_2)$ where:

$$\begin{aligned} F_2 = \{&(T, \mathit{void}) \mapsto \{++ \mapsto ++_{(\texttt{float},\texttt{void})}\}, \\ &(T, \mathit{value\_type}) \mapsto \{* \mapsto *_{(\texttt{int}*,\texttt{int})}\}\}. \end{aligned}$$

Then M_2 is not a Σ-model because the sorts mapping does not match with the mapping of operation profiles.

We have to show that Σ-models form a category since the definition of an institution requires that a signature be mapped to the corresponding category of models. The requirement of a category, however, is of technical nature; a category of models makes the definition more consistent with the categorical view and it makes every liberal institution (not defined in this paper) an institution (Goguen and Burstall, 1992). In principle, a set of models could be used instead.We will therefore not attempt to identify any structure in models and instead show that models trivially form a category, with identity mappings as the only arrows—in effect, the structure of any Σ-models category mimics the simple structure of a set. More intuitively speaking, we do not attempt to establish any semantic relationships between C++ types and functions.

Proposition 3.
The class of Σ-models as objects together with identity mappings as arrows form a category, denoted $\mathbf{CMod}_\Sigma$.

Proof. Trivial. □

Σ-models are mapped in the opposite direction to signature morphisms using the *reduct* mapping. The term opposite direction here means that if we change the notation of a concept by some translation, all models of the modified concept can be reduced to models of the original concept but not necessarily vice versa. The *reduct* mapping is defined in the usual way.

Definition 12 (Σ-model reduct)**.** Let $\Sigma = (S, O)$ be a signature, $\sigma : \Sigma \to \Sigma'$ be a signature morphism, and $M' = (T', F') \in |\mathbf{CMod}_{\Sigma'}|$ be a Σ'-model.

(a) The σ-reduct of $M'|_\sigma$ is a Σ-model $M = (T, F) \in |\mathbf{CMod}_\Sigma|$ where:

$$T'|_\sigma = T = T' \circ \sigma^S$$
$$F'|_\sigma = F = \{ws \mapsto F'_{\sigma(ws)} \circ \sigma^O_{ws} \mid ws \in \langle O(\Sigma)\rangle\}.$$

(b) Given an identity mapping $1_{M'}$, we define the σ-reduct $1_{M'}|_\sigma$ of the mapping as $1_{M'|_\sigma}$.

We note that the reduct mapping has the required functor property.

Proposition 4.
Let Σ, Σ' be signatures, $\sigma\colon \Sigma \to \Sigma'$ be a signature morphism. The σ-reduct is a functor $\_\_|_\sigma\colon \mathbf{CMod}_{\Sigma'} \to \mathbf{CMod}_\Sigma$.

Proof. Let $M' \in \mathbf{CMod}_{\Sigma'}$ be a Σ'-model. $M'|_\sigma \in |\mathbf{CMod}_\Sigma|$ holds by definition of $M'|_\sigma$. Furthermore, $1_{(M')}|_\sigma = 1_{(M'|_\sigma)}$ by definition of $\_\_|_\sigma$. This is sufficient to show the functor property since all arrows in **CMod** are identity arrows. □

As discussed before in this subsection, every signature induces the category of possible models. Furthermore, every signature morphism maps to the corresponding reduct functor that takes models of the target signature to models of the source signature (opposite to the direction of the signature morphism). We introduce the **CMod** functor that combines these two mappings, to keep signatures and models consistent under renaming.

Proposition 5.
Let $\Sigma \in |\mathbf{CSig}|$ be a signature and $\sigma\colon \Sigma \to \Sigma'$ be a signature morphism. Define **CMod** *as the mapping that gives*

- *the category $\mathbf{CMod}_\Sigma$ of Σ-models for every $\Sigma \in |\mathbf{CSig}|$; and*
- *the reduct functor $\_\_|_\sigma\colon \mathbf{CMod}_{\Sigma'} \to \mathbf{CMod}_\Sigma$ for every $\sigma\colon \Sigma \to \Sigma'$.*

Then $\mathbf{CMod}\colon \mathbf{CSig}^{op} \to \mathbf{CAT}$ *is a functor.*

Proof. Let $\Sigma, \Sigma', \Sigma'' \in |\mathbf{CSig}|$ be signatures and $\sigma\colon \Sigma \to \Sigma'$, $\tau\colon \Sigma' \to \Sigma''$ be signature morphisms.

(a) $\mathbf{CMod}(1_\Sigma) = 1_{\mathbf{CMod}_\Sigma} = 1_{\mathbf{CMod}(\Sigma)}$ holds by definition of $\_\_|_\sigma$ where $\sigma = 1_\Sigma$.

(b) $\mathbf{CMod}(\tau \circ \sigma) = \mathbf{CMod}(\sigma) \circ \mathbf{CMod}(\tau) = \_\_|_\sigma \circ \_\_|_\tau$ follows from function composition and the contravariance of reduct (proposition 4).

□

4.2 Concept Institutions

In this section we give the two most important definitions of this paper, the general institution of concepts, **C**, and the institution of concepts with equational logic, $\mathbf{C}_{\mathbf{EQ}}$.

4.2.1 The General Institution of Concepts

In the general institution of concepts that we define in this section, we define the set of (Σ-) sentences as the set of (Σ-) models:

$$\mathbf{CSen}_\Sigma := \mathbf{CMod}_\Sigma.$$

This choice may seem strange at first. After all, sentences are supposed to be some kind of propositions and models do not appear to be such in an obvious way. Intuitively, however, the identity of sentences and models corresponds to the use of C++ concept maps (section 3.1), where one declares that a certain C++ implementation (i.e., a tuple of types with some functions) models a concept, without having to prove explicitly stated semantics: by definition, the concept map establishes the satisfaction relation, and the way to represent that relation is to simply state which models meet the particular requirements.

We now have to define what happens when the notation of a concept is changed, as it the case when a certain signature morphism is applied to the signature of the concept. Since the sentences that describe the meaning of a concept are simply models, all we have to do is to find the models induced by the new notation that reduce, along the change of notation, to the models of the concept under the old notation; to that end, we introduce the notion of sentence translations. There is not any particular feature in C++ concepts that directly corresponds to sentence translation, but one can think of a situation when some names in a concept are changed and the concept maps have to be updated to match the change in the concept. In the following, we use φ to denote a sentence. The definition of sentence translation is independent of the particular definition of models and relies on the properties of the previously defined functor **CMod** (proposition 5).

Definition 13 (Sentence translation)**.** Let $\Sigma, \Sigma' \in |\mathbf{CSig}|$ be signatures, $\sigma\colon \Sigma \to \Sigma'$ be a signature morphism, and $\varphi = M \in |\mathbf{CMod}(\Sigma)|$ be a Σ'-sentence. We define the σ-*translation* of the sentence φ, denoted $\sigma(\varphi)$, as the Σ' model $M' \in |\mathbf{CMod}(\Sigma')|$ such that $M'|_\sigma = M$.

Example 11 (Σ-sentence and Σ-sentence translation).
Let Σ be a signature as in example 10, $\Sigma' = (T, O)$, where:

$$\begin{aligned} T =& \{target, unit, pointer\}, \\ O =& \{\{advance\}_{(pointer,unit)}, \{access\}_{(pointer,target)}\}, \end{aligned}$$

be a signature, and $\sigma\colon \Sigma \to \Sigma'$ be a signature morphism, where:

$$\begin{aligned} \sigma = (&\{value\_type \mapsto target, void \mapsto unit, T \mapsto pointer\}, \\ &\{(T, void) \mapsto \{++ \mapsto advance\}, \\ &(T, value\_type) \mapsto \{* \mapsto access\}\}). \end{aligned}$$

(a) Let T, F be as in example 10 (a). $M = (T, F)$ is a Σ-sentence and its σ-translation is the tuple $M' = (T', F')$ such that

$$\begin{aligned} T' =& \{target \mapsto \texttt{int}, unit \mapsto \texttt{void}, pointer \mapsto \texttt{int*}\}, \\ F' =& \{(pointer, unit) \mapsto \{advance \mapsto ++_{(\texttt{int*},\texttt{void})}\}, \\ & (pointer, target) \mapsto \{access \mapsto *_{(\texttt{int*},\texttt{int})}\}\}. \end{aligned}$$

Obviously, the Σ'-model M' fulfills the condition of sentence translation, namely, $M'|_\sigma = M$.

(b) Let T_1, F_1 be as in example 10 (b). The Σ-model $M_1 = (T_1, F_1)$ is a Σ-sentence and its σ-translation is the tuple $M_1' = (T_1', F_1')$ such that

$$F_1' = F'$$

and

$$T_1' = \{target \mapsto \texttt{mytype}, unit \mapsto \texttt{void}, pointer \mapsto \texttt{iterator}\}.$$

Since $M_1'|_\sigma = M$, M_1' is the σ-translation of M.

(c) Let T, F_2 be as in example 10 (c). Because $M_2 = (T, F_2)$ is not a Σ-model, M_2 is not a Σ-sentence.

Analogously to the functor **CMod** for models, we introduce a mapping **CSen** for sentences and show its functor property.

Proposition 6.
Define **CSen** *as the mapping that gives:*

- *the set of all Σ-sentences for every $\Sigma \in |\mathbf{CSig}|$; and*
- *the σ-translation function taking Σ-sentences to Σ'-sentences for every signature morphism $\sigma\colon \Sigma \to \Sigma'$.*

Then $\mathbf{CSen}\colon \mathbf{CSig}^{op} \to \mathbf{Set}$ *is a functor.*

Proof. We need to show that the identity arrow and the composition of arrows are preserved by **CSen**. Let $\Sigma, \Sigma', \Sigma'' \in |\mathbf{CSig}|$ be signatures and $\sigma\colon \Sigma \to \Sigma'$, $\tau\colon \Sigma' \to \Sigma''$ be a signature morphism. For the identity arrow, it holds:

$$\begin{aligned}
&\mathbf{CSen}(1_\Sigma) \\
&= \{M \mapsto M' \mid M \in \mathbf{CMod}_\Sigma, M' \in \mathbf{CMod}_{\Sigma'}, M'|_{1_\sigma} = M\} \\
&= 1_{\mathbf{CMod}_\Sigma}
\end{aligned}$$

because $M'|_{1_\sigma} = M'$ by definition of reduct. Basic transformations show that composition is preserved:

$$\begin{aligned}
&\mathbf{CSen}(\tau \circ \sigma) = \\
&\quad \{M \mapsto M'' \mid M \in \mathbf{CMod}_\Sigma, M'' \in \mathbf{CMod}_{\Sigma''}, \\
&\qquad M''|_{\tau\circ\sigma} = M\} = \\
&\quad \{M \mapsto M'' \mid M \in \mathbf{CMod}_\Sigma, M'' \in \mathbf{CMod}_{\Sigma''}, \\
&\qquad (M''|_\tau)|_\sigma = M\} = \\
&\quad \{M \mapsto M'' \mid M \in \mathbf{CMod}_\Sigma, M' \in \mathbf{CMod}_{\Sigma'}, \\
&\qquad M'' \in \mathbf{CMod}_{\Sigma''}, M''|_\tau = M', M'|_\sigma = M\} = \\
&\quad \{M' \mapsto M'' \mid M' \in \mathbf{CMod}_{\Sigma'}, M'' \in \mathbf{CMod}_{\Sigma''}, \\
&\qquad M''|_\tau = M'\} \circ \\
&\quad \{M \mapsto M' \mid M \in \mathbf{CMod}_\Sigma, M' \in \mathbf{CMod}_{\Sigma'}, \\
&\qquad M'|_\sigma = M\} = \\
&\mathbf{CSen}(\tau) \circ \mathbf{CSen}(\sigma)
\end{aligned}$$

□

For the general concept institution, a model satisfies exactly itself when used as a sentence.

Definition 14 (Satisfaction relation ($\models_\Sigma$))**.** Let $\Sigma \in |\mathbf{CSig}|$ be a signature, $\varphi \in \mathbf{CSen}(\Sigma)$ be a Σ-sentence, and $M \in \mathbf{CSen}(\Sigma)$ be a Σ-model. We define the *satisfaction relation*

$$\models_\Sigma \in |\mathbf{CMod}(\Sigma)| \times \mathbf{CSen}(\Sigma)$$

such that

$$M \models_\Sigma \varphi \text{ iff } M = \varphi.$$

Intuitively, this definition of satisfaction mirrors the effect of C++ concept maps, through which a developer *asserts* that a type models a certain C++ concept.

The main result of this subsection is the following proposition of a concept institution.

Proposition 7.
The category **CSig**, *the functor* **CSen**, *the functor* **CMod**, *and the satisfaction relation* $\models_\Sigma$ *for each signature* $\Sigma \in |\mathbf{CSig}|$ *form an institution, denoted by* **C**.

Proof. By proposition 2, proposition 5, and proposition 6, **CSig**, **CMod**, and **CSen** have the required properties. What remains to show is that the satisfaction condition holds.

Let $\Sigma, \Sigma' \in |\mathbf{CSig}|$ be signatures, $\sigma : \Sigma \to \Sigma'$ be a signature morphism, $\varphi \in \mathbf{CSen}(\Sigma)$ be a Σ-sentence, and $M' = (T', F') \in |\mathbf{CMod}_{\Sigma'}|$ be a Σ' model.

($\Leftarrow$) Assume $M' \models_{\Sigma'} \sigma(\varphi)$. From definition 14 of Σ'-satisfaction and definition 13 of σ-sentence translation $M' = \sigma(\varphi)$, $M'|_\sigma = \varphi$, thus $M'|_\sigma \models_\sigma \varphi$.

($\Rightarrow$) Assume $M'|_\sigma \models_\Sigma \varphi$. Again, from the definition of Σ-satisfaction and the σ-sentence translation $M'|_\sigma = \varphi$, it follows $M' = \sigma(\varphi)$, thus $M' \models_{\Sigma'} \sigma(\varphi)$. □

4.2.2 Concepts with Logics

In this section we refine the previous definition to define the institution of concepts with equational logic that corresponds to concepts with axioms. In contrary to the general institution of concepts, in the institution of concepts with equational logic the satisfaction relation is not established by the developer. Instead, a model satisfies a concept when it fulfills all the axioms of the concept, that is, there exists a proof, either automatically generated or explicitly provided, that certain types and functions satisfy the equations of the concept.

We follow the standard definition of Σ-equations as two elements of a term algebra defined over a signature Σ and a set of variables. Intuitively, a Σ-term corresponds to an expression composed of operations defined in Σ and some variables. The notion of a term algebra simplifies the view of expressions in C++, as the particular form that expressions can take is inessential to our presentation. Since concept signatures are defined as in standard algebraic signatures, the standard definition of a term algebra just carries over (Tarlecki, 2003). For the rest of this section, we assume the Σ-algebra $T_\Sigma(X)$ of terms with a set of variables, X. The set of all terms of $T_\Sigma(X)$ is denoted as $|T_\Sigma(X)|$ and the set of all terms of sort $s \in S(\Sigma)$ is denoted $|T_\Sigma(X)|_s$.

First, we define the σ-translation of sentences for any signature morphism $\sigma : \Sigma \to \Sigma'$.

Definition 15 (Σ-term translation)**.** Let $\Sigma = (S, O)$ and $\Sigma' = (S', O')$ be signatures, $\sigma\colon \Sigma \to \Sigma'$ be a signature morphism, and X be an S-sorted set of variables such that $X_s \neq X_{s'}$ for any $s, s' \in S$ (this condition simplifies presentation). Define $\sigma(X) = X'$ where $X'_{s'} = \bigcup_{\sigma(s)=s'} X_s$ for $s' \in S'$. The *σ-translation* of a Σ-term $t \in |T_\Sigma(X)|$, denoted as $\sigma(t)$, is the Σ'-term

$$\sigma(t) \in |T_{\Sigma'}(X')|,$$

obtained by replacing in t each operation name $o \in O(\Sigma)_{ws}$ by $\sigma^O_{ws}(o)$ for all $ws \in \langle O(\Sigma)\rangle$.

Given a Σ-term and a Σ-model, we need to be able to translate the expression represented by the Σ-term to a particular C++ expression determined by the model. That means, given a model, which is a map from symbols of the signature to C++ entities, we need to translate the expressions composed of the symbols of the signature to expressions composed of the C++ entities.

Definition 16 (Σ-term C++-valuation)**.** Let Σ be a signature, $M = (T, F)$ be a Σ-model, and $t \in T_\Sigma(X)$ be a Σ-term for a variable set X. The *C++-valuation* of the term t, denoted $v_{C++}(t, M)$, is defined in the same way as the σ-translation (definition 15), but with the mapping σ^S replaced by T and the mapping σ^O replaced by F.

Next, we introduce the notion of a Σ-equation in the standard way (e.g., see Tarlecki, 2003). Variables in equations are taken from an infinite but fixed set of variables $\mathscr{X}$ (i.e., the set of all possible variable names). An equation represents an axiom in a C++ concept and consists of a left- and right-hand side, each an expression that may use some variables.

Definition 17 (Σ-equation)**.** A *Σ-equation* φ is a triple (X, L, R) where $L, R \in |T_\Sigma(X)|_s$ for some $s \in S(\Sigma)$ and $X_s \in \mathscr{X}$ for all $s \in S(\Sigma)$.

If the notation of a concept changes, it is possible to change the equations accordingly so that the properties expressed by the equations do not change. The following definition describes how to change an equation when the notation of a concept changes.

Definition 18 (Σ-equation translation)**.** Given a signature morphism $\sigma\colon \Sigma \to \Sigma'$ and a Σ-equation $\varphi = (X, L, R)$, the *σ-translation* of the Σ-equation $\sigma(\varphi)$ is defined as the Σ'-equation $(\sigma(X), \sigma(L), \sigma(R))$.

Now, we can introduce the mapping $\mathbf{C_{EQ}Sen}$ from signatures to sentences and from signature morphisms to sentence translations, and show its functor property.

Proposition 8.
Define $\mathbf{C_{EQ}Sen}$ *as the mapping that gives:*

- *the set of all Σ-equations for every* $\in |\mathbf{CSig}|$*; and*
- *the σ-equation translation for every signature morphism*

$$\sigma\colon \Sigma \to \Sigma'.$$

Then $\mathbf{C_{EQ}Sen}\colon \mathbf{CSig}^{op} \to \mathbf{Set}$ *is a functor.*

Proof. Routine. □

As the last step on the way to forming the institution $\mathbf{C_{EQ}}$, we define the following satisfaction relation between models and sentences. It uses the functor $\mathbf{C_{EQ}Sen}$ just

defined and the functor **CMod** from proposition 5. Intuitively, the satisfaction relation states that a model satisfies an equation if, and only if, the right- and left-hand side expressions of the equations can always be substituted for another under the assignment of C++ entities to the symbols of a concept that the given model contains. In other words, the expressions must always have the same semantics under the given model.

Definition 19 (Satisfaction relation ($\models_{C_{EQ},\Sigma}$))**.** Let $M = (T, F)$ be a Σ-model and $\varphi = (X, L, R)$ be a Σ-equation. For each $\Sigma \in |\mathbf{CSig}|$, the satisfaction relation

$$\models_{\mathbf{C_{EQ}},\Sigma} \subseteq |\mathbf{CMod}(\Sigma)| \times \mathbf{C_{EQ}Sen}(\Sigma)$$

is defined as follows:

$$M \models_{C_{EQ},\Sigma} \varphi \text{ iff the valuation } v_{C++}(L, M) \rightsquigarrow v_{C++}(R, M),$$

where $\rightsquigarrow$ denotes a sequence of rewrites through which $v_{C++}(L, M)$ can be rewritten to $v_{C++}(R, M)$ in any C++ program.

Now we have all parts in place that are needed to define the institution $\mathbf{C_{EQ}}$ of concepts with equational logic.

Proposition 9.
The category **CSig**, *the functor* $\mathbf{C_{EQ}Sen}$, *the functor* **CMod**, *and the satisfaction relation* $\models_{\mathbf{C_{EQ}},\Sigma}$ *for each signature* $\Sigma \in |\mathbf{CSig}|$ *form an institution, denoted by* $\mathbf{C_{EQ}}$.

Proof. By proposition 2, proposition 5, and proposition 8, **CSig**, **CMod**, and $\mathbf{C_{EQ}Sen}$ have the required properties. What remains to show is that the satisfaction condition holds.

Let $\Sigma, \Sigma' \in |\mathbf{CSig}|$ be signatures, $\sigma\colon \Sigma \to \Sigma'$ be a signature morphism, $\varphi = (X, L, R) \in \mathbf{C_{EQ}Sen}(\Sigma)$ be a Σ-sentence, and $M' = (T', F') \in |\mathbf{CMod}_{\Sigma'}|$ be a Σ' model. We need to show

$$M'|_\sigma \models_{C_{EQ},\Sigma} \varphi \iff M' \models_{C_{EQ},\Sigma'} \sigma(\varphi).$$

($\Leftarrow$) Assume $M' \models_{C_{EQ},\Sigma'} \sigma(\varphi)$. From definition 14 (satisfaction), it suffices to show that the C++-valuation of the terms of φ with model $M'|_\sigma$ is the same as the C++-valuation of terms of $\sigma(\varphi)$ with model M'. To do that, we will show that C++-valuation is the same component-wise: the set of variables X and every operation name o is C++-valuated to the same C++ expression. From definition 15, 16 and from function composition, the following identities hold:

$$X''_{s''} = \bigcup_{T'(s')=s''} X_{s'} = \bigcup_{T'(s')=s''} \left(\bigcup_{\sigma(s)=s'} X_s \right) = \bigcup_{T'(\sigma(s))} X_s = \bigcup_{(T'\circ\sigma)(s))} X_s$$

$$T''_{w's'}(f') = T'_{\sigma^S(ws)}(\sigma^O_{ws}(f)) = (T'_{\sigma^S(ws)} \circ \sigma^O_{ws})(f)$$

Therefore, following from definition 12, $M'|_\sigma \models_{C_{EQ},\Sigma} \varphi$, qed.

($\Rightarrow$) Follows with arguments similar to the ones for the other direction of the implication. □

5 Related Work

Our work is related to previous formalizations of concepts. The origins of C++ concepts can be traced back to Kapur and Musser who introduced the idea of generic system components in the specification language Tecton (Kapur and Musser, 1992). One of the main goals of Tecton is to reduce the need for proof. Consequently, the design of Tecton concentrates on the ease of replacing parts of a specification without having to repeat full proofs, but there is no direct link to a programming language (multi-sorted algebras as models). In subsequent developments, it was shown how to specify parts of STL in Tecton (Musser et al., 2000) and how to encode such specification in C++ (Schupp et al., 2000). Yet, the connection between specification and software is only implicit in such setting while our approach lays foundations for a direct connection. The formalization of concepts by Willcock et al. (2004) draws on ideas from Tecton but is meant to be applicable more directly to programming languages. Their formalization provides a way of describing the signature of a concept, the existence of a logical admissibility conditions (expressed as a black-box predicate), and a *lookup* function for language-specific model lookup. Our institution of concepts makes the parts explicit: models are defined to be specific C++ implementations, signatures determine sentences, and models satisfy sentences. In essence, the two formalizations are comparable but ours has a goal of treating concepts as specifications of C++ programs while theirs captures concepts more at the level of a programming language construct. The most recent formalization of concepts, due to Siek and Lumsdaine (2005), extends system F (Girard, 1972) by concepts. Their goal is quite different from ours: they show how concepts can be made a part of a type system.

Our work is related to Goguen's ideas of parameterized programming and hyper-programming (Goguen, 1990, 1989) where code specification can be directly used to produce new code modules. In particular, our research has been inspired by LILEANNA (Library Interconnect Language extended with Anna) (Goguen and Tracz, 2000), a language for specification and manipulation of ADA (Ada) modules. LILEANNA depends on abstraction from Ada semantics: it assumes a fixed institution (which is not specified) to develop a formalism and an implementation for a module interconnect language (MIL) (Prieto-Diaz and Neighbors, 1986). Our formalization of concepts as institutions makes it possible to define a similar language for describing and performing the evolution of generic software libraries. An important difference between our formalization and LILEANNA is the level and purpose of modules and concepts: while modules describe complete software packages, the purpose of concepts is to capture requirements of particular generic algorithms.

Finally, our work is related to the recent developments in the field of algebraic specifications (Wirsing, 1990). Specifically, our institution for concepts goes in the direction proposed by Aspinall and Sannella (2002) of making specifications directly applicable to programming languages. In the same spirit, our formalization is related to Schröder and Mossakowski's HasCASL 2002, which is an extension of CASL based on an institution corresponding to the Haskell language (Jones, 2003). Their formalization, however, specifies models with semantics so that the properties of models can be proven.

6 Conclusions and Future Work

We have formalized concepts using the notion of institutions. This formalization is based on the formal development of the notions of *concept signatures*, *concept models*, and *concept sentences*. Our general institution of concepts captures the intuitive meaning of concepts where each concept describes a class of its models. It matches the current use of concepts where the semantics are declared implicitly by the concept developer, and the satisfaction relation is stated explicitly via concept maps. Furthermore, we show how to embed a particular logic in concepts. We give an example of a concept institution with equational logic and provide the corresponding satisfaction relation between C++ implementations and equations.

Concept institution, defined in this paper, is the base for further developments. The immediate task is to define structuring constructs, specifically refinement, requirements, and parameterization. This task is closely related to the existing specification tools: the work has been done both in CASL and LILEANNA. The structuring of specifications for concepts will be very similar.

We plan two kinds of further developments. First, we plan to use the formalization for the practical task of describing software library evolution, similarly to the way LILEANNA describes the evolution of modules. Using such description, we can further develop the change impact analysis we have proposed earlier for generic libraries. The second direction of future research is to investigate the use of concepts as a formal specification and verification tool. We will investigate the possibility of a formal framework, similar to the frameworks of HetCASL and CafeOBJ, to allow easy mixing of logics in concepts and thus extend their specification power. In the future, such extended concepts could allow us to consider the semantics of software libraries more concretely.

Acknowledgment

We thank the anonymous reviewers who helped to improve the presentation of the paper.

Paper IV

Multi-Language Library Development
From Haskell Type Classes to C++ Concepts

Multi-Language Library Development
From Haskell Type Classes to C++ Concepts

Marcin Zalewski, Andreas Priesnitz, Cezar Ionescu[1], Nicola Botta[1], and Sibylle Schupp

[1] Potsdam Institute for Climate Impact Research,
Potsdam, Germany
{ionescu|botta}@pik-potsdam.de

Abstract

We define a mapping from generic Haskell specifications to C++ with concepts, a recent extension to C++, that can ultimately be automated. More specifically, we provide a translation from Haskell multi-parameter type classes with functional dependencies to ConceptC++. Our translation consists of three major parts: the division of Haskell class variables into ConceptC++ concept parameters and associated types, the corresponding division of superclasses in the context of a type class, and the linearization of Haskell ASTs to the concrete syntax of ConceptC++. We also discuss cases in which there is no single correct translation from classes with functional dependencies to concepts. Our translation handles these cases in a reasonable way and is well-defined for the cases most common in practice. The translation is motivated by an ongoing project for distributed adaptive finite volume methods, in which software components are modeled in Haskell and implemented in C++.

1 Introduction

The goal of the S project at the Potsdam Institute for Climate Impact Research (Botta and Ionescu, 2007; Botta et al., 2006, 2008) is to provide reusable, generic software components for distributed adaptive finite volume methods. In the development process, Haskell is used as high-level modeling language and C++ as implementation language: Haskell allows reasoning about the soundness of constructs and algorithms, whereas C++ allows for an efficient implementation, which is crucial in numerical applications. The S project started in 2004, its software is released since early 2007. Up to now, however, the transition from Haskell to C++ was not formally specified.

The purpose of this paper is to define the mapping from generic Haskell specifications to C++ so, that it ultimately can be automated. The principal constituents of these generic specifications are abstractions of types and restrictions on possible parameter types to these abstractions. In Haskell, abstractions on *type variables* are represented as *type classes* that are parameterized by these variables (Wadler and Blott, 1989). *Superclasses* impose restrictions on the parameters. In *multi-parameter type classes*, relationships among the parameters are stated as *functional dependencies* (Jones, 2000). In ConceptC++ (Gregor and Stroustrup, 2007), on the other hand, which anticipates the upcoming revision of the C++ standard, abstractions are expressed by *concepts* (Gregor et al., 2006b; Dos Reis and Stroustrup, 2006), which take type parameters and require *associated types*. Restrictions on concept parameters and associated types are stated as *refinements* and as *requirements*.

In previous work, these language features have been compared with each other and with similar techniques for expressing *constrained genericity* in the same or the other language (Chakravarty et al., 2005a; Järvi et al., 2003; Järvi et al., 2003; Kothari and Sulzmann, 2005), and Haskell language elements and constructs have been emulated

in C++, for example in the FC++ library (McNamara and Smaragdakis, 2004b). In this paper, we go one step further by developing rules for translating Haskell specifications into the corresponding ConceptC++ code.

The presentation of this paper is based on a particular part of the S project software, namely the generic Relation-Based Algorithm (RBA) pattern. We first provide the necessary terminology of constructs in Haskell and ConceptC++, using the RBA as example (section 2). The translation rules themselves are discussed in section 3. In section 4, we outline future work and conclude.

2 Background and Terminology

The two corresponding features considered in this paper are Haskell *type classes* and ConceptC++ *concepts*. We introduce the necessary terminology by way of the RBA example. First we briefly describe the idea of RBAs, then we discuss a generic Haskell specification and the corresponding ConceptC++ code, which are based on type classes and concepts.

The Relation Based Algorithm pattern is a simple, yet powerful, computation pattern that has been developed within the S project and now plays a central role in the design of the software. In its original form, the RBA can be described by the following snippet of Haskell code (Botta and Ionescu, 2007):

```
rba :: [[Int]] -> [a] -> ([a] -> b) -> [Int] -> [b]
rba rss fs h js = [ h [ fs!!i | i <- rss!!j ] | j <- js ]
```

RBA takes as input a relation `rss`, a function `f`, a user-defined function `h`, and the points `js` at which to evaluate the relation `rss`. The details of the workings of an RBA and how it is used in the numerical software, are beyond the scope of this paper.

```
-- Generic RBA
class Functor f where
    fmap :: (a -> b) -> f a -> f b
class Set f => LambdaRelation r f a b | r -> f a b where
    lambda :: r -> a -> f b
class DiscreteFunction f a b | f -> a b where
    value :: f -> a -> b
rba rss fs h js = fmap f js
    where f j = h (fmap g (rss `lambda` j))
              where g i = fs `value` i

-- Making it work with lists
instance LambdaRelation [[a]] [] Int a where
    lambda rss x = rss !! x
instance DiscreteFunction [a] Int a where
    value fs i = fs !! i
```

Listing 5.1: Specification of generic RBA in Haskell

Presenting the RBA using list comprehension syntax has the advantage of simplicity, but also limits the RBA to specific types and hides some semantic nuances, including the precise requirements on the underlying container type. Most importantly for our purpose, however, it complicates the mapping from Haskell to ConceptC++ since

```
1  // concept example
2  concept DiscreteFunction<typename F>
3  : Function<F>  // refinement example
4  {
5    typename domain_t;
6    typename codomain_t;
7    requires CopyConstructible<codomain_t>;  // requirement example
8    codomain_t value(const F&, const domain_t&);
9  };
10
11 // concept_map example
12 template<typename T>
13 concept_map DiscreteFunction<std::vector<T>> {
14   typedef std::vector<T> function_t;
15   typedef typename function_t::size_t domain_t;
16   typedef typename function_t::value_t codomain_t;
17   codomain_t value(const function_t& f, const domain_t& d)
18         { return f[d]; }
19 };
20
21 // RBA algorithm
22 template<typename Rel, typename Fun, typename H, typename Js>
23 requires FunctConstr<Js>,
24          Functor<Js, FFun<H, Rel, GFun<Fun>,
                FunctConstr<Js>::value_t>>
25 Funct<Js, FFun<H, Rel, GFun<Fun>, FunctConstr<Js>::value_t>>::result_t
26 rba(Rel& r, Fun& f, H& h, Js& js) {
27   GFun<Fun> gh(f);
28   FFun<H, Rel, GFun<Fun>, FunctConstr<Js>::value_t> fh(h, r, gh);
29   return fmap(fh, js);
30 }
```

Listing 5.2: Translation of RBA type classes to ConceptC++

the details of the desired implementations differ considerably. As a first step towards the translation, we therefore lift the RBA to a generic version. The translation itself can then take place at a higher level of abstraction that captures the essence of the RBA and is not cluttered with inessential details of a problem. The generic RBA is shown in listing 5.1.

Lines 2–10 of the listing define the necessary type classes and the polymorphic function expressed in terms of these type classes. Each type class defines *class methods*, one per class in this case, which must have a unique name in the top-level namespace. This restriction is necessary because of Haskell's *type inference* mechanism that attempts to automatically deduce the type of each term. Because the name `value` is reserved for the `DiscreteFunction` class, the Haskell compiler can produce the corresponding constraint on line 10 without any explicit annotations from the user.

Each type class introduces at least one (only one in Haskell 98) type variable. The type variables may be used to specify class methods and must all be mentioned in the type of each class method. For example, the type of the `value` function on line 7 mentions all three variables `f`, `a`, and `b` of the `DiscreteFunction` class and, therefore, is legal. Classes can require a certain *context*. For example, the `LambdaRelation` class

requires that there must be an instance of the class `Set` (`Set` is not defined in listing 5.1) with variable `f`. In other words, `Set` is a *superclass* of `LambdaRelation`.

The classes `LambdaRelation` and `DiscreteFunction` specify *functional dependencies* (Jones, 2000): `r -> f a b` and `f -> a b`, respectively. A functional dependency of the form `lhs -> rhs` has two important implications. First, it states that all type variables in `rhs` are *reachable* given the variables in `lhs`; the notion of reachability extends the restriction on types of class methods having to mention all class variables—it is enough that all variables are reachable from that type. Second, a functional dependency restricts the possible *instances*. For example, once the instance on lines 15–16 is defined, no other instance is allowed with `[[a]]` as the first argument.

Type classes implement *ad-hoc* polymorphism (Cardelli and Wegner, 1985; Wadler and Blott, 1989), in which the implementation of an overloaded function must be specified for each set of possible types with which the function may get called. Each instance declaration specifies one case for one set of arguments, some of them possibly polymorphic themselves. For example, the instance declaration on lines 15–16 provides the definition of the overloaded function `value` from the `DiscreteFunction` class for any list `[a]` (where `a` is a type variable), the type `Int`, and the type `a` of the elements of the list. When the function `value` is called with the corresponding arguments, the Haskell compiler finds that instance and the specific definition of the `value` function.

listing 5.2 contains parts of (simplified) ConceptC++ code corresponding to the Haskell code presented in listing 5.1. *Concepts* in ConceptC++ correspond to type classes in Haskell. The `DiscreteFunction` concept, listed in lines 2–9, represents the same abstraction as the Haskell type class with the same name. A concept has *type parameters*, enclosed in angle brackets after the name of the concept, just as a Haskell class has type variables. In listing 5.2, the `DiscreteFunction` concept, for example, has one parameter `F`. The example is extended with a *refinement* clause: any `DiscreteFunction` must also be a `Function` and, therefore, provide all associated types and operations required by the `Function` concept.

Concepts also have *associated types* written as a `typename` declaration in the concept body. Associated types are determined by the parameters of a concept; for example, the `domain_type` and `codomain_type` associated types of the `DiscreteFunction` concept depend on the concept parameter `F`. Related to associated types are *inline requirements*, for example, the requirement that `codomain_t` be `CopyConstructible` (line 7). Inline requirements are similar to refined concepts but, in difference from refined concepts, may mention associated types (for further distinctions between refinements and requirements see Gottschling, 2006). Finally, *concept operations*, declared within a concept body, correspond to Haskell class methods; each operation has a signature specifying its argument and result types.

Lines 12–19 give an example of a *concept map*, more specifically, a *template concept map*. Without getting into details, the concept map states that `std::vector`, with any element type, models the `DiscreteFunction` concept and provides a definition of the associated types and operations required by the concept. The *constrained template* `rba` (lines 22–30) corresponds to the polymorphic Haskell function `rba`. The two important things to notice are that all requirements must be explicitly given in a `requires` clause of the template and that Haskell `where` expressions are represented by a ConceptC++ object. The need to specify requirements explicitly and to represent Haskell `where` expressions with objects stems from the fact that ConceptC++, unlike Haskell, does not perform type inference.

TypeClass:
    `class` [*SuperClass*+ `=>`] *TypeClassId TypeVarId*+ [`|` *FunDeps*]
Superclass:
    TypeClassId TypeExp+
TypeExp:
    TypeVarId | *TypeConstr* | *ConcreteType*
TypeConstr:
    TypeConstrId TypeExp+
FunDeps:
    Dependency+
Dependency:
    TypeVarId+ `->` *TypeVarId*+

Figure 5.1: Grammar of type classes without class methods

3 Translation Rules

In this section we describe the translation from Haskell multi-parameter type classes with non-constructor type variables and functional dependencies to concepts in ConceptC++. The grammar of the sublanguage of Haskell we consider is specified in figure 5.1. Throughout the translation process, we assume that the Haskell code given as input is valid in Haskell 98 with Glasgow extensions.

A class declaration, as specified by the grammar, consists of an optional context, the name of the class, the type variables of the class, and optional functional dependencies. The context consists of superclass requirements where each superclass can be thought of as a prerequisite for the currently declared class. A superclass takes an appropriate number of arguments where each of the arguments can be a type variable, an applied type constructor, or a concrete type. For example, for a type class with one variable, `a`, one can require that the type variable is equality-comparable and that it can be reduced to an `Int` by requiring the context `(Eq a, Reducible a Int)`. Haskell puts some additional semantic restrictions on the superclasses in a class context (GHC) but we do not enumerate them here.

In addition to superclass requirements, functional dependencies are a feature crucial to generic programming in Haskell. Each class can introduce a set of dependencies of the form `lhs -> rhs` where both `lhs` and `rhs` are sets of type variables. As discussed in section 2, functional dependencies allow one to tie types on the right hand side of a dependency to the types on the left hand side. For example, in listing 5.1, the functional dependency `f -> a b` is crucial to properly express the concept of a `DiscreteFunction` on line 15 since one function type `f` should always be tied to only one domain and codomain type, represented by the type variables `a` and `b`.

In the remainder of this section we describe the translation rules. The presentation is divided into two parts. First, in section 3.1, we discuss the four top-level translation steps, related to the four syntactic parts of a C++ concept. Then, in section 3.2, we describe the internals of the translation: the computations necessary to process functional dependencies and to split a Haskell class context into either refined concepts or inline requirements, and the linearization process.

$[\![$ `class` *superClasses* `=>` *C typeVars* `|` *funDeps* $]\!] =$

`concept` $\widetilde{C}$`<`*reduce*(*paramTypeVars*(*typeVars*,*funDeps*),*typeParamDecl*,‘,’)`>`
: *reduce*(*refinementClasses*(*paramTypeVars*(*typeVars*,*funDeps*),*superClasses*),
conceptInstance,
‘,’)
{
reduce(*assocTypeVars*(*funDeps*),*assocTypeDecl*,ε)
require *reduce*(*requirementClasses*(*assocTypeVars*(*funDeps*),*superClasses*),
conceptInstance,
‘,’);
}

Figure 5.2: Essence of translation rules

3.1 Top-Level Translation Steps

The top-level, logical view of the translation from Haskell type classes to ConceptC++ concepts is given in figure 5.2 as the composition of translation functions; the functions themselves are defined in appendix 9. The concrete syntax is typeset in a `typewriter` font; all expressions typeset in *italics* evaluate to a set of abstract syntax trees, and all *underlined* identifiers name a linearization function that produces concrete ConceptC++ syntax. The tilde “$\widetilde{\ }$” signifies identifier translation. The input to the translation is an abstract syntax tree of the grammar in figure 5.1.

The translation can be divided into four steps, marked by four *reduce* linearizations in figure 5.2, each of which is related to one part of a concept:

1. the parameters of the concept,
2. the list of refined concepts,
3. the associated types of the concept,
4. the requirements on concept parameters and associated types.

Given a Haskell type class with multiple parameters, some of the class variables will be translated into concept type parameters and some into associated types of the concept; the functional dependencies of a class control the division. In general, every class type variable could be translated into a concept parameter. However, that translation would make generic software less practical because quite a large number of parameters would have to be passed around (Garcia et al., 2003). In ConceptC++, and even earlier in plain C++, associated types are a natural and widely used way of decreasing the number of parameters. Whenever possible, we therefore target associated types (steps 1 and 3); the next section discusses the details. An extension of Haskell with a feature corresponding to associated types in ConceptC++ has been proposed (Chakravarty et al., 2005a) but is neither widely used yet nor included in Glasgow extensions to Haskell 98. If it were, it could directly guide the translation.

The division of superclasses in a class context into either refined concepts or inline requirements (steps 2 and 4) depends on the corresponding division of class variables into concept parameters and associated types (steps 1 and 3). Each superclass must mention at least one type variable in its type expression. Those superclasses that depend

on variables translated into associated types obviously cannot be translated into refined concepts.

3.2 Internal translation steps

The four high-level translation steps from the previous section section 3.1 depend on three kinds of internal computations: computations for processing functional dependencies, computations for splitting a Haskell class context into refined concepts and inline requirements, and the linearization process. We further elaborate on each of the three computations.

In most practical cases, a class specifies only one functional dependency `lhs -> rhs` with all class variables mentioned either in `rhs` or `lhs`, see the classes in listing 5.1 for an example. In this case, the variables on the right hand side of the dependency can safely be translated into associated types of a concept, the variables on the left hand side can be translated into concept parameters, and the functional dependency is upheld: only one concept map with particular arguments for the left hand side variables is possible and thus only one corresponding assignment of types to variables on the right hand side can be made. In the cases when more than one functional dependency is present or not all class variables are mentioned in the functional dependencies, however, it is not clear which class variables should become associated types and, generally, not possible to translate that type class to a single concept in a semantics-preserving way. Any exact solution would have to ensure that only one assignment of the types on the right hand side of every dependency in a class is possible for a corresponding assignment of types to variables on the left hand side, but such solution is not possible with the translation of a class into a single concept where class variables are divided only into the two sets of concept parameters and associated types. Instead of providing an exact solution, we choose an approximation of translating all variables that occur only on the right hand sides of the functional dependencies into the corresponding associated types. That translation does not preserve the semantic condition of functional dependencies but it safely chooses those variables that can always be translated into associated types. Associated type synonyms, recently proposed for Haskell (Chakravarty et al., 2005a), share with associated types in ConceptC++ the problem that they cannot be used to correctly represent functional dependencies.

Once the division of type variables into concept parameters and associated types of a concept is complete, the superclasses in the class context must be divided in the corresponding way. Because of the scoping rules in ConceptC++, every type class that mentions at least one type variable that has become an associated type must be translated into an inline requirement—not all concept parameters corresponding to the variables mentioned in the superclass are in the scope of the refine clause. To decide whether a superclass will become a ConceptC++ requirement, we compute the set of all type variables mentioned in the type expressions of the superclass (see the grammar in figure 5.1) and check if any type variable that will be translated into an associated type is an element of that set. The classes that are then translated into refined concepts are simply the difference between all superclasses and the ones translated into inline requirements.

The final kind of internal helper functions are the linearization functions. These functions comprise the *reduce* function and the specific functions corresponding to the syntactic parts of a concept. The reduce function takes a set of abstract syntax trees (ASTs) of the grammar in figure 5.1, a particular linearization function, and a delimiter that can be an empty string. The particular linearization function is applied

to every AST and the results are concatenated to a single string in an undefined order using the delimiter as separator. All linearizations are rather straightforward except the *conceptInstance* function. The type expressions of a superclass in Haskell may involve a mix of type variables and particular types. For example, a superclass `Collection c [a]` may be used to state that `c` must be a collection of lists of `a`s. To translate this superclass into a refined concept or an inline requirement, one needs to first decide how to translate `[a]`. In general, there is no "correct" translation between particular types of the two languages—one could, for example, target in ConceptC++ both `Collection<c, std::list<a>>` and `Collection<c, std::vector<a>>` to translate the Haskell class `Collection c [a]`. We therefore do not specify how to translate particular types and rather leave the particular mapping up to the application. An exception is the Haskell unit type `()`, a particularly important concrete type that closely corresponds to the C++ `void` type, thus can always be mapped to it.

4 Conclusions and Future Work

We have considered a practical example of multi-paradigm library development, namely the S project. In the S project, Haskell is used for modeling, specifications, and test implementations, while C++ is used for the actual implementation. Currently, the C++ code is not directly derived from specifications in Haskell and must be kept in sync with the specification manually. We have described a mapping between Haskell and C++ that is based on two kinds of abstractions, Haskell type classes on the one hand and C++ extended with concepts (ConceptC++) on the other hand.

The important parts of the mapping between type classes and concepts are the division of Haskell class variables into ConceptC++ concept parameters and concept associated types, the corresponding division of superclasses in the context of a type class, and the linearization of Haskell ASTs to the concrete syntax of ConceptC++. Our translation correctly handles the cases most common in practice when there is one functional dependency and the dependency mentions all class variables. In the other cases, our translation chooses an approximate solution that does not completely preserve the semantics of dependencies but results in a solution that will fit many practical cases. Our translation also correctly handles superclass declarations. When they involve a particular Haskell type, however, our translation does not specify how such type should be translated. Since there is no "correct" mapping between particular types of Haskell and ConceptC++, it is best to leave that translation to the developers, who can decide on a case-by-case basis. An exception is the Haskell unit type `()`, which can be translated into the `void` type in ConceptC++.

Ultimately, our goal is to translate Haskell generic algorithms to constrained templates in ConceptC++. As the immediate next step towards that goal, we plan to tackle the translation of Haskell class methods into ConceptC++. Since class methods can introduce new type variables and have their own context, their translation is not trivial. Interesting questions here center around the type computations that might be required to determine the result type of an operation. Briefly, in ConceptC++, type variables in class methods must become concept type parameters, so that clients of a concept can provide the necessary type computation. We also need to extend our translation by constructor classes. Such extension requires the representation of higher-kinded types and, again, type computations in ConceptC++, which both are non-trivial tasks. A possible solution could be to introduce concepts that represent applied type constructors; such concepts would allow for 'pattern matching' on an applied constructor to obtain

its arguments.

Acknowledgments

The authors acknowledge the Swedish Foundation for International Cooperation in Research and Higher Education (STINT) and the Deutsche Akademische Austauschdienst (DAAD) for their partial support of that work. We also thank Gustav Munkby for discussions and Doug Gregor for answering questions on ConceptC++.

Appendix A—Complete Translation Rules

$paramTypeVars(typeVars, funDeps) =$
$\quad typeVars - assocTypeVars(funDeps)$
$assocTypeVars(funDeps) =$
$\quad rhs(funDeps) - lhs(funDeps)$
$rhs(funDeps) =$
$\quad join(map(pickRhs, funDeps))$
$lhs(funDeps) =$ as $rhs\ldots$
$pickLeft([leftTypeVars \texttt{ -> } rightTypeVars]) =$
$\quad leftTypeVars$
$pickRight([leftTypeVars \texttt{ -> } rightTypeVars]) =$ as $pickLeft\ldots$
$join(aSetOfSets) =$ as monadic *join*, removes one level of set structure...
$map(fun, aSet) =$
$\quad \{fun(el) \mid el \in aSet\}$
$refinementClasses(assocTypeVars, superClasses) =$
$\quad superClasses - requirementClasses(assocTypeVars, superClasses)$
$requirementClasses(assocTypeVars, superClasses) =$
$\quad = join(\{mentionedInClasses(v, superClasses) \mid v \in assocTypeVars\})$
$mentionedInClasses(typeVar, superClasses) =$
$\quad \{c \mid c \in superClasses, typeVar \in mentionedIn(typeExpsOfClass(c))\}$
$typeExpsOfClass([typeClassId\ typeExps]) =$
$\quad typeExps$
$mentionedInExps(typeExps) =$
$\quad join(map(mentionedInExp, typeExps))$
$mentionedInExp([typeVarId]) =$
$\quad \{typeVarId\}$
$mentionedInExp([concreteType]) =$
$\quad \emptyset$
$mentioendInExp([typeConstr]) =$
$\quad mentionedInTypeConstr(typeConstr)$
$mentionedInTypeConstr([typeConstrId\ typeExps]) =$
$\quad mentionedInExps(typeExps)$

Figure 5.3: Translation functions

$\underline{reduce}(ASTs, \underline{linearization},$ delimiter $) =$
  $\underline{linearization}(AST_1) +\!\!+$ delimiter $+\!\!+ \cdots +\!\!+$ delimiter $+\!\!+ \underline{linearization}(AST_n)$
  where $n = |ASTs|$
$\underline{typeParamDecl}([typeVarId]) =$
  $\widetilde{typeVarId}$
$\underline{conceptInstance}([typeClassId\ typeExps]) =$
  $\widetilde{typeClassId} +\!\!+$ < $+\!\!+ \underline{conceptArguments}(typeExps) +\!\!+$ >
$\underline{conceptArguments}(typeExps) =$
  $\underline{reduce}(typeExps, \underline{conceptArgument},$ ',' $)$
$\underline{conceptArgument}([concreteType]) =$
  if *concreteType* is unit type `()` then `void` else undefined
$\underline{conceptArgument}([typeConstrId\ typeExps]) =$
  $\underline{reduce}(typeExps, \underline{conceptArgument},$ ',' $)$
$\underline{conceptArgument}([typeVarId]) =$
  $\widetilde{typeVarId}$
$\underline{assocTypeDecl}(typeVarId) =$
  `typename` $\widetilde{typeVarId}$;

Figure 5.4: Linearization functions

Paper V

Scrap++: Scrap Your Boilerplate in C++

Scrap++: Scrap Your Boilerplate in C++

Gustav Munkby, Andreas Priesnitz, Sibylle Schupp and , Marcin Zalewski

Abstract

"Scrap Your Boilerplate" (SYB) is a well studied and widely used design pattern for generic traversal in the Haskell language, but almost unknown to generic programmers in C++. This paper shows that SYB can be implemented in C++. It identifies the features and idioms of C++ that correspond to the Haskell constructs that implement SYB, or can be used to emulate them, and provides a prototype C++ implementation.

1 Introduction

"Scrap your boilerplate" (SYB), introduced by Lämmel and Peyton Jones (2003), is a technique for generic queries and traversal that was introduced in Haskell and further refined there (Hinze and Löh, 2006; Hinze et al., 2006; Lämmel and Peyton Jones, 2003; Lämmel and Peyton Jones, 2004, 2005); for many Haskell programmers, the SYB library Data.Generics has become a standard tool. Despite the apparent usefulness of the SYB library in Haskell, no comparable library is available to the C++ community. Since the current incarnation of SYB in Haskell depends on type classes, higher-order functions, pattern matching, and other features that are not available in C++, we wondered how, if at all, the counterpart of SYB in C++ looks. In this paper, we present and discuss our implementation of the original SYB version (Lämmel and Peyton Jones, 2003) in generic C++.

Generic programming in C++ depends on templates, which, unlike polymorphic functions in Haskell, cannot be type-checked when defined, but only when instantiated. Given the differences between the two languages, our C++ solution corresponds surprisingly closely to its Haskell counterpart: each ingredient of the SYB approach can be mapped to C++. However, C++ introduces complications not present in Haskell. For one, certain type checks in Haskell require explicit type computations in C++, which have to be provided as template metaprograms. Although parts of these computations can be automated in libraries (Boost; de Guzman et al., 2008; McNamara and Smaragdakis, 2004a; Priesnitz and Schupp, 2006), type checking becomes more inconvenient than in Haskell. Another important aspect in C++ concerns the difference between in-place and out-of-place computations that are sometimes difficult to unify in the same implementation—not an issue in Haskell, a pure, functional language. On the other hand, some parts of SYB are easier to implement in C++; the best example is the type extension of functions discussed in section 3.3. In Haskell, at least one language extension is necessary to support SYB; as we will see, in C++, no extensions are needed. In this paper we stay within the generic programming paradigm as understood in C++, thus do not consider object-oriented design alternatives.

The rest of the paper is organized in a conceptually top-down manner. In section 4 we briefly introduce the idea behind SYB using the standard "paradise" example from the original SYB paper (Lämmel and Peyton Jones, 2003). In section 3 we discuss the design issues and present an outline of the C++ implementation: recursive traversal combinators in section 3.1, one-layer traversal in section 3.2, and type extension in section 3.3. In section 4 we detail what has to be done for each type to make it SYB compatible and in section 5 we present one possible way of automating the task. We

reflect on the scope of SYB in the context of class-based languages in section 6 and discuss the remaining features of SYB not present in our current implementation along with possible generalizations in section 7. In section 8 we summarize related work. We conclude in section 9 with a summary and an outlook on future work.

All Haskell code is taken from the first SYB paper (Lämmel and Peyton Jones, 2003). For brevity, C++ code is provided in a simplified form omitting irrelevant technical details, for example, the `inline` specifier. While the Haskell code has been in use and tested for some time, the C++ code is for exposition; our current prototype does not claim to provide an industrial-strength solution.

2 The Paradise Example

SYB problems deal with applying local computations to tree-like structures. In the original SYB paper, the motivating "paradise" example is about increasing the salaries of all employees in a company. The type representing a company reflects a tree-like company structure: each company has subunits, each subunit has employees and managers, each employee is a person that has a salary, and so on; the local computation in that example is the increase of a salary.

The straight-forward implementation of the salary increase involves just a single function, `incS`, containing the type-specific computation for increasing a salary. To apply this function to a company, however, a lot of "boilerplate" code is required just to extract the relevant bits, namely the salaries within the company. In the SYB literature, the type-specific computation `incS` often is informally referred to as "the interesting" computation.

The SYB solution defines *generic traversal combinators* that, combined with `incS`, form a type-generic function `increase`. Applying `increase` to a company will traverse its tree-like structure and apply `incS` to all salaries:

1 ◇ Haskell

```
increase :: Float -> Company -> Company
increase k = everywhere (mkT (incS k))
```

2 ◇ C++

```
Company& increase(float k, Company& company) {
    return everywhere(mkT(incS(k)), company);
}
```

In the body of the `increase` function, there are two combinators at work. Firstly, the *recursive traversal combinator* `everywhere`, which traverses a structure and recursively applies a transformation to all parts. Secondly, the *generic transformation combinator* `mkT`, which makes a type-specific transformation type-generic, by applying the identity transformation to other types.

In the following section, we elaborate on each of the constituents of the SYB approach and compare the Haskell and C++ implementations.

3 SYB in C++

The SYB approach to generic traversal consists of three major ingredients: recursive traversal combinators, a one-layer traversal, and a type extension of the "interesting" functions. In this section we discuss the issues to be considered when implementing SYB in C++. The discussion of each part of SYB is accompanied by snippets of code in Haskell and C++. As said before, the C++ code presents an illustration of the discussion rather than a final solution; the complete listing can be found in the appendix.

3.1 Recursive Traversal

Conceptually, the most characteristic functions in the SYB approach are generic traversal strategies; in the paradise example in section 4, the strategy `everywhere` was used. Traversal strategies combine one-layer traversal with an "interesting" function to obtain a task-specific recursive traversal. The strategy `everywhere`, for example, takes an arbitrary transformation and a value, and applies this transformation to all nodes of the tree-like structure of the value. In Haskell, `everywhere` simply is a combinator that constructs a recursive traversal.

To implement `everywhere` in C++, we need to decide how to represent the recursive traversal. Recursive traversal requires passing as argument the composition of `everywhere` and the transformation to be applied. Like combinators in general, it implies the existence of higher-order functions, which have no direct support in C++. The common approximation to higher-order functions in C++ are *function objects*, which are classes that define the so-called function-call operator (a.k.a. parentheses operator). Instances of a function object can be passed to functions much as any other object, but behave like functions when their function-call operator is called. Moreover, function objects permit function composition by taking another function object as constructor argument and then defining the body of their function-call operator appropriately.

Since we can assume that we know at instantiation time with which transformation `everywhere` is composed, we represent `everywhere` in C++ as a function-object template, more precisely, a function object with a static type parameter and a corresponding dynamic constructor parameter; we call this function object `Everywhere`. At instantiation time its static parameter is bound to the type of the particular transformation function object. When an instance of `everywhere` is created, the actual function object is passed as an argument to the constructor.

Because C++ has no type signatures for template parameters, there is no way to directly specify that the transformation function passed to `Everywhere` must be polymorphic and to type-check function arguments against a type signature. Instead, a C++ compiler type-checks templates when they are instantiated. In the general case, the transformation passed to `Everywhere` will be applied to terms of different types. Therefore, if the transformation is not polymorphic and cannot be applied to terms of different types, the type check fails—in effect, `Everywhere` encodes a rank-2 polymorphic function without having a rank-2 type.

The code for the `everywhere` combinator in each of the two languages follows. In both examples, much of the work is forwarded to the function `gmapT`. This function performs a specific kind of one-layer traversal, namely the transformation of a term; `gmapT` will be defined in section 3.2.2. In section 4 we have shown how to use `everywhere` in Haskell and in C++, in the appendix we list a complete main program.

```haskell
everywhere :: Data a
           => (forall b. Data b => b -> b)
           -> a -> a
everywhere f x = f (gmapT (everywhere f) x)
```

#4 ◇ C++

```cpp
template <typename Fun_>
struct Everywhere
{
    Fun_ fun;
    Everywhere(Fun_ fun) : fun(fun) {}

    template <typename Param_>
    Param_& operator()(Param_& param)
    {
        return fun(gmapT(*this,param));
    }
};

template <typename Fun_, typename Param_>
Param_& everywhere(Fun_ fun, Param_& param) {
    return (Everywhere<Fun_>(fun))(param);
}
```

As the definition of the class `Everywhere` shows, the Haskell combinator `everywhere` is realized in C++ as a small class template, consisting of a constructor and one member function, the function-call operator. The class stores the transformation function in a member variable. When its function-call operator is called, it first applies itself to all subterms of the current term using the `gmapT` one-layer traversal function and finally applies the stored transformation to the current term, effectively encoding bottom-up traversal. Instances of `Everywhere` correspond to the composition `(everywhere f)` in Haskell. Thus, passing of the `*this` object to `gmapT` corresponds to the recursive call.

Following a standard idiom, we accompany the class template `Everywhere` by a function template `everywhere`, which creates an anonymous instance of the class, calls its function-call operator with the argument `param`, and returns the result. The function is provided for mere convenience and exploits the fact that C++ performs type inference on arguments to function templates, thus saves users the declaration of variables of class-template type. Finally, we note that the curried Haskell function of two arguments has become a binary function in C++. There are several ways of emulating currying in C++. Our implementation of `everywhere` stores the transformation function in a member variable. A similar effect is also achievable by using, for example, a `bind` construct like that from the Boost libraries (Dimov, 2008).

3.2 One-Layer Traversal

The definition of the generic traversal strategy relies on the function `gmapT` to perform one-layer traversal. Other combinators of the SYB approach similarly rely on the functions `gmapQ` (Q for queries) and `gmapM` (M for monadic transformation). In the SYB

approach, these mappings are abstracted to a generic fold function, `gfoldl`, which, conversely, allows defining `gmapT` and other combinators by applying `gfoldl` appropriately. We first discuss the definition of `gfoldl` and our corresponding implementation in C++, then one of its possible applications, `gmapT`.

3.2.1 The Generic Fold

The generic fold in SYB folds a term of some type into a term of a possibly different type: it starts by folding the empty constructor application and then, step by step, combines the folded application with the remaining subterms. For example, `gmapT`, aside from technical details, preserves the original constructor and applies it to the transformed subterms producing a new term of the same type as the original one. Unlike the usual fold, the subterms that `gfoldl` folds may be of different types; the first subterm is a special case of an empty constructor application. Since `gfoldl` has to be general enough to allow folding of a term to a new term of a possibly different type, its type is quite hard to digest even for Haskell professionals. Intuitively speaking, `gfoldl` takes two polymorphic functions and a term, where its first argument (polymorphically) controls the folding of an empty construction application and the second argument (polymorphically) controls the subsequent combination of the folded "tail" with the next subterm. Because both functions must be polymorphic, the type of `gfoldl` is rank-2, thus relies on an extension to Haskell 98, which, however, is now well-established. Its full type is given in ex. # 5 below.

Conceptually, the implementation of `gfoldl` in C++ is very similar to the one in Haskell, yet, we will refer to it as `gfold` from now on. The difference in name signifies that in C++ the order of subterms is generally undefined while in Haskell the order is imposed by the constructor application on which `gfoldl` (left fold) operates. The first argument of the C++ `gfold` corresponds closely to its Haskell counterpart. For the second argument one has to decide how to pass the information that is contained in a Haskell type-constructor argument. We decided to represent this second argument of `gfold` as a polymorphic function that takes the type of the current term and a reference to its value as static and dynamic arguments. This function allows one to obtain a type-specific initial value.

Before listing the `gfold` definition, we point out an important difference between C++ and Haskell. In Haskell, the dependency between the argument types and the return type of a function is captured in the function's type signature, while in C++ that dependency has to be provided by an explicit type computation—there are no type signatures for template arguments. Thus, to achieve the same effect of a fully polymorphic function one needs to accompany the run-time computation that the function performs by the corresponding compile-time computation for the return type. This was not necessary in the definition of the `Everywhere` combinator because the return type was constant, but is required in case of `gfold` where the return type depends not only on the type of the term, but also on the types of all its subterms and the results of applying the supplied functions.

We now list the type of the Haskell `gfoldl`, then (in pseudo-code) the C++ counterpart of the `gfoldl` function, the `gfold` function template. The Haskell listing includes the type signature and a specific definition of `gfoldl` for an `Employee` type (Lämmel and Peyton Jones, 2003), while the C++ listing shows only a specific case since, as pointed out, a type signature cannot be provided.

#5 ◇ Haskell

```
class Typeable a => Data a where
gfoldl ::(forall a b . Data a => w (a -> b)
                             -> a -> w b)
       -> (forall g. g -> w g)
       -> a -> w a
data Employee = E Person Salary
instance Data Employee where
  gfoldl k z (E p s) = (z E `k` p) `k` s
```

#6 ◇ C++

```
template <typename Fun_, typename InitFun_>
typename Result<...>::Type
gfold(Fun_ k, InitFun_ z, Employee& e) {
    return k(k(z(e), e.p), e.s);
}
```

In the definition of `gfold`, the parameter of type `InitFun_` is a function object that creates a type-specific initial value for a particular term; the other two parameters correspond to their Haskell counterparts. The return-type computation mirrors the body of `gfold` but operates on the corresponding types instead of the values (the technical details are not crucial, but see the appendix for an illustrating definition). The return type of `gfold` for `Employee` depends on the type transformations performed by `k` and `z`, on the `Employee` type itself, and on the types of `e.p` and `e.s`.

To use `gfold` properly, it is important to understand that for every function passed as either `Fun_` or `InitFun_` instance, a type-level computation has to be provided, corresponding to its run-time computation. When using `gfold` directly, these necessary type computations could require more than average C++ expertise. Typically, however, users employ specific instances of `gfold`, including the specific maps discussed in section 3.2.2. In these, common cases, the user of an SYB library will not even have to know that such computations occur.

Just as `gfoldl`, the `gfold` template has to be defined for every type that supports SYB. In the C++ listing, ad-hoc overloading on the third parameter correlates to the Haskell instance declaration; the definition for the `Employee` type is discussed in detail in section 4. In addition to function overloading for specific types, the metaprogramming facilities of C++ allow one to specialize `gfold` for families of types specified by some static predicate; standard techniques include partial specialization or the SFINAE ("substitution failure is not an error," see Järvi et al., 2003) idiom.

3.2.2 Specific Maps

The generic fold function suffices to express all generic traversals but it is quite complex and not easy to work with. Most commonly performed tasks can be provided in simpler, specialized map functions that are defined in terms of the general fold function. For illustration, we discuss the specialization of the function `gmapT`, used in the definition of the `everywhere` combinator in section 3.1. Essentially, `gmapT` describes a one-layer traversal for transformations, that is, type-preserving mappings.

In Haskell, specializing the generic fold to `gmapT` requires binding the three parameters of `gfoldl` appropriately: its second parameter must be bound in a way so that

its application preserves the original type (as required for a transformation). Its first parameter, much trickier, must be bound to a binary function that applies a transformation f to its second argument and applies its first argument to the result. The third and unproblematic parameter, finally, is the tree itself. For the Haskell implementation, it follows that the first parameter is bound to the folding function just described and the second parameter to the identity function; a dummy type constructor `ID` is necessary to avoid the need for a type-level identity function, a feature not supported by Haskell.

Our C++ implementation differs from the Haskell implementation in a crucial aspect. In C++, transformation in-place is a more natural paradigm. To perform an in-place transformation, the folding function passed to `gfold` as the first argument has to simply apply the transformation to each subterm. By expressing an in-place transformation, `gmapT` in C++ becomes simpler than in Haskell: our `gmapT` definition creates a function object that applies the transformation function f to the second argument that represents the current subterm—as in Haskell—but ignores the first argument that represents the folded "tail" of constructor application in Haskell, and just returns that argument. Conceptually, the return type has no meaning in the case of an in-place transformation, but it is necessary because of the `gfold` definition, which applies its folding function to the already folded part of the current term and the next subterm. Similarly, the initialization function does nothing but to return the current term. The Haskell implementation and a corresponding C++ in-place implementation follow:

#7 ◇ Haskell

```
newtype ID x = ID x
unID :: ID a -> a
unID (ID x) = x
gmapT f x = unID (gfoldl k ID x)
  where
    k (ID c) x = ID (c (f x))
```

#8 ◇ C++

```
template <typename Fun_, typename Param_>
Param_& gmapT(Fun_ fun, Param_& param) {
    return gfold(ApplyToSecond<Fun_,Param_&>(fun),
                 identity<Param_&>(),
                 param);
}
```

The C++ definition uses a reference to the `gmapT`'s parameter so that the transformation is performed in-place and the transformed term is returned. The `ApplyToSecond` function object is a simple wrapper that takes a unary function object and adapts it to a binary function object. The function call operator applies the unary function object to the second parameter and returns the first parameter, ignoring its value. The `Bind` function object holds a reference to the object to which `gmapT` is applied. When called, its argument is ignored and that reference is returned; for the details of the implementation, we refer to the appendix. While the in-place definition does not rely on returning a value, an out-of-place computation requires a return value to abstract gfold from the actual choice at this variation point.

Support for out-of-place transformation is possible as well but is certainly more difficult to accomplish than in-place transformation: while only one way exists to per-

form in-place transformations, there are different ways to realize out-of-place transformations. Since C++ does not guarantee immutable values, however, any efficient implementation needs to avoid copying of potentially large subtrees. Yet, it is conceivable that one implementation of `gmapT` can cover both in-place and out-of-place transformation by entirely encapsulating their differences in the appropriately specialized `Fun_` parameter. For simplicity, we refrain in this paper from such generalization.

3.3 Type Extension

We have discussed most of the machinery necessary to scrap the boilerplate. The last, crucial step that remains, is to generalize the "interesting" functions so that they can be applied throughout a tree and perform an action on the interesting bits while ignoring the others; in section 4, the `mkT` function takes such "interesting" function and turns it into a generic transformer. To generalize a non-polymorphic function, a test is necessary to decide if the function can or cannot be applied to the current term.

In the original Haskell SYB approach, type representations have to be compared at run time. If the two types are the same, `mkT` returns the "interesting" function and a polymorphic identity function otherwise. Because the comparison happens at run time, an explicit run-time type representation is necessary, complemented by an `unsafeCoerce` operation to perform nominal type cast, that is, a cast without change of representation. A later implementation of type-specific behavior for SYB is performed statically, but relies on introducing appropriate type classes (Lämmel and Peyton Jones, 2005).

In C++, where function overloading is statically resolved, the comparable effect of the type-specific cases can be achieved at compile time. By introducing two overloaded functions, which encapsulate the two cases of the type cast—the "interesting" function for one special type and the identity function for all others—and dispatching on the type of the "interesting" function, overload resolution performs the type test automatically and determines at compile time which of the two overloaded functions, thus which of the two type cases, to execute. The implementation of `mkT` in Haskell and C++ follows:

#9 ◇ Haskell

```
mkT :: (Typeable a, Typeable b)
    => (b -> b) -> a -> a
mkT f = case cast f of
          Just g -> g
          Nothing -> id
```

#10 ◇ C++

```
template <typename Fun_>
class MkT
{
    Fun_ fun_;
public:
    MkT(Fun_ fun) : fun_(fun) {}

    template <typename Param_>
    Param_& operator()(Param_& param)
```

```
    {
        return param;
    }
    typename Param1<Fun_>::Type
    operator()(typename Param1<Fun_>::Type param)
    {
        return fun_(param);
    }
};

template <typename Fun_>
MkT<Fun_> mkT(Fun_ fun) {
    return MkT<Fun_>(fun);
}
```

The C++ implementation is a parameterized function object, similarly to the implementation of the `Everywhere` combinator in section 3.1. The class `MkT` is a function object initialized with the "interesting" function. The first overloaded function-call operator implements a generic identity, the second operator implements the type-specific application. No overload resolution conflicts can occur since the non-generic operator, if applicable, is always chosen over the generic, template operator by the rules of C++ overload resolution. The type of the first argument of the type-specific operator is extracted from the "interesting" function using a traits template. Trait templates (Myers, 1995), another common idiom, can be compared to a compile-time database that stores type information. The free `mkT` function, finally, is provided as a convenience function for the same reason the `everywhere` function in section 3 accompanied the function object `Everywhere`.

In C++, function overloading is a standard technique to provide generic functions that are specialized for certain types. While a library-provided `mkT` function is useful, it would not be unusual in C++ for the user to specify the type cases by hand, for example, by providing a function that is generalized for families of types rather than specific ones.

4 Defining One-Layer Traversal

Almost all combinators of the SYB pattern are defined type-independently and can therefore be provided by a library. The only combinator that cannot be defined once and for all, is the `gfold` function. As discussed in section 3, `gfold` performs a type-specific traversal. Thus, its definition depends on the particular type of the argument. In the same section, we have outlined how `gfold` can be defined in C++. We now turn to the details of its definition.

Since the behavior of `gfold` is type-specific, it would be natural to provide it as a member function. Yet, we represent `gfold` as a non-member function, which can be overloaded to accept arguments of non-class type, like primitive and pointer types. Defining `gfold` as a free function requires access to possibly private member data of a class, which is granted only if the free function is declared as a `friend` of the class. Thus, `gfold` is defined as a non-member friend function of the class type traversed. As an alternative to friend functions, ex. # 6 could have been overloaded by a purely

global function for arguments of class type, which then invoked a corresponding member function that has data member access. Compared to our solution that is based on friends, however, code complexity would increase. ex. # 6 illustrated the definition of `gfold` for the `Employee` class. Its corresponding friend declaration is provided below.

11 ◇ C++

```
class Employee
{
    Person p;
    Salary s;
public:
    friend
    template <typename Fun_, typename InitFun_>
    typename Result<...>::Type
    gfold(Fun_ k, InitFun_ z, Employee& e);
};
```

Friend function declarations must be provided at the time of class definition. Defining the `gfold` function as above, thus, will not work for legacy code—even where the source code is available, so that one can look up all data members of a class, adding a friend declaration to the public interface of a class breaks encapsulation. Unless one is willing to modify legacy code, the recursive tree traversal, thus, ends when an instance of a legacy type is encountered, as if the legacy type was a primitive type. We will take up the questions of legacy code and data member access again in section 6.

Legacy code can be dealt with differently in `Haskell`, as `deriving` clauses may be provided separately from existing type definitions. Unless relying on compiler extensions, though, a developer still has to provide these clauses manually. One might say that one extra function per type is not a prohibitive burden, yet it is a bothersome duty we would like to avoid. Moreover, in C++ one `gfold` definition does not suffice; function parameters can be optionally qualified as `const`, `volatile`, or `const volatile`, and these qualifiers are part of the type check. Any realistic implementation of `gfold` would therefore have to support not just one, but all combinations of the qualifiers for each of its parameters, to preserve all qualification that occur in a generic context. The `gfold` function with its 3 parameters requires $2^3 = 8$ different parameter lists, thus 8 different function declarations—ironically with identical body. One might therefore wonder whether it is possible to avoid the need for writing this kind of boilerplate code. The next section will show how to generate `gfold` automatically.

5 Generating Generic Folds

The one-layer maps, as we have seen, must be supplied at least once for every data type. Although the implementation of such functions is relatively straightforward, writing this boilerplate code is still tedious, and automatically generating `gfold` is clearly desirable. A prerequisite for the automatic generation of the `gfold` functions is that the structure of data types is available. One possible approach is to let the compiler generate the `gfold` implementations since it already has the information about the structure of the data types. In Haskell, the SYB pattern was considered important enough to extend the compiler to automatically generate `gfold` for any type. In C++, it is not realistic to assume that all compilers would support a particular extension. However, one

can exploit the metaprogramming facilities of the language to generate `gfold` within an ordinary user program.

Conceptually, our approach relies on representing a user-defined type by a *heterogeneous container*, that is, a collection of elements of (possibly) different types. The idea behind this approach is to treat every user-defined type as a compile-time constructed tuple so that this type can be generated in an inductive manner through metaprogramming. In our C++ solution, we represent this inductive construction by the two classes `Insert` and `Empty`; in list-constructor terminology, `Insert` corresponds to `Cons`, `Empty` to `Nil`. Using `Insert` and `Empty`, the `Employee` type can be defined by successively inserting its two members of type `Person` and `Salary`, as demonstrated in the following snippet; conversely, such construction discipline and the use of `Insert` is required for any type for which `gfold` should be automatically generated. ex. # 12 illustrates the idea; a complete example is listed in the appendix:

12 ◇ C++

```
typedef Insert<Person,
               Insert<Salary,Empty> > Employee;
```

Because the heterogeneous container is defined in an inductive manner, `gfold` is the heterogeneous equivalent of a `fold` over a homogeneous container. The implementation of `gfold` can be expressed in terms of two straightforward cases, one for `Insert` and one for `Empty`, following the standard method for folding over a container.

The `Insert` class itself is the metaprogram that makes the structure of a data type available to `gfold`. Its definition relies on *parameterized inheritance*, that is, its own base class is one of the template parameters. This idiom has been widely applied to generating data structures (Czarnecki and Eisenecker, 2000; Eisenecker et al., 2000), generic containers representing tuples (Järvi and Powell, 2008), and maps of objects of different types (de Guzman et al., 2008; Winch, 2001). In our example, the second parameter to `Insert` is used as the base, and the heterogeneous container is built in terms of an inheritance chain (see the appendix for the full class definition). The base class of `Insert` can therefore be thought of as the tail of the list.

13 ◇ C++

```
template <typename Head_, class Tail_>
class Insert : public Tail_
{
    Head_ head_;
protected:
    template <typename Fun_, typename Init_>
    typename Result<...>::Type
    gfold_(Fun_ fun, Init_& init)
    {
        return Tail_::gfold_(fun,fun(init,head_));
    }
public:
    template <typename Fun_, typename InitFun_>
    friend
    typename Result<...>::Type
    gfold(Fun_ fun,InitFun_ initFun,Insert& param)
    {
```

```
        return param.gfold_(fun,initFun(param));
    }
};
```

The implementation of `gfold` for such types is split up into two functions. The `gfold` function, a free function defined as a friend, is merely a front-end that creates the respective initial value and passes it to another function, `gfold_`, which performs the fold over data members of the type represented by the given `Insert` instance. The implementation of `gfold_` corresponds exactly to a left fold over a homogeneous list. The call to `Tail_::gfold_` combines the current subterm (represented by the `head_` member) with the initial value and folds the result with the remaining tail. The `gfold_` function for the `Empty` base class just returns the initial value. It is declared `protected`, as `gfold_` is only intended to be called from `gfold` or from the `gfold_` method in a derived class. Note that in contrast to the general case discussed in section 3, `gfold` for `Insert` behaves like `gfoldl`, folding over data members in the order in which they are specified in the definition of the composition.

Providing static heterogeneous containers through metaprogramming and defining algorithms operating on these containers is not a new idea in C++(Winch, 2001). The Boost.Fusion library (de Guzman et al., 2008) pursues this idea systematically, providing heterogeneous maps, folds, lazy construction by *views*, and more. In particular, it offers return-type deduction for the functions it provides, which allows one to overcome the problems discussed in section 3 and to implement SYB quite directly. We do not use this library for our presentation, as we intend to make explicit the issues of porting SYB to generic C++ rather than hiding them in constructs of de-facto standard libraries.

6 SYB for Classes

Thus far, we have shown that the SYB approach can be implemented in C++ without a loss of genericity and applicability: each feature in the Haskell implementation can be mapped to the corresponding feature of C++ in a quite reasonable way. Yet, the philosophies of the two languages differ. Some assumptions that are natural in Haskell are less likely to apply in C++. In this section, we comment on the differences we observed.

One important difference concerns encapsulation. C++ depends on encapsulation at the level of data types rather than at the level of modules as Haskell. A class rarely exposes its members directly but instead provides a public interface to perform all actions. One reason for encapsulation is that internals of a class in C++ often represent low-level implementation details that should be hidden from the client of the class. The more important reason, however, is that objects encapsulate a state and need to protect this state from being freely modified. Methods like `gfold`, which expose all members, thus break encapsulation in the C++ sense. A class-based application of SYB would therefore use `gfold` most likely for low-level tasks directly related to the data members of a class, for example, for memory-related operations or the export of the state of an object to an external storage format. We note that it would not help to declare `gfold` as a private member function. While the data members then indeed would be invisible, the recursive traversal would at the same time stop to work, since a private method cannot be invoked outside its own class.

As an immediate technical consequence of encapsulation, one is prevented from providing a specialization of `gfold` after completing the definition of a class with private members, which is particularly a problem for classes in legacy code one wishes to adopt. For example, the definition of `gfold` for the `Employee` class in section 4 had to be defined as a `friend` to access the private data members; such friend declarations can only be provided when a class is defined and cannot be retrofitted later.

Classes in C++ are characterized by a separation of interface and implementation. Many classes are even considered *abstract data types* (ADTs), which are types that provide a set of values and associated operations independently from the implementation, by a public interface. Both interfaces and ADTs indicate that one might want to traverse a class in more than one way. The `gfold` function, as discussed so far, always treats a class as a collection of members. To additionally support the interface or ADT view, a class could provide a high-level view of itself for the purpose of folding. In such *interface traversal*, the class designer controls over which subterms `gfold` should fold. For example, SYB could be enabled to treat the Standard Template Library (STL) containers (Stepanov and Lee, 1994 (revised in October 1995 as tech. rep. HPL-95-11) differently from other types by accessing the members on the interface level, by the means of the iterator's interface. In general, interface traversal could be based on concepts (Gregor et al., 2006a; Reis and Stroustrup, 2006), where a class would be subject of a specific interface traversal if it implements a concept that provides the necessary interface.

Finally, the SYB approach depends on the explicit representation of data; data not explicitly represented but only computed, is not taken into account during traversal. In illustration, imagine that the "paradise" example company has a group of employees with a different kind of contract, which specifies their salary implicitly as a function of the length of their tenure—those employees would not benefit from the salary increase implemented in terms of SYB. The separation of interface and implementation makes it transparent to the clients whether, say, a salary exists as a data member or rather is the result of a method, while the SYB approach forces class designers to both hard-wire and expose their decision. The conflict between SYB and data abstraction is not unique to C++ (Nogueira Iglesias, 2006), but the issue is more relevant there, since the explicit structure is not even available for standard lists in C++.

7 Generalization and Extension

In its fully-developed form (Lämmel and Peyton Jones, 2003; Lämmel and Peyton Jones, 2004, 2005), the SYB pattern is more than just the three combinators presented so far. In this section, we discuss some of the issues involved in implementing the remaining features within C++.

We start by discussing how to extend a query or transformation with a new type-specific case, which we anticipate to be straightforward to implement. We continue with the issue of parallel traversal, and note that it would require significant changes to the whole implementation of SYB in C++. Finally, we discuss the implementation of generic queries, which pose interesting questions regarding the genericity of the traversal combinators.

We intentionally focus on algorithms consuming data structures; the implementation of SYB-style producer-algorithms in C++ remains future work.

7.1 Type-Case Extensions

In the SYB approach, type-specific functions have to be made polymorphic. Sometimes, such "extended" functions have to be extended by new type-specific cases. The task of extending polymorphic functions with new type-specific cases applies equally to queries and transformations; we use query extension in our discussion after the second SYB paper (Lämmel and Peyton Jones, 2004). In Haskell, queries are extended by the function `extQ`, which takes a generic query and extends it with a new type-specific case. The implementation of `extQ` has several shortcomings (Lämmel and Peyton Jones, 2005), most of them boil down to the fact that `extQ` relies on a dynamic cast operation. The most significant issue, however, is that a practical default case cannot be defined without resorting to mutual recursion and therefore losing extensibility.

The extensibility of `extQ` becomes problematic since the functions in the mutual recursion rely on calling each other by name. Therefore, one cannot simply change one without changing the other. The mutual recursion problem is resolved in Haskell by using type-classes since they enable a function to be overloaded with a type-specific meaning instead of having to introduce a new name. The same solution can be applied in C++; we have already seen overloaded functions at work in the definition of the type-extension function `mkT` in section 3.3.

7.2 Parallel traversals

Within the Haskell implementation of SYB, parallel traversal is defined in terms of zip-like functions, which traverse the elements of two heterogeneous containers in parallel (Lämmel and Peyton Jones, 2004). Since the implementation of `gfold` corresponds to an *internal* iteration of the elements to be folded, such zip-like function cannot easily be defined in a non-lazy programming language (Kühne, 1999). Zipping two sequences requires stepping through the sequences one element at a time. Since `gfold` applies the supplied function to all subterms in one go, there is no easy way to advance one subterm at a time. Even with support for laziness, the iteration would still be complicated since side-effects would also have to be applied in a zip-like manner.

Traditionally, C++ iteration is implemented in terms of *external* iterators, where the caller decides how the traversal proceeds. Implementing SYB in terms of external iterators would however affect almost the entire implementation. In section 5, we discussed the Boost.Fusion library, which implements iterators for traversal of *heterogeneous containers* (de Guzman et al., 2008), and recent activities towards support for iterators of hierarchies as required for SYB (Niebler, 2006).

7.3 Generic queries

The major remaining part of the SYB approach is the implementation of generic queries and generic query combinators. In Haskell, these are represented by the two functions `gmapQ`, corresponding to `gmapT`, and `everything`, corresponding to `everywhere`. Implementing either of these constructs in C++ does not constitute a fundamental problem, but the implementation would be different because of the lack of laziness. The Haskell implementation of `gmapQ` applies a function to each of the subterms of a given term, and then collects the output in a list. For performance reasons, constructing a list of values to be folded is not an option in C++. Instead, the subterms would be folded directly.

Reasoning about the implementation of `everything` however raises the question whether the genericity of the recursive traversal combinators could be increased. Cur-

rently, the two combinators `everywhere` and `everything` construct the same kind of hierarchical traversal, but by using different mapping functions, namely `gmapT` and `gmapQ`.

Can we devise a more generic construct where `everywhere` could be obtained by supplying `gmapT` as the one-layer traversal? An obvious strategy is to specify the recursive traversal in terms of `gfold` instead of `gmapT`. Such a construct, however, is complicated and the convenience of `gmapT` and `gmapQ` is lost.

In section 6, we discussed different ways of folding over the same term, and this motivates parameterizing `everything` and `everywhere` by a folding strategy. The introduction of another parameter is an argument in favor of merging the two, into a more general construct, to avoid duplication.

8 Related Work

Work related to ours can be divided into three categories: approaches to recursive traversal, approaches to automation, and C++ techniques for functional-style programming and advanced type computation. We discuss the first two categories from the standpoint of imperative, object-oriented programming; a good overview of work related to the original SYB approach can be found elsewhere (Hinze and Löh, 2006; Hinze et al., 2006; Lämmel and Peyton Jones, 2003).

The natural approach to recursive traversal in the object-oriented world is the *visitor* design pattern (Gamma et al., 1994). The basic visitor pattern has severe limitations. For one, the action to be performed and the traversal strategy are mixed together (Palsberg and Jay, 1998). This can be remedied to some degree by *visitor combinators* (Van Deursen and Visser, 2004; Visser, 2001), which allow traversal to be encoded separately from actions. Unlike visitor combinators, our approach does not exploit object-oriented features; we explicitly focus on the generic features of C++. Furthermore, visitor combinators as originally presented cannot be easily applied across different class hierarchies while our approach is universally applicable to any type for which `gfold` is defined. The visitor pattern can be generalized to a non-object-oriented setting. The Loki library (Alexandrescu, 2001) provides a generic visitor pattern for C++ and the Boost Graph Library (BGL) (Siek et al., 2002) employs the visitor pattern to implement generic graph algorithms. Still, even when generalized beyond objects, the visitor pattern does not provide full control over traversal. In C++, traversal is usually represented by *iterators* as in the Standard Template Library (STL, Stepanov and Lee, 1994 (revised in October 1995 as tech. rep. HPL-95-11). When it comes to generic traversal, STL-style iterators have two shortcomings, namely, they are designed to iterate over a range of homogeneous values and they impose a performance penalty when iterating over tree-like structures. The first issue has partly been addressed in the Boost.Fusion library (de Guzman et al., 2008), the second issue with the introduction of *segmented iterators* (Austern, 2000), which can efficiently traverse non-linear ranges. A technique similar to segmented iterators can be applied to heterogeneous containers (Niebler, 2006).

Automation of the SYB approach requires explicit type representation or at least means for reflection on the structure of a type. The SYB approach has been encoded in the "spine" view where the structure of a type is represented explicitly (Hinze and Löh, 2006; Hinze et al., 2006). In our approach, an explicit representation of the structure of values rather than types; values are represented as heterogeneous containers. As explained in section 5, the Boost.Fusion library uses a similar technique.

Functional style programming and advanced type computations for C++ are well-explored topics. The Boost library collection (Boost) provides the function and lambda (Järvi et al., 2003) libraries for functional programming and the result_of library for type computation. These libraries, however, do not provide all features necessary for implementing SYB. The FC++ (McNamara and Smaragdakis, 2004a) library provides a powerful collection of techniques for functional programming and an implementation of a large part of the Haskell Standard Prelude (Peyton Jones, 2003); our implementation could hide technical aspects of currying, higher-order functions, and other functional features in C++ by using FC++. Lastly, the C++ language allows, for example, static or dynamic and strict or lazy computations. Techniques for uniform expression of all available types of computation (Priesnitz and Schupp, 2006) could make our implementation more widely applicable.

9 Conclusions and Future Work

The SYB pattern in Haskell depends on a number of features that are not directly available in C++, most notably rank-2 types, higher-order functions, and polymorphic type extension. As we have shown in this paper, however, SYB can be implemented in C++ in a way that resembles the original Haskell version quite closely. Not surprisingly, the C++ solution depends heavily on the template feature; both class templates and function templates are used. Other relevant features and idioms include function objects, template member functions, and function overloading.

One of the important design decisions for SYB in C++ concerns the distinction between in-place and out-of-place transformation. In Haskell, the `gmapT` function supports out-of-place transformation, while the natural choice in C++ rather is an in-place transformation. As we have seen, with in-place semantics some parameters from the original Haskell signature become trivial in C++. Of course, it is also possible to support transformation with out-of-place semantics.

The formulation of polymorphic return types that depend on the types of input parameters presents an important technical difficulty for SYB in C++. In Haskell, type signatures allow for type checks of those dependent return types much as any other types. In C++, on the other hand, return types that depend on the types of input parameters must be modeled as class templates that represent the return type computation. Although neither conceptually nor technically very difficult to write, these classes mean extra work and impact the readability of code and, famously, of error messages. At the same time, C++ also simplifies matters: the issue of type extension, which makes the introduction of a special type cast construct necessary in the Haskell implementation, can be handled by function overloading and common specialization techniques. An advantage of the C++ solution, finally, is that it can be provided entirely within the current language standard.

Most of the SYB combinators can be provided once and for all, independently of specific types. The basic one-layer traversal combinator `gfold`, however, must be defined once per type. Although the code for `gfold` can be easily generated by a compiler extension, it nevertheless remains a burden. The need for an external tool can be avoided, granted an explicit representation of types is available within the implementation language. Given such representation, a metaprogram can be used to generate the definition of `gfold`. In section 5, we proposed a particular solution to generating `gfold` and discussed corresponding design issues.

While implementing SYB in C++ we observed that some assumptions, natural in

a pure, functional language, are less likely to apply in an imperative language with classes. In particular, SYB views a type as a publicly available collection of its elements. Yet, a class type in C++ often contains internal, low-level representation details that must be hidden. Furthermore, it is natural that a class type provides a public interface to its functionality, independently of the implementation. Consequently, we proposed *interface traversal* for SYB where one-layer traversal can be performed in terms of the one-layer traversal interface that a class provides, rather than directly traversing its members. Finally, SYB assumes that all interesting properties of a type are hard-wired and exposed as one of the explicit constituents, be it a class member or an argument of type constructor application.

In section 7 we discussed possible extensions and generalizations of our implementation outlining the directions for future work. Currently, our implementation does not fully cover SYB functionality. Particularly, we consider implementing query combinators, parallel traversal, and investigating possible generalizations of SYB traversal schemes. Generic producer algorithms (Lämmel and Peyton Jones, 2004) constitute an important direction of future work that we have not investigated in this paper. Finally, an extension of our approach to dynamic analysis and traversal of data structures seems natural and promises to be useful for modeling large systems.

Acknowledgments

This work was in part supported by Vetenskapsrådet as well as the project CEDES, which is funded within the Intelligent Vehicle Safety Systems (IVSS) program, a joint research program by the Swedish industry and government. We thank the reviewers of WGP for pointers to related work and suggestions that helped improving the presentation of the paper.

Appendix A—

```
#include <iostream>
#include <list>
#include <numeric>

// Helper traits
//  to determine function result and parameter types
//  limited to traits required by the example
template <typename Fun_>
struct Result {
    typedef typename Fun_::Result Type;
};
template <typename Fun_>
struct Param1 {
    typedef typename Fun_::Param1 Type;
};
template <typename Result_, typename P1_, typename P2_>
struct Param1<Result_ (*)(P1_,P2_)> {
    typedef P1_ Type;
};

// Implementation of gfold
//  base case followed by standard list example
template <typename Fun_, typename InitFun_,
          typename Param_>
typename Result<InitFun_>::Type
gfold(Fun_ fun, InitFun_ initFun, Param_& param) {
    return initFun(param);
}

template <typename Fun_, typename InitFun_, typename T>
typename Result<InitFun_>::Type
gfold(Fun_ fun, InitFun_ initFun, std::list<T>& param) {
    return std::accumulate<
             typename std::list<T>::iterator,
             typename Result<InitFun_>::Type>(
        param.begin(), param.end(), param, fun);
}

// Implementation of gmapT
//  including required helper function objects
template <typename Fun_, typename Param1_>
struct ApplyToSecond {
```

```
43      typedef Param1_ Result;
44      typedef Param1_ Param1;
45      Fun_ fun;
46      ApplyToSecond(Fun_ fun) : fun(fun) {}
47      template <typename Param2_>
48      Param1_ operator()(Param1_ p1, Param2_& p2) {
49          fun(p2);
50          return p1;
51      }
52  };
53  template <typename Bound_>
54  struct Bind {
55      typedef Bound_ Result;
56      Bound_ bound;
57      Bind(Bound_ bound) : bound(bound) {}
58      template <typename Param_>
59      Result operator()(Param_&) {
60          return bound;
61      }
62  };
63  template <typename Fun_, typename Param_>
64  Param_& gmapT(Fun_ fun, Param_& param) {
65      return gfold(ApplyToSecond<Fun_,Param_&>(fun),
66                   Bind<Param_&>(param),
67                   param);
68  }
69
70  // Implementation of everywhere
71  //  function object followed by convenience wrapper
72  template <typename Fun_>
73  struct Everywhere {
74      Fun_ fun;
75      Everywhere(Fun_ fun) : fun(fun) {}
76      template <typename Param_>
77      Param_& operator()(Param_& param) {
78          return fun(gmapT(*this,param));
79      }
80  };
81  template <typename Fun_, typename Param_>
82  Param_& everywhere(Fun_ fun, Param_& param) {
83      return (Everywhere<Fun_>(fun))(param); }
84  // Implementation of MkT
85  //  function object followed by convenience wrapper
86  template <typename Fun_>
87  struct MkT {
88      Fun_ fun_;
89      MkT(Fun_ fun) : fun_(fun) {}
90      template <typename Param_>
91      Param_& operator()(Param_& param) {
92          return param;
93      }
94      typename Param1<Fun_>::Type
95      operator()(typename Param1<Fun_>::Type param) {
96          return fun_(param);
97      }
```

```
98 };
99
100 template <typename Fun_>
101 MkT<Fun_> mkT(Fun_ fun) { return MkT<Fun_>(fun); }
102
103 // Implementation of explicit data-types
104 //  Class At for accessing the N:th element of a map
105 //  The Empty base case and the Insert cons constructor
106 template <unsigned int N_, class Map_> class At;
107
108 struct Empty {
109     template <typename Fun_, typename Init_>
110     typename Result<Fun_>::Type
111     gfold_(Fun_ fun, Init_& init) { return init; }
112 };
113
114 template <typename Data_, class Base_>
115 class Insert : public Base_ {
116     Data_ data_;
117 public:
118     Insert(Data_ data) : Base_(), data_(data) {}
119     template <typename D1_, typename D2_>
120     Insert(D1_ d1, D2_ d2)
121       : Base_(d2), data_(d1) {}
122     template <typename D1_, typename D2_, typename D3_>
123     Insert(D1_ d1, D2_ d2, D3_ d3)
124       : Base_(d2, d3), data_(d1) {}
125 protected:
126     template <typename Fun_, typename Init_>
127     typename Result<Fun_>::Type
128     gfold_(Fun_ fun, Init_& init) {
129         return Base_::gfold_(fun,fun(init,data_));
130     }
131 public:
132     template <typename Fun_, typename InitFun_>
133     friend typename Result<Fun_>::Type
134     gfold(Fun_ fun, InitFun_ initFun, Insert& param) {
135         return param.gfold_(fun,initFun(param));
136     }
137     friend typename At<1, Insert>::Result
138     At<1, Insert>::at(Insert& i);
139 };
140
141 template <unsigned int N_, class Data_, class Base_>
142 struct At <N_, Insert<Data_, Base_> >
143   : At<N_ - 1, Base_> {};
144 template <class Data_, class Base_>
145 struct At <1, Insert<Data_, Base_> > {
146     typedef Data_ & Result;
147     static Result at(Insert<Data_, Base_> & i) {
148         return i.data_;
149     }
150 };
151 template <unsigned int N_, class Map_>
152 typename At<N_, Map_>::Result at(Map_& m) {
```

```
153     return At<N_, Map_>::at(m);
154 }
155
156 // The C++ type hierarchy
157 //  DeptUnit omitted for simplicity
158 typedef const char *Name, *Address;
159 typedef Insert<float,Empty> Salary;
160 typedef Insert<Name,Insert<Address,Empty> > Person;
161 typedef Insert<Person,Insert<Salary,Empty> >
162     Employee, Manager;
163 typedef Insert<Employee,Empty> PersonUnit, SubUnit;
164 typedef Insert<Name,Insert<Manager,
165     Insert<std::list<SubUnit>,Empty> > > Dept;
166 typedef Insert<std::list<Dept>,Empty> Company;
167
168 // Implementation of currying in C++
169 template <typename Fun_>
170 struct Curry;
171 template <typename Result_, typename P1_, typename P2_>
172 struct Curry<Result_ (*)(P1_,P2_)> {
173     typedef Result_ Result;
174     typedef P2_ Param1;
175     Result_ (*fun)(P1_,P2_);
176     P1_ value;
177     Curry(Result_ (*fun)(P1_,P2_), P1_ value)
178       : fun(fun), value(value) {}
179     Result operator()(Param1 param1) const {
180         return fun(value, param1);
181     }
182 };
183
184 // Implementation of incS
185 //  binary function followed by curried unary function
186 Salary& incS(float k, Salary& salary) {
187     at<1>(salary) *= 1+k;
188     return salary;
189 }
190 Curry<Salary& (*)(float,Salary&)> incS(float k) {
191     return Curry<Salary& (*)(float,Salary&)>(incS,k);
192 }
193
194 // Implementation of increase algorithm
195 Company& increase(float k, Company& company) {
196     return everywhere(mkT(incS(k)), company);
197 }
198
199 // Implementation of main program
200 //  helper function for list insertion followed by
201 //  a main procedure constructing the "paradise"
202 //  example hierarchy and printing Blair's salary
203 //  after a 20% salary increase
204
205 template <typename T>
206 std::list<T> append(T t, std::list<T> list) {
207     list.push_back(t);
```

```
208     return list;
209 }
210
211 int main() {
212     Company company(append(Dept("Research", Employee(
213         Person("Ralf", "Amsterdam"), Salary(8000)),
214         append(PersonUnit(Employee(Person("Joost",
215         "Amsterdam"), Salary(1000))), append(PersonUnit(
216         Employee(Person("Marlow", "Cambridge"), Salary(
217         2000))),std::list<SubUnit>()))), append(Dept(
218         "Strategy", Employee(Person("Blair", "London"),
219         Salary(100000)), std::list<SubUnit>()),
220         std::list<Dept>())));
221
222     increase(0.20f, company);
223
224     std::cout << at<1>(at<2>(at<2>(
225         at<1>(company).front()))) << std::endl;
226 }
```

Paper VI

A Semantic Definition of Separate Type Checking in C++ with Concepts

Abstract syntax and the complete semantic definition

Some semantic rules are changed in this version of the report, comparing to the published version.

A Semantic Definition of Separate Type Checking in C++ with Concepts

Abstract syntax and the complete semantic definition

Marcin Zalewski
Technical Report No. 2008:12
Chalmers University of Technology, Sweden

1 Introduction

The C++ language (ISO, 2003b) is currently being extended by concepts (Gregor et al., 2006b; Dos Reis and Stroustrup, 2006); we refer to the extended language as ConceptC++. Concepts provide an interface between *templates* and template arguments. Concept-constrained templates can be checked before they are used, in contrast to unconstrained templates in the current C++ that can only be fully checked when used.

This report describes a simplified version of the concept sublanguage that represents integral elements of ConceptC++. The purpose of the language is to capture the basic semantics of separate type checking with concepts as specified in (Gregor et al., 2008e). This report provides the syntax and the semantics along with a low-level discussion; the overview, motivation, and high-level discussion are meant to be provided elsewhere. The abstract syntax definition and the semantic rules in this report are generated from a meta-description in Ott (Sewell et al., 2007), a tool for developing semantic definitions of programming languages. Semantic rules are given as, sometimes mutually-recursive, relations on the abstract syntax.

The features of C++ with concepts included in our formalization are:

- Concepts
  - refinement clauses
  - associated types
  - associated functions
- Templates
  - requirement clauses
  - single-expression bodies
- Concept maps
  - implementations for associated types and functions
- Auxiliary features
  - overload resolution
  - simple expressions

The following features are not supported:

- Template parameters

- Concept parameters
- Template concept maps
- Concept-based overloading
- Candidate set implementations of associated functions

2 Syntax

We describe the syntax of the core ConceptC++ language in the BNF form.

Some productions have annotations to the right of the right-hand side. The following annotations are understood by Ott.

- M indicates a metaproduction. These are not part of the free grammar for the relevant nonterminal, but instead are given meaning (in the theorem prover models) by translation into non-metaproductions. The translations, specified in the Ott source, are specific to each theorem prover.
- S is identical to M except that productions marked with S are admitted in parsing concrete terms.

The following annotations are for informational purposes only.

- [I] indicates a production that is not intended to be available in user programs but is useful in the metatheory.
- [L] indicates a library-implemented facility.

Since Ott currently does not provide a way to define functions (only relations are supported), some simple functions are embedded into the abstract syntax. Functions are usually indicated with function call syntax and a return type of the function is the non-terminal for which the function's production is given.

Note that some rules repeat production in other rules; this is intended and the repetition is captured by *subrule* constraints in section 2.2.

Also note that some terminals (see the "terminals" production in the grammar) are pretty printed identically to a non-terminal; however, they are not the same. It should be clear at the point of use if a particular symbol refers to a terminal or a non-terminal.

Furthermore, Ott synthesizes a `user_syntax` grammar (the last grammar rule) that lists all syntax available to the user.

2.1 Grammar Rules

typ_identifier	*id* in cpp standard, here type name
f	*id* in cpp standard, here function name
c	*id* in cpp standard, here concept name
tn	template name
var	variable names

v_{bool}	boolean value
v_{int}	integral value
i, j, k, n, m	index variables used in the grammar and the semantic rules

id_τ ::= type identifier
- | *typ_identifier* — type name
- | $id_\tau(tydef)$ M[I] — extract from *tydef*

$\overline{id_\tau}$ ::= type identifier sequence
- | $\overline{id_{\tau i}}^{\,i}$ M[I] — many
- | $\overline{id_\tau}(\overline{ty^a})$ M[I] — extract from $\overline{ty^a}$
- | $\overline{id_\tau}(\overline{tydef})$ M[I] — extract from $\overline{tydef}$

con ::= concept name
- | *c* — concept name

cid, cid_{cmap}, cid_{impl} ::= concept identifier
- | *con* — concept name
- | $cid(cm)$ M[I] — extract from *cm*

cid? ::= concept identifier option
- | *None* M[I] — none
- | *cid* M[I] — some
- | $icm.cid$ M[I] — extract from *icm*

$\overline{cid}$, $\overline{cid}^{\prec_1}$ ::= concept identifier sequence
- | *cid* M[I] — one
- | $\overline{\overline{cid}}^{\,i}$ M[I] — many
- | $\{cid \in \overline{cid} \mid formula\}$ M[I] — set comprehension for $\overline{cid}$

scope ::= scope (used in qualified names)
- | *con* — concept name
- | $cid(cm)$ M[I] — extract from *cm*

req ::= requirements clause
- | `requires` $\overline{cid_i}^{\,i}$ — def.

τ ::= type specifier
- | `int` — built-in
- | `void` — built-in
- | `bool` — built-in
- | id_τ — type name
- | $scope :: id_\tau$ — qualified type name

bintype ::= built-in types
- | `int`
- | `void`
- | `bool`

$\tau?$ ::= type specifier option
- | *None* M[I] none
- | τ M[I] some
- | $\Gamma(var)$ M[I] variable environment lookup
- | **find_typesp** $(\overline{\tau_{defs}}, \tau)$ M[I] find τ in $\overline{\tau_{defs}}$

$\overline{\tau}$ ::= type specifier sequence
- | τ M[I] one
- | $\overline{\overline{\tau}_i}^{\,i}$ M[I] many

val ::= values
- | v_{int} integer value
- | v_{bool} boolean value
- | `obj` τ object value

e ::= expressions
- | *var* variable
- | `obj` τ object
- | $f(\overline{e_i}^{\,i})$ application
- | v_{int} integral
- | v_{bool} boolean
- | (e) S parenthesis

Γ ::= type environment ($var \rightharpoonup \tau$)
- | $\emptyset$ M[I] empty environment
- | $[\overline{var_i : \tau_i}^{\,i}]$ M[I] convert function parameters to type environment

tparam ::= type parameter
- | `typename` id_τ def.

tl ::= template
- | `template` $req\,tn\,\{e\}$ def.

T ::= templates environment
- | $\emptyset$ M[I] empty
- | *tl* M[I] one
- | $\overline{T_i}^{\,i}$ M[I] flatten

ty^a ::= associated type
- | *tparam* def.

$\overline{ty^a}$, $\overline{ty^a}^{\prec}$ ::= associated type sequence
- | ty^a M[I] one
- | $\overline{\overline{ty^a}_i}^{\,i}$ M[I] many

f^a ::= associated function
- | $\tau f(\overline{\tau})$ return type, function name, parameters

$\overline{f^a}$, $\overline{f^a}^{\prec}$ ::= associated function sequence
- | f^a M[I] one
- | $\overline{\overline{f^a}_i}^{\,i}$ M[I] many

cp ::= concept
| con `refines` $\overline{cid}\,\{\,\overline{ty^a}\;\overline{f^a}\,\}$ def.

$cp?$ ::= concept option
| *None* M[I] none
| cp M[I] some
| *find-concept*(C, con) M[I] find a concept

C ::= concept sequence
| $\emptyset$ M[I] empty
| cp M[I] one
| $\overline{C_i}^{\,i}$ M[I] many

sig ::= function signature
| $\tau f\,(\overline{\tau})$ return type, name, parameters
| $sig\,(fdef)$ M[I] from $fdef$

$sig?$ ::= function signature option
| *None* M[I] none
| sig M[I] some

$sig?_{amb}$ ::= signature option or ambiguity
| ¿? [I] ambiguous
| $sig?$ [I] sig option

$\overline{sig}$, $\overline{sig}^{\prec}$ ::= function signature sequence
| sig M[I] one
| $\overline{\overline{sig_i}}^{\,i}$ M[I] flatten
| $rdup\,(\overline{sig})$ M[I] remove duplicates
| $\overline{sig}\,(\overline{fdef})$ M[I] from $\overline{fdef}$
| $\overline{sig_1} \setminus \overline{sig_2}$ M[I] set difference

$fdef$ ::= function definition
| $\tau f\,(\overline{var_i\colon \tau_i}^{\,i})\,\{\,e\,\}$ return type, name, parameters, body

$fdef?$ ::= function declaration option
| *None* M[I] none
| $fdef$ M[I] some
| $\overline{fdef} \vdash find\,(sig)$ M[I] find definition

$\overline{fdef}$, $\overline{fdef}^{\prec}$ ::= function definition sequence
| $fdef$ M[I] one
| $\overline{\overline{fdef}_i}^{\,i}$ M[I] many
| $\{\,fdef \in \overline{fdef} \mid formula\,\}$ M[I] set comprehension for $\overline{fdef}$

$tydef$ ::= type definition
| `typedef` $\tau\, id_\tau$ alias id_τ to τ

$\overline{tydef}$, $\overline{tydef}^{\prec_1}$ ::= type definition sequence
- | *tydef* M[I] one
- | $\overline{\overline{tydef}_i}^{i}$ M[I] flatten
- | $\{\, tydef \in \overline{tydef} \mid formula \,\}$ M[I] set comprehension
- | *find-defs*$(id_\tau, \overline{tydef})$ M[I] find definitions for id_τ

tydef? ::= type definition option
- | *None* M[I] none
- | *tydef* M[I] some

cm ::= concept map
- | `concept_map` *cid* $\{\, \overline{tydef}\,\overline{fdef} \,\}$ def.
- | *icm* . *cm* M[I] implicit concept map

$\overline{cm}$ ::= concept map sequence
- | *cm* M[I] one
- | $\overline{\overline{cm}_i}^{i}$ M[I] many
- | *rm*?$(\overline{cm?})$ M[I] remove "none" options

cm? ::= concept map option
- | *None* M[I] none
- | *cm* M[I] some
- | *cm*(M, cid) M[I] find *cid* in *M*

$\overline{cm?}$ ::= concept map option sequence
- | *cm*? M[I] one
- | $\overline{\overline{cm?}_i}^{i}$ M[I] many

icm ::= implicit concept map
- | $(cm, cid?)$ M[I] concept map and optional "parent"

$\overline{icm}$ ::= implicit concept map sequence
- | *icm* M[I] one
- | $\overline{\overline{icm}_i}^{i}$ M[I] flatten
- | $\{\, icm \in \overline{icm} \mid formula \,\}$ M[I] set comprehension

icm? ::= implicit concept map option
- | *None* M[I] none
- | *icm* M[I] some
- | *cm*(M, cid) M[I] find *cid* in *M*

M ::= concept map environment
- | $\emptyset$ M[I] empty
- | *icm* M[I] one
- | $\overline{M_i}^{i}$ M[I] flatten

inst ::= template instantiation
- | `inst` *tl* instantiation

$\tau pair$ ::=
- | (τ_1, τ_2) M[I] a tuple of type specifiers

$\overline{\tau_{defs}}$ ::=
- $\emptyset$ M[I] epmty
- $\tau pair$ M[I] one
- $\overline{\tau pair_i}^{\,i}$ M[I] flatten

itl ::= instantiated template
- $\overline{\tau_{defs}}\overline{fdef}\ tn\{e\}$ M[I] instantiated template

d_{user} ::= user definitions
- *cp* concept
- *tl* template
- *cm* concept map

P_{user} ::= user syntax program
- d_{user} M one
- $\overline{P_{user\,i}}^{\,i}$ M flatten

d ::= definitions
- *cp* concept
- *tl* template
- *cm* concepts map
- *icm* [I] implicit map

P ::= program
- *d* M one
- $\overline{P_i}^{\,i}$ M flatten

terminals	::=	pretty printing of terminals
	$\triangleright$	gives
	(	
	)	
	$\mapsto$	maps to
	$\vdash$	context
	$\cup$	union
	::	scope operator
	`refines`	
	`requires`	
	`int`	
	`bool`	
	`void`	
	`template`	
	$\prec_1$	direct refinement
	$\prec$	refinement
	{	
	}	
	`concept_map`	
	`typedef`	
	`typename`	
	`inst`	
	$\forall$	for all
	$\exists$	there exists
	$\in$	in
	$\notin$	not in
	$\vee$	or
	$\wedge$	and
	$\neq$	is not equal to
	,	separator (with correct spacing)
	[	left environment bracket
	]	right environment bracket
	:	types to
	¿?	ambiguity
	$\subseteq$	subset
	$\surd$	well-formed
	$\Downarrow$	instantiates to
	not defined in	
	defined in	
	$\setminus$	set minus for lists (preserves duplicates and order)
	$\neg$	
	$\mid$	
	$\approx$	set equality on lists
	$\emptyset$	empty
	find-rec	
	distinct	
	type-decls-check	
	fdefs-check	

$tydefs_{=\!=}$	
$fdefs_{=\!=}$	
$=\!=$	type definitions compatibility (one-level)
icms	
overload-res	
returns	
tydefs-check	
find-scope	
rdup	
con	
find-concept	
$\overline{f^a}$	
overload-set	
normalize	
normalize	
normalize	
refined	
find-rec	
rm?	remove *None* options
`obj`	object creation
None	empty option
$\overline{fdef}$	
$\overline{fdef}$	
cm	
$\overline{ty^a}$	extract required associated types for a concept and its refinements
$\overline{ty^a}$	extract associated types from a concept
$\overline{f^a}$	
id_τ	
$\overline{id_\tau}$	
$\overline{sig}$	extract function definitions from a concept map
$\overline{sig}$	extract function definitions from a concept map and its refinements
$\overline{tydef}$	
find-defs	
sig	
find-scope	
$\bullet$	separate a quantifier from a formula
fname	
params	
sig	
fdef	
con	
$\overline{f^a}$	
$\overline{ty^a}$	
$\overline{tydef}$	
refined	
find	
length	
matching	

formula ::=	formulas that can be used in premises		
	$judgement$		judgement
|	$formula_1 \ \ldots\ formula_k$	M[I]	conjunction of formulas
|	$(formula)$	M[I]	parenthesized
|	$\neg formula$	M[I]	formula negation
|	$cp \in P$	M[I]	concept definition in P
|	$sig \in \overline{sig}$	M[I]	sig in $\overline{sig}$
|	$tydef \in \overline{tydef}$	M[I]	type declaration in type declarations
|	$\overline{tydef}_1 = \overline{tydef}_2$	M[I]	type declarations aliasing
|	$\overline{tydef}_1 \approx \overline{tydef}_2$	M[I]	type declarations set equality
|	$\overline{fdef}_1 = \overline{fdef}_2$	M[I]	function definitions aliasing
|	$fdef_1 = fdef_2$	M[I]	function definition aliasing
|	$cp?_1 = cp?_2$	M[I]	concept option alias
|	$tydef?_1 = tydef?_2$	M[I]	type declaration option equality
|	$cm?_1 = cm?_2$	M[I]	concept map equality
|	$cid?_1 = cid?_2$	M[I]	cid options equality
|	$cid?_1 \neq cid?_2$	M[I]	concept id options inequality
|	$\overline{cm} = \overline{cm}'$	M[I]	pointwise equality of concept map lists
|	$cid \in \overline{cid}$	M[I]	cid is in $\overline{cid}$
|	$id_\tau \in \overline{id_\tau}$	M[I]	list of $\overline{id_\tau}$ contains a type named id_τ
|	$id_\tau \notin \overline{id_\tau}$	M[I]	id_τ not in $\overline{id_\tau}$
|	$cid \neq cid'$	M[I]	concept id inequality
|	$\overline{cid} = \overline{cid}'$	M[I]	concept id sets equality
|	$\overline{f^a} = \overline{f^a}'$	M[I]	associated function sets equality
|	$\overline{sig} = \overline{sig}'$	M[I]	function declarations equality
|	$\overline{sig} \approx \overline{sig}'$	M[I]	function declarations set equality
|	$\overline{\tau}_1 \approx \overline{\tau}_2$	M[I]	type specifiers set equality
|	$\tau?_1 \neq \tau?_2$	M[I]	type specifier option inequality
|	$\tau?_1 = \tau?_2$	M[I]	type specifier option alias
|	$formula \wedge formula'$	M[I]	and
|	$formula \vee formula'$	M[I]	or
|	$\exists cid \in \overline{cid} \bullet formula$	M[I]	there exists concept id in a set of ids
|	$\forall cid \in \overline{cid} \bullet formula$	M[I]	for all cid in cid set
|	$\forall tydef \in \overline{tydef} \bullet formula$	M[I]	for all $tydef$ in $\overline{tydef}$ set
|	$\forall fdef \in \overline{fdef} \bullet formula$	M[I]	for all $fdef$ in $\overline{fdef}$ set
|	$distinct(\overline{sig})$	M[I]	distinct function signatures
|	$distinct(\overline{id_\tau})$	M[I]	distinct type identifiers
|	$e = e'$	M[I]	expression alias
|	$\overline{sig} \subseteq \overline{sig}' matching f\ \overline{\tau}$	M[I]	$\overline{sig}$ in $\overline{sig}'$ matching call profile
|	$length \overline{sig} >= n$	M[I]	$\overline{sig}$ length greater than n
|	$length \overline{sig} = n$	M[I]	$\overline{sig}$ length equal n
|	$\overline{id_\tau}_1 \approx \overline{id_\tau}_2$	M[I]	type ids set equality
|	$id_\tau = id_\tau'$	M[I]	type identifier equality
|	$tl_1 = tl_2$	M[I]	template equality
|	tl not defined in T	M[I]	tl not defined in T (name check)
|	tl defined in T	M[I]	tl defined in T (definition check)
|	cp not defined in C	M[I]	concept not defined in concepts
|	cid defined in C	M[I]	a concept corresponding to cid exists in C

$\mid$ *cm* not defined in *M*	M[I]	*cm* is not defined in *M* (i.e., no map for *cid* of *cm*)
$\mid \asymp (tydef, \overline{tydef})$	M[I]	definition *tydef* is compatible with definitions $\overline{tydef}$
$\mid icm?_1 = icm?_2$	M[I]	equality of implicit concept map options
$\mid rm?(\overline{icm?_i}^{\,i}) = \overline{icm}$	M[I]	remove empty options
$\mid rm?(\overline{\tau?_i}^{\,i}) = \overline{\tau}$	M[I]	remove empty options
$\mid \overline{icm}_1 = \overline{icm}_2$	M[I]	compare lists of implicit concept maps
$\mid \overline{\tau_{defs}}; \overline{fdef}; \Gamma \vdash e : \tau$	M[I]	type insantiated template expressions

fdecls_name_def ::=
$\mid fname(sig) = f$ extract function name from a function signature

fdecl_return_def ::=
$\mid returns(sig) = \tau$ extract return type from a function signature

fdecl_params_def ::=
$\mid params(sig) = \overline{\tau}$ extract parameters from a function signature

construct_function_def ::=
$\mid sig(f, \tau, \overline{\tau}) = sig$ construct function signature

construct_fdef_def ::=
$\mid fdef(f^a, \overline{var_i}^{\,i}, e) = fdef$ construct function definition

cid_name_def ::=
$\mid con(cid) = con$ extract concept name from a concept identifier

concept_name_def ::=
$\mid con(cp) = con$ extract concept name from a concept definition

concept_functions_def ::=
$\mid \overline{f^a}(cp) = \overline{f^a}$ extract associated functions from a concept definition

cid_functions_def ::=
$\mid \overline{f^a}(C, cid) = \overline{f^a}$ extract associated functions given a concept identifier

cmap_functions_def ::=
$\mid \overline{fdef}(cm) = \overline{fdef}$ extract function definitions from a concept map

concept_fields_def ::=
$\mid \overline{ty^a}(cp) = \overline{ty^a}$ extract associated types from a concept definition

cid_types_def ::=
$\mid \overline{ty^a}(C, cid) = \overline{ty^a}$ extract associated types given a concept identifier

cmap_types_def ::=
$\mid \overline{tydef}(cm) = \overline{tydef}$ extract type definitions from a concept map

cid_refined_cids_def ::=
$\mid refined(C, cid) = \overline{cid}$ concept identifiers refinement

con_refined_cids_def ::=
$\mid refined(C, con) = \overline{cid}$ refined concepts of a concept

find_fcall_fdecls_def ::=
$\mid overload\text{-}res(f\,\overline{\tau}, \overline{sig}) = sig?_{amb}$ overload resolution

refines_dir_many ::=
| $C \vdash cid \prec_1 \overline{cid}$ direct concept refinement

cid_refines_many ::=
| $C \vdash cid \prec \overline{cid}$ concept refinement

con_refines_many ::=
| $C \vdash con \prec \overline{cid}$ concept refinement (for concept names)

find_type_def ::=
| *find-scope*$(C, cid, \tau) = \tau?$ type lookup, non-recursive

find_type_rec_def ::=
| *find-rec*$(C, cid, \tau) = \tau?$ recursive type lookup

find_fun_def ::=
| *find-scope*$(C, cid, f) = \overline{sig}$ non-recursive associated function lookup

get_overset_def ::=
| *overload-set*$(f, \overline{f^a}) = \overline{sig}$ generate overload set

normalize_fdecls_def ::=
| *normalize*$(C, cid, \overline{sig}) = \overline{sig}'$ fully qualify type names in an overload set

normalize_expr_def ::=
| *normalize*$(C, cid, e) = e'$ fully qualify type names in an expression

normalize_fdefs_def ::=
| *normalize*$(C, cid, \overline{fdef}) = \overline{fdef}'$ fully qualify type names in an overload set

find_fun_rec_def ::=
| *find-rec*$(C, cid, f) = \overline{sig}$ compute overload set recursively

T_def ::=
| $C; \Gamma; \overline{cid} \vdash e : \tau$ constrained context typing

typedecl_wf_def ::=
| *tydef* $\surd$ type definition well-formedness

required_assfs_def ::=
| $\overline{f^a}(C, cid) = \overline{f^a}$ required associated functions

required_assts_def ::=
| $\overline{ty^a}(C, cid) = \overline{ty^a}$ required associated types

defined_fdecls_ref_def ::=
| $\overline{sig}(C, M, cid?, cid) = \overline{sig}$ defined function signatures (for concept maps)

defined_fdefs_ref_def ::=
| $\overline{fdef}(C, M, cid?, cid) = \overline{fdef}$ defined functions (for concept maps)

defined_assts_def ::=
| $\overline{tydef}(M, \overline{cid}) = \overline{tydef}$ defined types (for concept maps)

assts_compat_def ::=
| $\overline{ty^a} \vdash \overline{tydef} \asymp \overline{tydef}^{\prec_1}$ one-level type definitions compatibility

typedecls_check_def ::=
| $tydefs\text{-}check\,(C, cid, \overline{tydef})$ type definitions check

typedecls_comatibility_def ::=
| $tydefs_{\asymp}\,(C, M, cid, \overline{tydef})$ type definitions compatibility

fdefs_check_def ::=
| $fdefs\text{-}check\,(C, cid, \overline{fdef})$ function definitions check

fdefs_compat_def ::=
| $fdefs_{\asymp}\,(C, M, cid_{cmap}, cid?, \overline{fdef})$ function definitions compatibility

implicit_cmaps_def ::=
| $icms\,(C, M, cid, cid?, \overline{tydef}, \overline{fdef}) = \overline{icm}'$ generate implicit concept maps

p_inst_def ::=
| $C; M; T \vdash P \Downarrow C'; M'; T'$ program instantiation

p_user_inst_def ::=
| $P \Downarrow C; M; T$ user program instantiation

judgement ::=
| *fdecls_name_def*
| *fdecl_return_def*
| *fdecl_params_def*
| *construct_function_def*
| *construct_fdef_def*
| *cid_name_def*
| *concept_name_def*
| *concept_functions_def*
| *cid_functions_def*
| *cmap_functions_def*
| *concept_fields_def*
| *cid_types_def*
| *cmap_types_def*
| *cid_refined_cids_def*
| *con_refined_cids_def*
| *find_fcall_fdecls_def*
| *refines_dir_many*
| *cid_refines_many*
| *con_refines_many*
| *find_type_def*
| *find_type_rec_def*
| *find_fun_def*
| *get_overset_def*
| *normalize_fdecls_def*
| *normalize_expr_def*
| *normalize_fdefs_def*
| *find_fun_rec_def*
| *T_def*
| *typedecl_wf_def*
| *required_assfs_def*
| *required_assts_def*
| *defined_fdecls_ref_def*
| *defined_fdefs_ref_def*
| *defined_assts_def*
| *assts_compat_def*
| *typedecls_check_def*
| *typedecls_comatibility_def*
| *fdefs_check_def*
| *fdefs_compat_def*
| *implicit_cmaps_def*
| *p_inst_def*
| *p_user_inst_def*

user_syntax ::=
| *typ_identifier*
| *f*
| *c*
| *tn*
| *var*
| v_{bool}
| v_{int}
| *i*
| id_{τ}
| $\overline{id_{\tau}}$
| *con*
| *cid*
| *cid*?
| $\overline{cid}$
| *scope*
| *req*
| τ
| *bintype*
| τ?
| $\overline{\tau}$
| *val*
| *e*
| Γ
| *tparam*
| *tl*
| *T*
| ty^{a}
| $\overline{ty^{a}}$
| f^{a}
| $\overline{f^{a}}$
| *cp*
| *cp*?
| *C*
| *sig*
| *sig*?
| $sig?_{amb}$
| $\overline{sig}$
| *fdef*
| *fdef*?
| $\overline{fdef}$
| *tydef*
| $\overline{tydef}$
| *tydef*?
| *cm*
| $\overline{cm}$
| *cm*?
| $\overline{cm?}$

| icm
| $\overline{icm}$
| $icm?$
| M
| $inst$
| $\tau pair$
| $\overline{\tau_{defs}}$
| itl
| d_{user}
| P_{user}
| d
| P
| $terminals$
| $formula$

2.2 Subrules

The above grammar is constrained by some *subrule* constraints that are checked by Ott. A subrule constraint $r1 \prec r2$ means that every production in $r1$ must also exist in $r2$ but $r2$ may have additional productions not in $r1$. The grammar above is constrained by the following subrule constraints:

- $val \prec e$,
- $bintype \prec \tau$,
- $f^a \prec sig$,
- $P_{user} \prec P$,
- $d_{user} \prec d$,
- $\overline{f^a} \prec \overline{sig}$,
- $cid \prec scope$.

3 Helper Judgements

This section contains the "helper" parts of the semantics, i.e., the simple but necessary parts. Each rule is listed in its own subsection. The type of the rule is given in a $\boxed{\text{box}}$. The type indicates which parts of the grammar are the "inputs" to a given rule. For example, the first rule in this section takes a signature *sig* and a function name f (*fname* and $=$ are terminals). The type is followed by a brief comment. Next, the rules defining a relation on abstract syntax are given along with a short commentary.

extract function name from a function signature

$$\boxed{fname\,(sig) = f}$$

FDECL_NAME

$$\overline{fname\,(\tau f\,(\overline{\tau})) = f}$$

Extract the name of a function from a function signature.

extract return type from a function signature

$$\boxed{returns\,(sig) = \tau}$$

FDECL_RETURN

$$\overline{returns\,(\tau f\,(\overline{\tau})) = \tau}$$

Extract return type from a function signature.

extract parameters from a function signature

$$\boxed{params\,(sig) = \overline{\tau}}$$

FDECL_PARAMS

$$\overline{params\,(\tau f\,(\overline{\tau})) = \overline{\tau}}$$

Extract parameters from a function signature

construct function signature

$$\boxed{sig\,(f, \tau, \overline{\tau}) = sig}$$

CONSTRUCT_FUNCTION

$$\overline{sig\,(f, \tau, \overline{\tau}) = \tau f\,(\overline{\tau})}$$

Construct a function signature $\tau f\,(\overline{\tau})$ from its parts.

construct function definition

$$\boxed{fdef\,(f^a, \overline{var_i}^i, e) = fdef}$$

CONSTRUCT_FDEF

$$\overline{fdef\,(\tau f\,(\overline{\tau_i}^i), \overline{var_i}^i, e) = \tau f\,(\overline{var_i : \tau_i}^i)\,\{\,e\,\}}$$

Construct a function definition from its parts.

extract concept name from a concept identifier

$$\boxed{con(cid) = con}$$

CID_NAME

$$\frac{}{con(con) = con}$$

Given a concept identifier *con*, return *con*. In this version of the semantics a concept identifier is just a concept name; in future versions this rule may become more complex.

extract concept name from a concept definition

$$\boxed{con(cp) = con}$$

CONCEPT_NAME

$$\frac{}{con(con\,\texttt{refines}\,\overline{cid}\,\{\overline{ty^a}\,\overline{f^a}\}) = con}$$

Given a concept $con\,\texttt{refines}\,\overline{cid}\,\{\overline{ty^a}\,\overline{f^a}\}$, extract concept name *con*.

extract associated functions from a concept definition

$$\boxed{\overline{f^a}(cp) = \overline{f^a}}$$

CONCEPT_FUNCTIONS

$$\frac{}{\overline{f^a}(con\,\texttt{refines}\,\overline{cid}\,\{\overline{ty^a}\,\overline{f^a}\}) = \overline{f^a}}$$

Given a concept $con\,\texttt{refines}\,\overline{cid}\,\{\overline{ty^a}\,\overline{f^a}\}$, extract associated function $\overline{f^a}$.

extract associated functions given a concept identifier

$$\boxed{\overline{f^a}(C, cid) = \overline{f^a}}$$

CID_FUNCTIONS

$$\frac{1.\ con(cid) = con \qquad 2.\ \textit{find-concept}(C, con) = con\,\texttt{refines}\,\overline{cid}\,\{\overline{ty^a}\,\overline{f^a}\}}{\overline{f^a}(C, cid) = \overline{f^a}}$$

Given an environment *C* and a concept identifier *cid*, extract associated functions of the concept designated by *cid*. First extract the concept name *con* from *cid* and then find the corresponding concept in *C* and return its associated functions $\overline{f^a}$. Note that there is no error handling; if *cid* does not name a defined concept then the second premise cannot be fulfilled and this judgment does not apply.

extract function definitions from a concept map

$$\boxed{\overline{fdef}\,(cm) = \overline{fdef}}$$

CMAP_FUNCTIONS

$$\overline{\overline{fdef}\,(\texttt{concept\_map}\,cid\,\{\,\overline{tydef}\;\overline{fdef}\,\})} = \overline{fdef}$$

Given a concept map $\texttt{concept\_map}\,cid\,\{\,\overline{tydef}\;\overline{fdef}\,\}$, return its associated function definitions $\overline{fdef}$.

extract associated types from a concept definition

$$\boxed{\overline{ty^a}\,(cp) = \overline{ty^a}}$$

CONCEPT_TYPES

$$\overline{\overline{ty^a}\,(con\,\texttt{refines}\,\overline{cid}\,\{\,\overline{ty^a}\;\overline{f^a}\,\})} = \overline{ty^a}$$

Given a concept $con\,\texttt{refines}\,\overline{cid}\,\{\,\overline{ty^a}\;\overline{f^a}\,\}$, return its associated types $\overline{ty^a}$.

extract associated types given a concept identifier

$$\boxed{\overline{ty^a}\,(C, cid) = \overline{ty^a}}$$

CID_TYPES

$$\frac{1.\ con\,(cid) = con \qquad 2.\ \textit{find-concept}\,(C, con) = con\,\texttt{refines}\,\overline{cid}\,\{\,\overline{ty^a}\;\overline{f^a}\,\}}{\overline{ty^a}\,(C, cid) = \overline{ty^a}}$$

Given an environment C and a concept identifier cid, extract associated types of the concept designated by cid. First extract the concept name con from cid and then find the corresponding concept in C and return its associated types $\overline{ty^a}$. Note that there is no error handling; if cid does not name a defined concept then the second premise cannot be fulfilled and this judgment does not apply.

extract type definitions from a concept map

$$\boxed{\overline{tydef}\,(cm) = \overline{tydef}}$$

CMAP_TYPES

$$\overline{\overline{tydef}\,(\texttt{concept\_map}\,cid\,\{\,\overline{tydef}\;\overline{fdef}\,\})} = \overline{tydef}$$

Given an environment C and a concept identifier cid, extract associated types of the concept designated by cid. First extract the concept name con from cid and then find the corresponding concept in C and return its associated types $\overline{ty^a}$. Note that there is no

error handling; if *cid* does not name a defined concept then the second premise cannot be fulfilled and this judgment does not apply.

concept identifiers refinement

$$\boxed{\mathit{refined}\,(C, \mathit{cid}) = \overline{\mathit{cid}}}$$

CRC_REF_CIDS

$$\frac{1.\ \mathit{con}\,(\mathit{cid}) = \mathit{con} \qquad 2.\ \mathit{find\text{-}concept}\,(C, \mathit{con}) = \mathit{con}\,\texttt{refines}\,\overline{\mathit{cid}}\,\{\overline{\mathit{ty}^a}\ \overline{f^a}\}}{\mathit{refined}\,(C, \mathit{cid}) = \overline{\mathit{cid}}}$$

A concept identifier designates a concept *use*. This judgment extracts concept identifiers for concepts that a concept designated by a concept identifier *cid* refines. The concept designated by *cid* is looked up in the concept environment C and its refinements are transitively extracted and converted to appropriate concept identifiers $\overline{\mathit{cid}}$. In the current version of the semantics, $\overline{\mathit{cid}}$ is a sequence of concept names. Note that if *cid* names an undefined concept there is no explicit error handling, the judgment simply does not apply then.

refined concepts of a concept

$$\boxed{\mathit{refined}\,(C, \mathit{con}) = \overline{\mathit{cid}}}$$

CONRC_REF_CIDS

$$\frac{1.\ \mathit{find\text{-}concept}\,(C, \mathit{con}) = \mathit{con}\,\texttt{refines}\,\overline{\mathit{cid}}\,\{\overline{\mathit{ty}^a}\ \overline{f^a}\}}{\mathit{refined}\,(C, \mathit{con}) = \overline{\mathit{cid}}}$$

Given a concept name *con*, find the corresponding concept $\mathit{con}\,\texttt{refines}\,\overline{\mathit{cid}}\,\{\overline{\mathit{ty}^a}\ \overline{f^a}\}$ in the concepts environment C and extract its refined concepts $\overline{\mathit{cid}}$. The sequence $\overline{\mathit{cid}}$ actually names concept identifiers, i.e., a refinement clause refers to concept *uses*. In the current version of the semantics, $\overline{\mathit{cid}}$ is a sequence of concept names. Again, error handling is not explicit and the judgment does not apply if there is no concept named *con*.

overload resolution

$$\boxed{\mathit{overload\text{-}res}\,(f\,\overline{\tau}, \overline{\mathit{sig}}) = \mathit{sig}?_{\mathit{amb}}}$$

OVERLOAD_AMB

$$\frac{1.\ \mathit{distinct}\,(\overline{\mathit{sig}}) \qquad 2.\ \overline{\mathit{sig}} \subseteq \overline{\mathit{sig}}\,\mathit{matching}\,f\,\overline{\tau} \qquad 3.\ \mathit{length}\,\overline{\mathit{sig}} >= 2}{\mathit{overload\text{-}res}\,(f\,\overline{\tau}, \overline{\mathit{sig}}) = ¿?}$$

Given a list of function signatures $\overline{\mathit{sig}}$ and a function call $f\,\overline{\tau}$ (f is a function name and $\overline{\tau}$ is a list of argument types) decide which of the function signatures in $\overline{\mathit{sig}}$ to use.

This rule (OVERLOAD_AMB) handles the ambiguous cases. First, ensure that there are no duplicate function signatures. Second, find all signatures that match the function call. Third, assert that there were at least two signatures that match; therefore, overload resolution concludes with ambiguity.

Note that *matching* is currently not specified in our semantics; it is just a placeholder for the appropriate matching strategy.

OVERLOAD_FOUND

$$\frac{1.\ \mathit{distinct}\,(\overline{\mathit{sig}}) \quad 2.\ \overline{\mathit{sig}} \subseteq \overline{\mathit{sig}}\,\mathit{matching} f\,\overline{\tau} \quad 3.\ \mathit{length}\,\overline{\mathit{sig}} = 1 \quad 4.\ \mathit{sig} \in \overline{\mathit{sig}}}{\mathit{overload\text{-}res}\,(f\,\overline{\tau}, \overline{\mathit{sig}}) = \mathit{sig}}$$

This rule (OVERLOAD_FOUND) handles the case where only one of the signatures in $\overline{sig}$ matches the call $f\,\overline{\tau}$. In this case, the matching function signature ("some" option) is returned as the result of overload resolution.

OVERLOAD_NOT_FOUND

$$\frac{1.\ \mathit{distinct}\,(\overline{\mathit{sig}}) \quad 2.\ \overline{\mathit{sig}} \subseteq \overline{\mathit{sig}}\,\mathit{matching} f\,\overline{\tau} \quad 3.\ \mathit{length}\,\overline{\mathit{sig}} = 0}{\mathit{overload\text{-}res}\,(f\,\overline{\tau}, \overline{\mathit{sig}}) = \mathit{None}}$$

This rule handles the case where no function signatures match the function call. The "*None*" option is returned as the result of overload resolution.

4 Type System

direct concept refinement

$$\boxed{C \vdash \mathit{cid} \prec_1 \overline{\mathit{cid}}}$$

REFM_DIR

$$\frac{1.\ \mathit{refined}\,(C, \mathit{cid}) = \overline{\mathit{cid}}}{C \vdash \mathit{cid} \prec_1 \overline{\mathit{cid}}}$$

A concept designated by a concept identifier *cid directly* refines concepts designated by concept identifiers $\overline{cid}$, in the context of concept environment C. To find $\overline{cid}$, the helper relation *refined* is applied.

concept refinement

$$\boxed{C \vdash cid \prec \overline{cid}}$$

CIDREF_REC

$$\frac{1.\, C \vdash cid \prec_1 \overline{cid_i}^{\,i} \qquad 2.\, \overline{C \vdash cid_i \prec \overline{cid}_i}^{\,i}}{C \vdash cid \prec \overline{cid_i \overline{cid}_i}^{\,i}}$$

The refinement relation is a transitive closure of the direct refinement relation. In the first premise, the sequence of concept identifiers for concepts directly refined by *cid* is found. Then, in premise 2, the refinement relation is applied recursively to every directly refined concept. The recursion terminates when a base case of a *cid* without direct refinements is reached. The result is a sequence containing every directly refined concept identifier cid_i and the concept identifiers $\overline{cid}_i$ that it refines.

concept refinement (for concept names)

$$\boxed{C \vdash con \prec \overline{cid}}$$

CONREF_REC

$$\frac{1.\, C \vdash cid \prec \overline{cid} \qquad 2.\, con\,(cid) = con}{C \vdash con \prec \overline{cid}}$$

The previous refinement judgments define refinement between concept *uses*, i.e., concept identifiers. Refinement also occurs between a concept and concept identifiers, when a concept is defined. This judgment defines the refinement relation between a concept name *con*, which refers to a concept definition, and concept identifiers $\overline{cid}$, which refer to concept uses in a refinement clause. The judgment finds an appropriate concept identifier *cid* (premise 2) and forwards the job to refinement between concept identifiers (premise 1).

type lookup, non-recursive

$$\boxed{\mathit{find\text{-}scope}\,(C, cid, \tau) = \tau?}$$

FT_BINTYPE

$$\frac{}{\mathit{find\text{-}scope}\,(C, cid, bintype) = bintype}$$

Built-in types are always in scope.

FT_QUALIFIED

$$\frac{1.\, con\,(cid) = con \qquad 2.\, \mathit{find\text{-}concept}\,(C, con) = cp \qquad 3.\, \overline{ty^a}\,(cp) = \overline{ty^a} \qquad 4.\, id_\tau \in \overline{id_\tau}\,(\overline{ty^a})}{\mathit{find\text{-}scope}\,(C, cid, cid :: id_\tau) = cid :: id_\tau}$$

This rule describes successful qualified name lookup. The type identifier id_τ qualified with *cid* is looked up in the scope of *cid* and in the environment C. The associated types of *cid* are extracted and if one of the associated types is named id_τ then it is returned as the result of the lookup. Note that the concept in which the type is looked up is the same as the concept with which the looked up name is qualified (both are *cid*).

FT_UNQUALIFIED

$$\frac{1.\ con(cid) = con \quad 2.\ \textit{find-concept}(C, con) = cp \quad 3.\ \overline{ty^a}(cp) = \overline{ty^a} \quad 4.\ id_\tau \in \overline{id_\tau}(\overline{ty^a})}{\textit{find-scope}(C, cid, id_\tau) = cid :: id_\tau}$$

This rule describes successful unqualified name lookup. The associated type named id_τ is looked up in the concept identified by *cid*.

FT_NONE_SCOPE

$$\frac{1.\ cid \neq cid'}{\textit{find-scope}(C, cid, cid' :: id_\tau) = None}$$

This rule describes unsuccessful qualified name lookup. The type named id_τ, qualified by the concept identifier cid', is looked up in the concept identified by *cid*. The concept identifier cid' used to qualify id_τ does not match the identifier *cid* of the concept in which the lookup is being performed (premise 1) and the lookup returns *None* to indicate failure.

FT_NONE_QUALIFIED

$$\frac{1.\ con(cid) = con \quad 2.\ \textit{find-concept}(C, con) = cp \quad 3.\ \overline{ty^a}(cp) = \overline{ty^a} \quad 4.\ id_\tau \notin \overline{id_\tau}(\overline{ty^a})}{\textit{find-scope}(C, cid, cid :: id_\tau) = None}$$

This rule describes unsuccessful qualified name lookup. Here the qualifying scope and the concept in which the lookup is performed are the same (*cid*) but there is no associated type named id_τ (premise 4).

FT_NONE

$$\frac{1.\ con(cid) = con \quad 2.\ \textit{find-concept}(C, con) = cp \quad 3.\ \overline{ty^a}(cp) = \overline{ty^a} \quad 4.\ id_\tau \notin \overline{id_\tau}(\overline{ty^a})}{\textit{find-scope}(C, cid, id_\tau) = None}$$

This rule describes unsuccessful unqualified name lookup. Similarly to the previous rule, there is no associated type named id_τ.

recursive type lookup

$$\boxed{\mathit{find\text{-}rec}\,(C, \mathit{cid}, \tau) = \tau?}$$

FTREC_DIRECT

$$\frac{1.\ \mathit{find\text{-}scope}\,(C, \mathit{cid}, \mathit{id}_\tau) = \tau}{\mathit{find\text{-}rec}\,(C, \mathit{cid}, \mathit{id}_\tau) = \tau}$$

If the associated type is found in the current scope it is returned without looking any further.

FTREC_REC_SOME

$$\frac{\begin{array}{l}1.\ \mathit{find\text{-}scope}\,(C, \mathit{cid}, \mathit{id}_\tau) = \mathit{None} \\ 2.\ C \vdash \mathit{cid} \prec_1 \overline{\mathit{cid}_i}^{\,i} \\ 3.\ \overline{\mathit{find\text{-}rec}\,(C, \mathit{cid}_i, \mathit{id}_\tau) = \tau?_i}^{\,i} \\ 4.\ \mathit{rm?}\,(\overline{\tau?_i}^{\,i}) = \tau\,\overline{\tau}\end{array}}{\mathit{find\text{-}rec}\,(C, \mathit{cid}, \mathit{id}_\tau) = \tau}$$

If the type named id_τ is not found in the current scope (premise 1) it is searched for in the directly refined concepts, which are extracted in premise . The search is performed recursively in the refined concept instances (premise 3) and empty options from the results are removed (premise 4). Since refinement hierarchy is traversed depth-first, the type returned by the lookup is the one found in depth-first search. Note that type specifiers may repeat in the order in which they were reached from the current scope. Any shadowed identifiers will not be found because the search is terminated as soon as we find something.

FTREC_REC_NONE

$$\frac{\begin{array}{l}1.\ \mathit{find\text{-}scope}\,(C, \mathit{cid}, \mathit{id}_\tau) = \mathit{None} \\ 2.\ C \vdash \mathit{cid} \prec_1 \overline{\mathit{cid}_i}^{\,i} \\ 3.\ \overline{\mathit{find\text{-}rec}\,(C, \mathit{cid}_i, \mathit{id}_\tau) = \tau?_i}^{\,i} \\ 4.\ \mathit{rm?}\,(\overline{\tau?_i}^{\,i}) =\end{array}}{\mathit{find\text{-}rec}\,(C, \mathit{cid}, \mathit{id}_\tau) = \mathit{None}}$$

Same as above but treats the case where no associated types named id_τ were found.

FTREC_REC_SCOPE_NONE

$$\frac{\begin{array}{l}1.\ \mathit{find\text{-}rec}\,(C, \mathit{cid}', \mathit{id}_\tau) = \tau? \\ 2.\ \tau? = \mathit{None} \\ 3.\ C \vdash \mathit{cid} \prec \overline{\mathit{cid}_i}^{\,i} \\ 4.\ \mathit{cid}' \in \overline{\mathit{cid}_i}^{\,i} \vee \mathit{cid}' = \mathit{cid}\end{array}}{\mathit{find\text{-}rec}\,(C, \mathit{cid}, \mathit{cid}' :: \mathit{id}_\tau) = \mathit{None}}$$

FTREC_REC_SCOPE_SOME

$$\frac{1.\ \mathit{find\text{-}rec}(C, \mathit{cid}', \mathit{id}_\tau) = \tau? \quad 2.\ \tau? = \tau \quad 3.\ \mathit{find\text{-}rec}(C, \mathit{cid}, \tau) = \tau' \quad 4.\ C \vdash \mathit{cid} \prec \overline{\mathit{cid}_i}^{\,i} \quad 5.\ \mathit{cid}' \in \overline{\mathit{cid}_i}^{\,i} \vee \mathit{cid}' = \mathit{cid}}{\mathit{find\text{-}rec}(C, \mathit{cid}, \mathit{cid}' :: \mathit{id}_\tau) = \tau'}$$

The search for a qualified identifier is simply performed in the qualifying scope. Note that the scope must be either the current scope or one of the refined scopes. That requirements does not seem to be present in the concept wording (Gregor et al., 2008e) although it may be implicit in clause 3.4.3.3 para. 1 because if one refers to a concept map in a constrained context there must be a corresponding concept map archetype. The lookup in scope is split depending on whether the lookup is successful or not. In the second case of successfull lookup, the discovered type specifier is looked up again in the context of the concept in which it was looked up. This is done to assure that a depth-first search representative of an equivalence class of associated types is returned and this class is determined by *cid*.

FTREC_REC_SCOPE

$$\frac{1.\ \mathit{find\text{-}rec}(C, \mathit{cid}', \mathit{id}_\tau) = \tau? \quad 2.\ C \vdash \mathit{cid} \prec \overline{\mathit{cid}_i}^{\,i} \quad 3.\ \neg(\mathit{cid}' \in \overline{\mathit{cid}_i}^{\,i} \vee \mathit{cid}' = \mathit{cid})}{\mathit{find\text{-}rec}(C, \mathit{cid}, \mathit{cid}' :: \mathit{id}_\tau) = \mathit{None}}$$

This rule complements the previous rule and returns *None* if the qualifier of the qualified type that is being looked up was not one of the refined concepts. This rule prevents type lookup from getting stuck and instead returns the *None* option. An alternative would be to return a special error value but for our purposes name lookup failure will prevent unwanted cases from passing concept check.

non-recursive associated function lookup

$$\boxed{\mathit{find\text{-}scope}(C, \mathit{cid}, f) = \overline{\mathit{sig}}}$$

FIND_FUN_SCOPE

$$\frac{1.\ \mathit{con}(\mathit{cid}) = \mathit{con} \quad 2.\ \mathit{find\text{-}concept}(C, \mathit{con}) = \mathit{cp} \quad 3.\ \overline{f^a}(\mathit{cp}) = \overline{f^a} \quad 4.\ \mathit{overload\text{-}set}(f, \overline{f^a}) = \overline{\mathit{sig}} \quad 5.\ \mathit{normalize}(C, \mathit{cid}, \overline{\mathit{sig}}) = \overline{\mathit{sig}}'}{\mathit{find\text{-}scope}(C, \mathit{cid}, f) = \overline{\mathit{sig}}'}$$

Given the concept environment C, a concept identifier *cid*, and a function name f, look up the set of functions named f in the concept identified by *cid*. First, a concept identified by *cid* is looked up (premise 1 and 2). The case where *cid* does not identify

a defined concept, i.e., *find-concept* returns *None*, is not handled. Next, the associated functions $\overline{f^a}$ of the concept looked up in premise 2 are extracted. Then, the function signatures for functions named f are extracted into $\overline{sig}$. In premise 5, the signatures in $\overline{sig}$ are normalized, i.e., all type names are fully qualified. Finally, the normalized signatures $\overline{sig}'$ are returned.

generate overload set

$$\boxed{\mathit{overload\text{-}set}\,(f, \overline{f^a}) = \overline{sig}}$$

G_OVER_EMPTY

$$\frac{}{\mathit{overload\text{-}set}\,(f, \ ) =}$$

An empty set of associated functions produces an empty overload set.

G_OVER_REC

$$\frac{1.\ \overline{f^a} = f^a\,\overline{f^a}' \quad 2.\ \mathit{fname}\,(f^a) = f \quad 3.\ \mathit{overload\text{-}set}\,(f, \overline{f^a}') = \overline{sig}}{\mathit{overload\text{-}set}\,(f, \overline{f^a}) = f^a\,\overline{sig}}$$

Examine each associated function in $\overline{f^a}$ by "calling" *overload-set* recursively. For every signature, check if the function is named f and if it is, add it to the result of the recursive overload set lookup.

fully qualify type names in an overload set

$$\boxed{\mathit{normalize}\,(C, cid, \overline{sig}) = \overline{sig}'}$$

N_FDECLS_EMPTY

$$\frac{}{\mathit{normalize}\,(C, cid, \ ) =}$$

Do not do anything for an empty overload set.

N_FDECLS_REC

$$\frac{\begin{array}{l}1.\ \overline{sig} \approx f^a\,\overline{sig}' \quad 2.\ \mathit{fname}\,(f^a) = f \\ 3.\ \mathit{returns}\,(f^a) = \tau \qquad 4.\ \mathit{params}\,(f^a) = \overline{\tau} \\ 5.\ \mathit{find\text{-}rec}\,(C, cid, \tau) = \tau' \\ 6.\ \mathit{find\text{-}rec}\,(C, cid, \tau_1) = \tau_1' \ \ .. \ \ \mathit{find\text{-}rec}\,(C, cid, \tau_n) = \tau_n' \\ 7.\ \mathit{normalize}\,(C, cid, \overline{sig}') = \overline{sig}'' \\ 8.\ \mathit{sig}\,(f, \tau', \tau_1' .. \tau_n') = f^{a\prime}\end{array}}{\mathit{normalize}\,(C, cid, \overline{sig}) = f^{a\prime}\,\overline{sig}''}$$

Given a concept environment C, a concept identifier *cid*, and an overload set $\overline{sig}$, fully

qualify type names in each signature in $\overline{sig}$. A single signature is first extracted, in premise 1, from the overload set. Then it is disassembled into parts in premises 2–4. In premises 5–6, each type occurring in the signature is looked up in the concept identified by *cid*. In premises 7–8, the normalization process is called recursively for the remaining signatures and the current signature is reassembled, with the fully qualified types looked up earlier. The final result is the reassembled signature prepended to the result of the recursive normalization process.

fully qualify type names in an expression

$$\boxed{normalize\,(C, cid, e) = e'}$$

N_EXPR_VAR

$$\frac{}{normalize\,(C, cid, var) = var}$$

N_EXPR_OBJ

$$\frac{1.\ find\text{-}rec\,(C, cid, \tau) = \tau'}{normalize\,(C, cid, \texttt{obj}\,\tau) = \texttt{obj}\,\tau'}$$

N_EXPR_APP

$$\frac{1.\ \overline{normalize\,(C, cid, e_i) = e_i'}^{\,i}}{normalize\,(C, cid, f\,(\overline{e_i}^{\,i})) = f\,(\overline{e_i'}^{\,i})}$$

N_EXPR_INT

$$\frac{}{normalize\,(C, cid, v_{int}) = v_{int}}$$

N_EXPR_BOOL

$$\frac{}{normalize\,(C, cid, v_{bool}) = v_{bool}}$$

Normalization but for expressions.

fully qualify type names in an overload set

$$\boxed{normalize\,(C, cid, \overline{fdef}) = \overline{fdef}'}$$

N_FDEFS_EMPTY

$$\frac{}{normalize\,(C, cid, \ ) = }$$

Do not do anything for an empty function definitions set.

N_FDEFS_REC

$$\frac{\begin{array}{ll}1.\ \overline{fdef} = fdef\,\overline{fdef}' & 2.\ fdef = \tau''\,f\,(\overline{var_i\colon \tau_i''}^{\,i})\,\{e\} \\ 3.\ fname\,(sig\,(fdef)) = f & 4.\ returns\,(sig\,(fdef)) = \tau \\ 5.\ params\,(sig\,(fdef)) = \overline{\tau} & \\ 6.\ find\text{-}rec\,(C, cid, \tau) = \tau' & \\ 7.\ find\text{-}rec\,(C, cid, \tau_1) = \tau_1' \quad .. \quad find\text{-}rec\,(C, cid, \tau_n) = \tau_n' & \\ 8.\ normalize\,(C, cid, e) = e' & \\ 9.\ normalize\,(C, cid, \overline{fdef}') = \overline{fdef}'' & \\ 10.\ sig\,(f, \tau', \tau_1' .. \tau_n') = f^{a\prime} & \\ 11.\ fdef\,(f^{a\prime}, \overline{var_i}^{\,i}, e') = fdef' & \end{array}}{normalize\,(C, cid, \overline{fdef}) = fdef'\,\overline{fdef}''}$$

Normalization of function definitions, as for function declarations but with normalization of expressions.

compute overload set recursively

$$\boxed{find\text{-}rec\,(C, cid, f) = \overline{sig}}$$

FIND_FUN_REC

$$\frac{\begin{array}{ll}1.\ C \vdash cid \prec \overline{cid_i}^{\,i} & \\ 2.\ find\text{-}scope\,(C, cid, f) = \overline{sig} & \\ 3.\ \overline{find\text{-}scope\,(C, cid_i, f) = \overline{sig_i}}^{\,i} & \\ 4.\ \overline{sig}' = \overline{sig}\,\overline{\overline{sig_i}}^{\,i} & \\ 5.\ normalize\,(C, cid, \overline{sig}') = \overline{sig}'' & \\ 6.\ rdup\,(\overline{sig}'') = \overline{sig}''' & \end{array}}{find\text{-}rec\,(C, cid, f) = \overline{sig}'''}$$

Given the concept environment C and the concept identifier *cid*, look up the overload set for a function named f in the concept identified by *cid* and the concepts it refines. The set of concept identifers refined by *cid* is found in premise 1 and, in premises 2–3, the overload set for function f is looked up for each of these concepts. In premise 4, all of these overload sets are concatenated. Then, in premise 5, function signatures must be normalized again (they were first normalized when found in scope) to make sure that identical types are identified as such. In premise 6, duplicates (if there are any) are removed. The sequence of concept identifiers found in premise 1 maintains the depth-first order of discovery in traversal of refinement clauses. Furthermore, the *rdup* function in premise 6 keeps the first occurrence of each duplicate. Consequently, when there are duplicate signatures, the ones found first by depth-first traversal of the refinement hierarchy are kept.

constrained context typing

$$\boxed{C;\Gamma;\overline{cid} \vdash e : \tau}$$

T_INT

$$\frac{}{C;\Gamma;\overline{cid} \vdash v_{int} : \texttt{int}}$$

T_BOOL

$$\frac{}{C;\Gamma;\overline{cid} \vdash v_{bool} : \texttt{bool}}$$

Values of built in types have the corresponding built-in type.

T_VAR

$$\frac{1.\ \Gamma(var) = \tau}{C;\Gamma;\overline{cid} \vdash var : \tau}$$

Types of variables are looked up in the variable typing environment Γ.

T_OBJ

$$\frac{\begin{array}{l}1.\ \overline{\mathit{find\text{-}rec}(C, cid_i, \tau') = \tau?_i}^{\,i} \\ 2.\ \mathit{rm?}(\overline{\tau?_i}^{\,i}) = \overline{\tau} \\ 3.\ \overline{\tau} \approx \tau\end{array}}{C;\Gamma;\overline{cid_i}^{\,i} \vdash \texttt{obj}\,\tau' : \tau}$$

This judgment gives the type of an object created by an object creation expression $\texttt{obj}\,\tau'$. To type the object, the type τ' must be looked up in the constraints $\overline{cid_i}^{\,i}$. In premise 1, the type τ' is looked up in each of the requirements and, in premise 2, the *None* results of unsuccessful searches are removed. Finally, the third premise asserts that the sequence of results $\overline{\tau}$ contains only one type τ (but possibly more than once). This type is the type of the object created by the object creation expression.

T_APP

$$\frac{\begin{array}{l}1.\ \overline{\mathit{find\text{-}rec}(C, cid_i, f) = \overline{sig_i}}^{\,i} \\ 2.\ \mathit{distinct}(\overline{\overline{sig_i}}^{\,i}) \\ 3.\ \overline{C;\Gamma;\overline{cid_i}^{\,i} \vdash e_k : \tau_k}^{\,k} \\ 4.\ \mathit{overload\text{-}res}(f\,\overline{\tau_k}^{\,k}, \overline{\overline{sig_i}}^{\,i}) = sig \\ 5.\ \mathit{returns}(sig) = \tau\end{array}}{C;\Gamma;\overline{cid_i}^{\,i} \vdash f(\overline{e_k}^{\,k}) : \tau}$$

This judgment types a function call $f(\overline{e_k}^{\,k})$. First, in premise 1, an overload set with functions named f is looked up; the second premise ensures that there are no duplicates in the set. In the third premise every argument is typed and overload resolution is performed in premise 4, finding the appropriate function signature *sig*. The return type extracted from that signature is the type of the function call. Note that if overload resolution is ambiguous or fails a function application cannot be typed.

type definition well-formedness

$\boxed{tydef\ \surd}$

TD_WFBINTYPE

$$\frac{}{\texttt{typedef}\ bintype\ id_\tau\ \surd}$$

The core language does currently not allow user-defined types. Therefore, the only types that can be used in a type declaration are built-in types.

required associated functions

$\boxed{\overline{f^a}\,(C, cid) = \overline{f^a}}$

ARF_REF

$$\frac{\begin{array}{l}1.\ C \vdash cid \prec \overline{cid_i}^{\,i} \\ 2.\ \overline{\overline{f^a}\,(C, cid_i) = \overline{f^a}_i}^{\,i} \\ 3.\ \overline{normalize\,(C, cid_i, \overline{f^a}_i) = \overline{f^a}'_i}^{\,i} \\ 4.\ \overline{normalize\,(C, cid, \overline{f^a}'_i) = \overline{f^a}''_i}^{\,i} \\ 5.\ \overline{f^a}\,(C, cid) = \overline{f^a} \\ 6.\ normalize\,(C, cid, \overline{f^a}) = \overline{f^a}'\end{array}}{\overline{f^a}\,(C, cid) = \overline{f^a}'\,\overline{\overline{f^a}_i}^{\,i}}$$

This judgment shows how associated functions are extracted from a concept identified by *cid* and from all the concepts that *cid* refines. In the first 4 premises, associated functions are extracted from refined concepts cid_i and normalized, first in the context in which they were found and then in the context of *cid*. Then the list of associated functions of *cid* is extracted and normalized. Notice that a list of function declarations is returned and not a set. There may be duplicates but they cause no problems since name lookup will remove them.

required associated types

$\boxed{\overline{ty^a}\,(C, cid) = \overline{ty^a}}$

ARA_REF

$$\frac{\begin{array}{l}1.\ C \vdash cid \prec \overline{cid_i}^{\,i} \\ 2.\ \overline{\overline{ty^a}\,(C, cid_i) = \overline{ty^a}_i}^{\,i} \\ 3.\ \overline{ty^a}\,(C, cid) = \overline{ty^a}\end{array}}{\overline{ty^a}\,(C, cid) = \overline{ty^a}\,\overline{\overline{ty^a}_i}^{\,i}}$$

This judgment shows how required associated types are extracted for a concept. Note that all types are returned without qualification. That is, if a name is shadowed it will occur more than once in the result but it will be impossible to detect any difference between the occurrences.

defined function signatures (for concept maps)

$$\boxed{\overline{sig}\,(C, M, cid?, cid) = \overline{sig}}$$

DFDECLS_NONE

$$\frac{\begin{array}{ll}1.\ C \vdash cid \prec \overline{cid_i}^{\,i} & 2.\ cid? = None \\ 3.\ \overline{cm\,(M, cid_i) = cm?_i}^{\,i} & \\ 4.\ rm?\,(\overline{cm?_i}^{\,i}) = \overline{cm_j}^{\,j} & \\ 5.\ \overline{\overline{fdef}\,(cm_j) = \overline{fdef}_j}^{\,j} & \\ 6.\ \overline{normalize\,(C, cid\,(cm_j), \overline{sig}\,(\overline{fdef}_j)) = \overline{sig}_j}^{\,j} & \\ 7.\ normalize\,(C, cid, \overline{\overline{sig}_j}^{\,j}) = \overline{sig} & \end{array}}{\overline{sig}\,(C, M, cid?, cid) = \overline{sig}}$$

This judgment extracts function signatures from concept maps for concept instances refining *cid*. The first rule handles the concept maps that have not been implicitly generated (premise 2). In premises 3–5, the concept maps for concepts refined by *cid* are looked up, and their function definitions are extracted. In premise 6, a signature of each definition is extracted and then normalized. Fnially, signatures are normalized in the context of *cid* to make type naming consistent.

DFDECLS_SOME

$$\frac{\begin{array}{ll}1.\ C \vdash cid \prec \overline{cid_i}^{\,i} & 2.\ cid? = cid_{impl} \\ 3.\ \overline{cm\,(M, cid_i) = icm?_i}^{\,i} & \\ 4.\ rm?\,(\overline{icm?_i}^{\,i}) = \overline{icm} & \\ 5.\ \overline{icm_k}^{\,k} = \{\, icm \in \overline{icm} \mid icm\,.\,cid \neq cid_{impl} \,\} & \\ 6.\ \overline{\overline{fdef}\,(icm_k\,.\,cm) = \overline{fdef}_k}^{\,k} & \\ 7.\ \overline{normalize\,(C, cid\,(cm_k), \overline{sig}\,(\overline{fdef}_k)) = \overline{sig}_k}^{\,k} & \\ 8.\ normalize\,(C, cid, \overline{\overline{sig}_k}^{\,k}) = \overline{sig} & \end{array}}{\overline{sig}\,(C, M, cid?, cid) = \overline{\overline{sig}_k}^{\,k}}$$

The second rule is similar to the first but it handles the concept maps that have been implicitly defined (premise 2). In these cases, the concept maps for refined concepts that have been implicitly defined by the same source are ignored. In the third premise, implicit concept maps are searched for (the result of each search is an *icm*?); in the first rule, the implicit "parent" part of implicit concept maps was ignored (the result of each search was a *cm*?). The fifth premise in the second rule filters out all concept maps for refined concepts that have been implicitly defined by the same parent ($icm\,.\,cid \neq cid_{impl}$). This is done to ignore functions in refined concept maps that have exactly the same definitions since the definitions came from the same source: the "parent" concept map that caused implicit definition.

defined functions (for concept maps)

$$\boxed{\overline{fdef}\,(C, M, cid?, cid) = \overline{fdef}}$$

DFDEFS_NONE

$$\frac{\begin{array}{l}1.\ C \vdash cid \prec \overline{cid_i}^{\,i} \\ 2.\ cid? = None \\ 3.\ \overline{cm\,(M, cid_i) = cm?_i}^{\,i} \\ 4.\ rm?\,(\overline{cm?_i}^{\,i}) = \overline{cm_j}^{\,j} \\ 5.\ \overline{\overline{fdef}\,(cm_j) = \overline{fdef}_j}^{\,j} \\ 6.\ \overline{normalize\,(C, cid\,(cm_j), \overline{fdef}_j) = \overline{fdef}_j}^{\,j} \\ 7.\ \overline{normalize\,(C, cid, \overline{fdef}'_j) = \overline{fdef}''_j}^{\,j}\end{array}}{\overline{fdef}\,(C, M, cid?, cid) = \overline{\overline{fdef}''_j}^{\,j}}$$

DFDEFS_SOME

$$\frac{\begin{array}{l}1.\ C \vdash cid \prec \overline{cid_i}^{\,i} \quad 2.\ cid? = cid_{impl} \\ 3.\ \overline{cm\,(M, cid_i) = icm?_i}^{\,i} \\ 4.\ rm?\,(\overline{icm?_i}^{\,i}) = \overline{icm} \\ 5.\ \overline{icm_k}^{\,k} = \{\, icm \in \overline{icm} \mid icm\,.\,cid \neq cid_{impl} \,\} \\ 6.\ \overline{\overline{fdef}\,(icm_k\,.\,cm) = \overline{fdef}_k}^{\,k} \\ 7.\ \overline{normalize\,(C, cid\,(icm_k\,.\,cm), \overline{fdef}_k) = \overline{fdef}'_j}^{\,k} \\ 8.\ \overline{normalize\,(C, cid, \overline{fdef}'_k) = \overline{fdef}''_k}^{\,k}\end{array}}{\overline{fdef}\,(C, M, cid?, cid) = \overline{\overline{fdef}''_k}^{\,k}}$$

This judgment is very similar to the previous one but function definitions instead of signatures are extracted.

defined types (for concept maps)

$$\boxed{\overline{tydef}\,(M, \overline{cid}) = \overline{tydef}}$$

DEFASSTS

$$\frac{\begin{array}{l}1.\ \overline{cm\,(M, cid_i) = cm?_i}^{\,i} \\ 2.\ rm?\,(\overline{cm?_i}^{\,i}) = \overline{cm_k}^{\,k} \\ 3.\ \overline{\overline{tydef}\,(cm_k) = \overline{tydef}_k}^{\,k}\end{array}}{\overline{tydef}\,(M, \overline{cid_i}^{\,i}) = \overline{\overline{tydef}_k}^{\,k}}$$

This judgment extracts definitions of associated types in concept maps for concepts identified by $\overline{cid_i}^{\,i}$. In contrast to the previous two judgments for associated function definitions, the concepts $\overline{cid_i}^{\,i}$ for which definitions should be extracted are provided directly instead of a concept that they refine. Also, since well-formed type definitions must define a type identifier to a built-in type no normalization is required as in the

similar associated functions loookup.

one-level type definitions compatibility

$$\boxed{\overline{ty^a} \vdash \overline{tydef} \eqsim \overline{tydef}^{\prec_1}}$$

ASSTS_COMPAT

$$\frac{1.\ \overline{tydef}' = \{\, tydef \in \overline{tydef} \mid \neg\, id_\tau\,(tydef) \in \overline{id_\tau}\,(\overline{ty^a})\,\} \qquad 2.\ \forall\, tydef \in \overline{tydef}' \bullet \mathit{find\text{-}defs}\,(id_\tau\,(tydef), \overline{tydef}^{\prec_1}) \approx tydef}{\overline{ty^a} \vdash \overline{tydef} \eqsim \overline{tydef}^{\prec_1}}$$

Compatibility of type declarations is decided in light of the associated types of the concept for which the concept map is defined. The rule consists of two parts:

1. Narrow the set of type declarations to those that are not for one of the associated types.
2. Make sure that the type declarations provided for a new concept map are compatible with the existing declarations.

The compatibility of "siblings," i.e., type declarations in $\overline{tydef}^{\prec_1}$ does not have to be checked separately. For non-shadowed type names, line 2 ensures sibling compatibility through the new definition in $\overline{tydef}$. For shadowed names, compatibility was already ensured when these names were defined. In particular, definitions for shadowed names may differ if the sources of a name in the refinement hierarchy are different.

type definitions check

$$\boxed{\mathit{tydefs\text{-}check}\,(C, cid, \overline{tydef})}$$

TYPEDECLS_CHECK

$$\frac{1.\ \overline{ty^a}\,(C, cid) = \overline{ty^a}^{\prec} \qquad 2.\ \overline{id_\tau}\,(\overline{tydef}) \approx \overline{id_\tau}\,(\overline{ty^a}^{\prec}) \qquad 3.\ \forall\, tydef \in \overline{tydef} \bullet tydef\ \surd \qquad 4.\ \mathit{distinct}\,(\overline{id_\tau}\,(\overline{tydef}))}{\mathit{tydefs\text{-}check}\,(C, cid, \overline{tydef})}$$

This relation is a helper relation used in the P_INST_CMAP rule. When processing a concept map, it must be checked that there is a type declaration corresponding to every associated type in the current concept instance (represented by *cid*). Also, there cannot be more than one type declaration for each associated type (the distinctiveness check) and every type declaration must be well-formed.

type definitions compatibility

$$\boxed{tydefs_{=\!=}(C, M, cid, \overline{tydef})}$$

TYPEDECLS_COMPAT

$$\frac{\begin{array}{l}1.\ \overline{ty^a}\,(C, cid) = \overline{ty^a} \\ 2.\ C \vdash cid \prec_1 \overline{cid}^{\prec_1} \\ 3.\ \overline{tydef}\,(M, \overline{cid}^{\prec_1}) = \overline{tydef}^{\prec_1} \\ 4.\ \overline{ty^a} \vdash \overline{tydef} =\!= \overline{tydef}^{\prec_1}\end{array}}{tydefs_{=\!=}(C, M, cid, \overline{tydef})}$$

This judgment is a helper judgment used in the P_INST_CMAP rule. First (premise 1), the associated types required by the concept instance represented by *cid* are extracted; no associated type requirements from refined concepts are included. Next (premise 2), concepts instances directly refined by *cid* are extracted into $\overline{cid}^{\prec_1}$. Then (premise 3), all type definitions provided by concept maps for concept identifiers in $\overline{cid}^{\prec_1}$ are extracted into $\overline{tydef}^{\prec_1}$. It is important to remember that each concept map must provide type definitions for all associated type requirements reached through refinement. Since type definitions in $\overline{tydef}^{\prec_1}$ were extracted from existing concept maps, these were already checked for compatibility conflicts and inconsistencies. For concept instances in $\overline{cid}^{\prec_1}$ for which concept maps do not exist yet, implicit concept maps will be created, assuming that the currently processed definition passes compatibility and consistency checks. The implicit concept maps will propagate the definitions from the currently processed map and if an incompatibility is introduced, it will be detected later as implicit concept maps are created. Finally (premise 4), the one-level compatibility check is performed between type definitions in $\overline{tydef}$ and in $\overline{tydef}^{\prec_1}$.

function definitions check

$$\boxed{fdefs\text{-}check\,(C, cid, \overline{fdef})}$$

FDEFS_CHECK

$$\frac{\begin{array}{l}1.\ \forall \tau f\,(\overline{var_i\colon \tau_i}^{\,i})\,\{\,e\,\} \in \overline{fdef} \bullet C\,;\,[\overline{var_i\colon \tau_i}^{\,i}]\,;\,cid \vdash e\colon \tau \\ 2.\ distinct\,(\overline{sig}\,(\overline{fdef}))\end{array}}{fdefs\text{-}check\,(C, cid, \overline{fdef})}$$

This judgment is a helper judgment used in the P_INST_CMAP rule. First (premise 1), all function definitions are type checked. After the definitions are type checked, a distinctiveness check is performed to ensure that there are no conflicting definitions. Which definitions should be provided is checked in the FDEFS_COMPATIBILITY rule.

function definitions compatibility

$$\boxed{fdefs_{=}(C, M, cid_{cmap}, cid?, \overline{fdef})}$$

FDEFS_COMPAT

$$\frac{\begin{array}{l}1.\ \overline{f^a}(C, cid) = \overline{f^a}^{\prec} \\ 2.\ \overline{sig}(C, M, cid?, cid) = \overline{sig}^{\prec} \\ 3.\ distinct(\overline{sig}^{\prec}) \\ 4.\ normalize(C, cid, \overline{sig}(\overline{fdef})) = \overline{sig} \\ 5.\ \overline{sig} \approx \overline{f^a}^{\prec} \setminus \overline{sig}^{\prec}\end{array}}{fdefs_{=}(C, M, cid, cid?, \overline{fdef})}$$

This judgment is a helper judgment used in the P_INST_CMAP rule. First (premise 1), all required associated functions are collected; both for the concept instance represented by *cid* and for any concept instance that *cid* refines. Next (premise 2), all function definitions for concept maps of the refined concepts are collected. Then (premise 3), a distinctiveness check is performed on the function definitions from the concept maps for the refined concepts. Finally (premise 5), there must be a function definition for every associated function except those that are currently defined. Premise 4 normalizes the set of declarations representing the implementations in the current concept map to avoid name capture later on. Note that both $\overline{f^a}$ and $\overline{sig}$ (premises 1 and 2) already normalize returned signatures properly.

generate implicit concept maps

$$\boxed{icms(C, M, cid, cid?, \overline{tydef}, \overline{fdef}) = \overline{icm}'}$$

IMPLICIT_CMAPS_SOME

$$\frac{\begin{array}{ll}1.\ cid? = cid_{impl} & 2.\ C \vdash cid \prec_1 \overline{cid}^{\prec_1} \\ 3.\ \overline{cid_i}^{\,i} = \{\, cid \in \overline{cid}^{\prec_1} \mid cm(M, cid) = None \,\} & \\ 4.\ \overline{fdef}(C, M, cid_{impl}, cid) = \overline{fdef}^{\prec} & \\ 5.\ \overline{\overline{ty^a}(C, cid_i) = \overline{ty^a}^{\prec}{}_i}^{\,i} & 6.\ \overline{\overline{f^a}(C, cid_i) = \overline{f^a}^{\prec}{}_i}^{\,i} \\ 7.\ normalize(C, cid, \overline{fdef}) = \overline{fdef}' & \\ 8.\ \overline{\overline{tydef}_i = \{\, tydef \in \overline{tydef} \mid id_\tau(tydef) \in \overline{id_\tau}(\overline{ty^a}^{\prec}{}_i) \,\}}^{\,i} & \\ 9.\ \overline{\overline{fdef}_i = \{\, fdef \in \overline{fdef}'\, \overline{fdef}^{\prec} \mid sig(fdef) \in \overline{f^a}^{\prec}{}_i \,\}}^{\,i} & \\ 10.\ \overline{cm_i = \texttt{concept\_map}\, cid_i \{\, \overline{tydef}_i \overline{fdef}_i \,\}}^{\,i} & \end{array}}{icms(C, M, cid, cid?, \overline{tydef}, \overline{fdef}) = \overline{(cm_i, cid_{impl})}^{\,i}}$$

This judgment shows how implicit concept maps are generated. The "inputs" are the concepts environment C, the concept maps environment M, the concept identifier *cid* of the concept map that initiated generation process, the potential "parent" concept map *cid*?, and the type ($\overline{tydef}$) and function ($\overline{fdef}$) definitions from the concept map for the concept identified by *cid*. The "result" of the judgment is a set of generated concept maps. The first rule shows generation for concept maps that have a "parent" and the second rule shows generation for concept maps that do not have one (premise 1 in both rules). The second premise extracts identifiers of the concepts directly refined by the

concept identified by *cid* and the third premise keeps only the ones for which no concept map exists yet. Premise 4 finds function definitions in concept maps for concepts refined by *cid*; these are used later in premise 8. In premises 5–6, required associated types and functions are extracted for each of the directly refined concepts. In premise 7, the function definitions from the generating concept map are normalized, making sure that all type names are properly qualified to avoid name capture later on; the resulting set of function definitions is named $\overline{fdef}'$. In premises 8–9, the definitions for associated functions and types are taken from the available definitions. For associated types, the concept map for *cid* is guaranteed to contain a definition for every associated type in each refined concept, as described earlier. For associated functions, the definitions are taken from the current concept map and from the refined concept maps ($\overline{fdef}'\,\overline{fdef}^{\prec}$). According to the judgments described earlier, this sequence of functions is guaranteed to contain a definition for every associated function in refined concepts that is still undefined. Finally, premise 10 assembles the definitions into concept maps. Note that in the first rule the returned concept maps are constructed with cid_{impl} as the "parent" (in the conclusion of the rule), while in the second rule *cid* is set to be the "parent." This is because in the first rule the concept map for *cid* was itself implicitly defined and it simply propagates the "parent," while in the second rule the concept map for *cid* is explicitly defined and it is the "parent" itself.

IMPLICIT_CMAPS_NONE

$$\frac{\begin{array}{ll} 1.\ cid? = None & 2.\ C \vdash cid \prec_1 \overline{cid}^{\prec 1} \\ 3.\ \overline{cid_i}^{\,i} = \{\, cid \in \overline{cid}^{\prec 1} \mid cm(M, cid) = None \,\} & \\ 4.\ \overline{fdef}\,(C, M, None, cid) = \overline{fdef}^{\prec} & \\ 5.\ \overline{\overline{ty^a}\,(C, cid_i) = \overline{ty^a}^{\prec}{}_i}^{\,i} & 6.\ \overline{\overline{f^a}\,(C, cid_i) = \overline{f^a}^{\prec}{}_i}^{\,i} \\ 7.\ normalize\,(C, cid, \overline{fdef}) = \overline{fdef}' & \\ 8.\ \overline{\overline{tydef}_i = \{\, tydef \in \overline{tydef} \mid id_\tau(tydef) \in \overline{id_\tau}(\overline{ty^a}^{\prec}{}_i) \,\}}^{\,i} & \\ 9.\ \overline{\overline{fdef}_i = \{\, fdef \in \overline{fdef}'\,\overline{fdef}^{\prec} \mid sig(fdef) \in \overline{f^a}^{\prec}{}_i \,\}}^{\,i} & \\ 10.\ \overline{cm_i = \texttt{concept\_map}\, cid_i\, \{\, \overline{tydef}_i \overline{fdef}_i \,\}}^{\,i} & \end{array}}{icms\,(C, M, cid, cid?, \overline{tydef}, \overline{fdef}) = \overline{(cm_i, cid)}^{\,i}}$$

program instantiation

$$\boxed{C; M; T \vdash P \Downarrow C'; M'; T'}$$

P_INST_NIL

$$\frac{}{C; M; T \vdash \Downarrow C; M; T}$$

This judgment processes a meta-program, definition by definition. The check is performed in the environment $C; M; T$ where C is the sequence of all defined concepts, M is the sequence of all defined concept maps, and T is the sequence of all defined templates. The program P produces a new environment $C'; M'; T'$ if it is correct—every definition in P extends the appropriate part of the environment.

The first rule handles empty programs: nothing has to be done.

P_INST_TMPL

1. $tl = \texttt{template}\,\texttt{requires}\,\overline{cid_i}^{\,i}\;tn\,\{\,e\,\}$
2. $\overline{cid_i}$ defined in C^i
3. tl not defined in T
4. $C\,;\emptyset\,;\overline{cid_i}^{\,i} \vdash e : \tau$
5. $C\,;M\,;tl\,T \vdash P \Downarrow C'\,;M'\,;T'$

$$C\,;M\,;T \vdash tl\,P \Downarrow C'\,;M'\,;T'$$

The second rule handles template definitions. The first premise simply creates an alias for a template given as the "input" in the conclusion so that parts of the template can be accessed. The second premise checks that concepts used in the refinement clause of the template are defined in the environment and the third rule checks that the template named tn is not already defined. In premise 4, the expression e contained in the template is type checked given C, an empty variable environment $\emptyset$, and the requirements of the template $\overline{cid_i}^{\,i}$. Finally, the last premise asserts that the rest of the program evaluates to a new environment given the original environment extended with the newly defined template.

P_INST_CONCEPT

1. cp not defined in C
2. $cp\,C\,;M\,;T \vdash P \Downarrow C'\,;M'\,;T'$

$$C\,;M\,;T \vdash cp\,P \Downarrow C'\,;M'\,;T'$$

Currently, there are is no checks for concept definitions beyond ensuring that a concept is not being redefined (premise 1).

P_INST_CMAP

1. $C\,;M\,;T \vdash (cm, None)\,P \Downarrow C'\,;M'\,;T$

$$C\,;M\,;T \vdash cm\,P \Downarrow C'\,;M'\,;T$$

Checking of a concept map is forwarded to the next rule for checking of implicit concept maps. The "input" concept map cm, given in the conclusion, is used to construct an implicit concept map $(cm, None)$, which is checked by the rule for implicit concept maps in the only premise.

P_INST_CMAP_IMPL

1. $icm = (cm, cid?)$
2. $cm = \texttt{concept\_map}\,cid\,\{\,\overline{tydef}\,\overline{fdef}\,\}$
3. cm not defined in M
4. $tydefs\text{-}check\,(C, cid, \overline{tydef})$
5. $tydefs_{=}\,(C, M, cid, \overline{tydef})$
6. $fdefs\text{-}check\,(C, cid, \overline{fdef})$
7. $fdefs_{=}\,(C, M, cid, cid?, \overline{fdef})$
8. $icms\,(C, M, cid, cid?, \overline{tydef}, \overline{fdef}) = \overline{icm_i}^{\,i}$
9. $C\,;icm\,M\,;T \vdash \overline{icm_i}^{\,i}\,P \Downarrow C'\,;M'\,;T'$

$$C\,;M\,;T \vdash icm\,P \Downarrow C'\,;M'\,;T'$$

The last rule processes implicit concept maps. The first premise simply creates an alias to the implicit concept map given as an "input" and the second premise creates an alias

to the concept map part of the implicit concept map tuple. The third premise asserts that *cm* does not redefine an existing concept map. The following 5 premises perform, in order, checking of type definitions, compatibility check of type definitions, function definitions check, function definitions compatibility check, and generation of the implicit concept map. All of these tasks have been described earlier in this document. The last premise processes the rest of the program.

5 Separate Type Checking Safety

The goal of separate type checking is to guarantee that templates can be checked against their requirements at the time of definition. Then, definitions in concept maps are checked separately. Finally, when using a template, it only has to be checked that the necessary concept maps exist. In our core language, the only task remaining is to substitute definitions but no type checking is required: the previous checks guarantee that type checking succeeds. The details of separate type checking are given in (Zalewski and Schupp, 2009), here we give only a brief overview.

Assume $\emptyset;\emptyset;\emptyset \vdash P \Downarrow C';M';T'$

Then: For any template $tmpl \in T$, if concept maps enumerated in the requirement clause of the template exist, then the template expression is guaranteed to type check after type definitions from concept maps have been substituted for type names used in the expression (type checking succeeds in the context of function definitions given in concept maps).

This is an informal statement of the separate type checking safety property. The following properties are implied:

- No Argument-Dependent Lookup (ADL) is necessary. All names are resolved during type checking of templates and then bound to the implementations provided in concept maps.
- No errors can occur during template instantiation if the required concept maps are defined.
- Programs developed independently against a common set of concepts are guaranteed to type check when used together.

In our restricted language, separate type checking safety is indeed guaranteed for all cases yet in C++ there are some cases where separate type checking does not exclude errors at template instantiation time. Järvi et al. (2006) discuss in detail how separate type checking may fail, due to concept-based overloading. Our core language investigates other aspects of concepts design, in particular we concentrate on name lookup and refinement. Since we do not include the features that may break the safety of separate type checking, template instantiation never results in errors.

user program instantiation

$$\boxed{P \Downarrow C;M;T}$$

$$\frac{1.\ \emptyset;\emptyset;\emptyset \vdash P_{user} \Downarrow C;M;T}{P_{user} \Downarrow C;M;T}\ \text{P\_USER\_INST}$$

6 Statistics

```
Definition rules:        72 good    0 bad
Definition rule clauses: 273 good    0 bad
```

References

D. Abrahams and A. Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond*. C++ In-Depth Series. Addison-Wesley, 2004.
→ 1 citation on page: 21

D. Abrahams, J. Siek, and T. Witt. The Boost Iterator Library. `http://www.boost.org/libs/iterator`, Sept. 2008.
→ 1 citation on page: 3

Ada. *The Ada Programming Language Reference Manual*. US Department of Defense, 1983. ANSI/MIL-STD-1815A-1983 Document.
→ 1 citation on page: 114

A. Alexandrescu. *Modern C++ Design*. C++ In–Depth Series. Addison-Wesley, 2001.
→ 1 citation on page: 147

G. Antoniol, G. Canfora, G. Casazza, A. D. Lucia, and E. Merlo. Recovering traceability links between code and documentation. *IEEE Trans. Softw. Eng.*, 28 (10):970–983, 2002. ISSN 0098-5589. doi: http://dx.doi.org/10.1109/TSE.2002.1041053.
→ 1 citation on page: 75

D. Aspinall and D. Sannella. From specifications to code in CASL. In *Proc. 9th Int. Conf. on Algebraic Methodology and Software Technology*, volume 2422 of *LNCS*, pages 11–40. Springer, Sept. 2002.
→ 1 citation on page: 114

M. H. Austern. *Generic Programming and the STL: Using and Extending the C++ Standard Template Library*. Addison-Wesley, 1998.
→ 2 citations on 2 pages: 2 and 67

M. H. Austern. Segmented iterators and hierarchical algorithms. In M. Jazayeri, R. Loos, and D. R. Musser, editors, *Selected Papers from the International Seminar on Generic Programming*, pages 80–90. Springer, 2000.
→ 1 citation on page: 147

S. Bates and S. Horwitz. Incremental program testing using program dependence graphs. In *Proc. 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 384–396. ACM Press, 1993.
→ 1 citation on page: 75

J.-P. Bernardy, P. Jansson, M. Zalewski, S. Schupp, and A. Priesnitz. A comparison of C++ concepts and Haskell type classes. In *Proc. ACM SIGPLAN Workshop on Generic Programming*, 2008. Accepted for publication.
→ 3 citations on 3 pages: 15, 16, and 17

M. Bidoit and P. D. Mosses. *CASL User Manual*, volume 2900 of *LNCS*. Springer, 2004.
→ 4 citations on 4 pages: 20, 94, 101, and 102

G. Birkhoff and J. Lipson. Heterogeneous algebras. *Journal of Combinatorial Theory*, 8(1):115–133, Jan. 1970.
→ 1 citation on page: 18

S. A. Bohner. Software change impacts—an evolving perspective. In *Software Maintenance, Proc..*, pages 263–272. IEEE Press, 2002.
→ 1 citation on page: 25

S. A. Bohner and R. S. Arnold. An introduction to software change impact analysis. In *Software Change Impact Analysis* Bohner and Arnold (1996b), pages 1–26.
→ 1 citation on page: 25

S. A. Bohner and R. S. Arnold. *Software Change Impact Analysis*. Wiley-IEEE, 1996b.
→ 2 citations on 2 pages: 25 and 200

Boost. The Boost initiative for free peer-reviewed portable C++ source libraries. `http://www.boost.org`, Sept. 2008.
→ 3 citations on 3 pages: 2, 133, and 148

N. Botta and C. Ionescu. Relation-Based computations in a monadic BSP model. *Parallel Computing*, 33(12), Dec. 2007.
→ 2 citations on 2 pages: 119 and 120

N. Botta, C. Ionescu, C. Linstead, and R. Klein. Structuring distributed relation-based computations with SCDRC. Technical Report 103, Potsdam Institute for Climate Impact Research, Sept. 2006.
→ 1 citation on page: 119

N. Botta, C. Ionescu, R. Klein, and C. Linstead. S—Software Components for Distributed Adaptive Finite Volume Methods. `http://www.pik-potsdam.de/research/research-domains/transdisciplinary-concepts-and-methods/s`, Sept. 2008.
→ 1 citation on page: 119

A. Breuer, P. Gottschling, D. Gregor, and A. Lumsdaine. Effecting parallel graph eigensolvers through library composition. In *Proc. 20th Int. Parallel and Distributed Processing Symp. (IPDPS)*, Apr. 2006.
→ 1 citation on page: 67

L. Briand, Y. Labiche, and L. O'Sullivan. Impact analysis and change management of UML models. In *Proc. 19th IEEE Int. Conf. on Software Maintenance (ICSM)*, pages 256–265. IEEE Computer Society, 2003.
→ 1 citation on page: 75

R. M. Burstall. Proving properties of programs by structural induction. *The Computer Journal*, 12(1):41–48, 1969.
→ 1 citation on page: 25

L. Cardelli and P. Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *ACM Comput. Surv.*, 17(4):471–523, 1985.
→ 3 citations on 3 pages: 20, 21, and 122

CGAL. Computational Geometry Algorithms Library (CGAL). `http://www.cgal.org/`, Aug. 2008.
→ 1 citation on page: 3

M. M. T. Chakravarty, G. Keller, and S. Peyton Jones. Associated type synonyms. In *Proc. 10th ACM SIGPLAN Int. Conf. on Functional Programming (ICFP)*, pages 241–253. ACM Press, Sept. 2005a.
→ 4 citations on 4 pages: 14, 119, 124, and 125

M. M. T. Chakravarty, G. Keller, S. Peyton Jones, and S. Marlow. Associated types with class. In *Proc. 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 1–13. ACM Press, 2005b.
→ 1 citation on page: 14

M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Maude: specification and programming in rewriting logic. *Theoretical Computer Science*, 285(2):187–243, August 2002.
→ 1 citation on page: 27

K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Tools and Applications*. Addison-Wesley, 2000.
→ 1 citation on page: 143

A. Van Deursen and J. Visser. Source model analysis using the JJTraveler visitor combinator framework. *Software Practice and Experience*, 34(14):1345–1379, 2004.
→ 1 citation on page: 147

DevX.com. Special report. C++0x: The dawning of a new standard. http://www.devx.com/SpecialReports/Door/38865, Aug. 2008.
→ 1 citation on page: 5

P. Dimov. Boost Bind library. `http://www.boost.org/libs/bind`, Sept. 2008.
→ 1 citation on page: 136

U. W. Eisenecker, F. Blinn, and K. Czarnecki. A solution to the constructor-problem of mixin-based programming in C++. In *Proc. 1st Workshop on C++ Template Programming*, Oct. 2000.
→ 1 citation on page: 143

M. J. Fyson and C. Boldyreff. Using application understanding to support impact analysis. *Software Maintenance: Research and Practice*, 10(2):93–110, 1998.
→ 2 citations on 2 pages: 25 and 75

E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
→ 1 citation on page: 147

R. Garcia and A. Lumsdaine. Multiarray: a C++ library for generic programming with arrays. *Software: Practice and Experience*, 35(2):159–188, 2005.
→ 1 citation on page: 21

R. Garcia, J. Järvi, A. Lumsdaine, J. G. Siek, and J. Willcock. A Comparative Study of Language Support for Generic Programming. In *Proc. 18th ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 115–134. ACM Press, Oct. 2003.
→ 2 citations on 2 pages: 12 and 124

R. Garcia, J. Jarvi, A. Lumsdaine, J. Siek, and J. Willcock. An extended comparative study of language support for generic programming. *J. Funct. Program.*, 17(2): 145–205, Mar. 2007.
→ 7 citations on 4 pages: 12, 13, 14, and 15

GHC. Glasgow Haskell Compiler. `http://haskell.org/ghc/`, Sept. 2008.
→ 1 citation on page: 123

J. Gibbons. Datatype-generic programming. In R. Backhouse, J. Gibbons, R. Hinze, and J. Jeuring, editors, *Spring School on Datatype-Generic Programming*, volume 4719 of *LNCS*, pages 1–71. Springer, 2007.
→ 3 citations on page: 1

J.-Y. Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Universite Paris VII, France, 1972.
→ 4 citations on 4 pages: 5, 6, 67, and 114

GLAS. GLAS: Generic Linear Algebra Software. `http://glas.sourceforge.net/`, Aug. 2008.
→ 1 citation on page: 3

J. Goguen. Parameterized programming. *IEEE Transactions on Software Engineering*, 10(5):528–543, 1984.
→ 1 citation on page: 20

J. Goguen. Reusing and interconnecting software components. *Computer*, 19(2): 16–28, Feb. 1986.
→ 1 citation on page: 102

J. Goguen. Hyperprogramming: A formal approach to software environments. In *Proc. Symposium on Formal Methods in Software Development*, Tokyo, Japan, Jan. 1990.
→ 1 citation on page: 114

J. Goguen. Principles of parameterized programming. In *Software Reusability*, volume 1, pages 159–225. Addison-Wesley, 1989.
→ 1 citation on page: 114

J. Goguen and R. Burstall. Institutions: abstract model theory for specification and programming. *J. ACM*, 39(1):95–146, 1992.
→ 7 citations on 5 pages: 19, 67, 92, 100, and 107

J. Goguen and K. Levitt, editors. *Report on Program Libraries Workshop*, Menlo Park, CA, USA, 1983. SRI International.
→ 1 citation on page: 102

J. Goguen and G. Roşu. Institution morphisms. *Formal Aspects of Computing*, 13 (3–5):274–307, 2002.
→ 1 citation on page: 100

J. Goguen and W. Tracz. An implementation-oriented semantics for module composition. In G. Leavens and M. Sitaraman, editors, *Foundations of Component-Based Systems*, pages 231–263. Cambridge, Apr. 2000.
→ 2 citations on 2 pages: 102 and 114

J. Goguen, J. W. Thatcher, and E. G. Wagner. An initial algebra approach to the specification, correctness and implementation of abstract data types. In R. Yeh, editor, *Current Trends in Programming Methodology: Data Structuring*, volume 4, pages 80–149. Prentice-Hall, 1978.
→ 2 citations on 2 pages: 17 and 18

R. Goldblatt. *Topoi, The Categorial Analysis of Logic*. North-Holland, revised edition edition, 1984.
→ 3 citations on 2 pages: 98 and 99

J. Gosling, B. Joy, and G. L. Steele. *The Java Language Specification*. Addison-Wesley, 1996.
→ 1 citation on page: 1

P. Gottschling. Fundamental algebraic concepts in concept-enabled C++. Technical Report 638, Indiana University, 2006.
→ 3 citations on 3 pages: 93, 96, and 122

GrammaTech. Pathinspector. `http://www.grammatech.com/products/codesurfer`, Sept. 2008.
→ 1 citation on page: 87

G. Grätzer. *Universal Algebra*. Springer, second edition, 1979.
→ 1 citation on page: 18

D. Gregor. ConceptGCC: Concept extensions for C++. `http://www.generic-programming.org/software/ConceptGCC/`, Sept. 2008a.
→ 4 citations on 4 pages: 22, 27, 87, and 94

D. Gregor. *High-Level Static Analysis for Generic Libraries*. PhD thesis, Rensselaer Polytechnic Institute, May 2004.
→ 1 citation on page: 93

D. Gregor. Easier C++: An introduction to concepts. `http://www.devx.com/SpecialReports/Article/38864/`, Aug. 2008b.
→ 1 citation on page: 7

D. Gregor. Type-soundness and optimization in the concepts proposal. Technical Report N2576=08-0086, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, Mar. 2008c.
→ 2 citations on 2 pages: 7 and 17

D. Gregor and A. Lumsdaine. Lifting sequential graph algorithms for distributed-memory parallel computation. In *Proc. 20th ACM SIGPLAN Conf. on*

Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), pages 423–437. ACM, 2005.
→ 1 citation on page: 67

D. Gregor and A. Lumsdaine. Concepts for the C++0x standard library: Iterators (revision 3). Technical Report N2734=08-0244, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, Aug. 2008a.
→ 1 citation on page: 22

D. Gregor and A. Lumsdaine. Concepts for the C++0x standard library: Numerics (revision 3). Technical Report N2736=08-0246, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, Aug. 2008b.
→ 1 citation on page: 22

D. Gregor and A. Lumsdaine. Concepts for the C++0x standard library: Utilities (revision 4). Technical Report N2735=08-0245, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, Aug. 2008c.
→ 3 citations on 3 pages: 22, 93, and 97

D. Gregor and J. Siek. Implementing concepts. Technical Report N1848=05-0108, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, Aug. 2005.
→ 1 citation on page: 87

D. Gregor and B. Stroustrup. Proposed wording for concepts (revision 1). Technical Report N2307=07-0167, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, July 2007.
→ 6 citations on 4 pages: 95, 96, 103, and 119

D. Gregor, J. Siek, J. Willcock, J. Jarvi, R. Garcia, and A. Lumsdaine. Concepts for C++0x (revision 1). Technical Report N1849=05-0109, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, 2005.

→ 2 citations on 2 pages: 84 and 87

D. Gregor, J. Järvi, J. Siek, A. Lumsdaine, G. D. Reis, and B. Stroustrup. Concepts: First-class language support for generic programming. In *Proc. 21st ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Oct. 2006a.
→ 2 citations on 2 pages: 74 and 145

D. Gregor, J. Järvi, J. Siek, B. Stroustrup, G. Dos Reis, and A. Lumsdaine. Concepts: Linguistic support for generic programming in C++. In *Proc. ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 291–310. ACM Press, 2006b.
→ 7 citations on 7 pages: 5, 22, 31, 67, 91, 119, and 159

D. Gregor, M. Marcus, and P. Halpern. Concepts for the C++0x standard library: Containers (revision 3). Technical Report N2738=08-0248, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, Aug. 2008a.
→ 1 citation on page: 22

D. Gregor, M. Marcus, T. Witt, and A. Lumsdaine. Concepts for the C++0x standard library: Algorithms (revision 4). Technical Report N2740=08-0250, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, Aug. 2008b.
→ 1 citation on page: 22

D. Gregor, M. Marcus, T. Witt, and A. Lumsdaine. Foundational concepts for the C++0x standard library (revision 4). Technical Report N2737=08-0247, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, Aug. 2008c.
→ 1 citation on page: 22

D. Gregor, J. Siek, and A. Lumsdaine. Iterator concepts for the C++0x standard library (revision 4). Technical Report N2739=08-0249, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, Aug. 2008d.
→ 2 citations on 2 pages: 5 and 22

D. Gregor, B. Stroustrup, J. Widman, and J. Siek. Proposed wording for concepts (revision 5). Technical Report N2617=08-0127, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, May 2008e.
→ 18 citations on 14 pages: 2, 22, 23, 31, 32, 33, 35, 37, 39, 42, 63, 68, 159, and 183

D. Gregor, B. Stroustrup, J. Widman, and J. Siek. Proposed wording for concepts (revision 8). Technical Report N2741=08-0251, ISO/IEC JTC1/SC22/WG21 - C++, Aug. 2008f.
→ 3 citations on 3 pages: 7, 11, and 22

J. de Guzman, D. Marsden, and T. Schwinger. Fusion library homepage, Sept. 2008. `http://www.boost.org/libs/fusion`.
→ 5 citations on 5 pages: 133, 143, 144, 146, and 147

J. Han. Supporting impact analysis and change propagation in software engineering environments. In *Proc. 8th IEEE Int. Workshop on Software Technology and Engineering Practice Incorporating Computer Aided Software Engineering*, pages 172–182. IEEE Computer Society, 1997.
→ 2 citations on page: 75

M. Haveraaen. Institutions, property-aware programming and testing. In *Proc. ACM SIGPLAN Symp. on Library-Centric Software Design (LCSD)*, 2007. To appear.
→ 5 citations on 3 pages: 22, 67, and 93

J. Heflin and J. A. Hendler. Dynamic ontologies on the web. In *Proc. 17th National Conf. on Artificial Intelligence and 12th Conf. on Innovative Applications of Artificial Intelligence (AAAI/IAAI)*, pages 443–449. AAAI Press/MIT Press, 2000.
→ 1 citation on page: 75

R. Hinze and A. Löh. "Scrap your boilerplate" revolutions. In T. Uustalu, editor, *Proc. 8th Int. Conf. on the Mathematics of Program Construction (MPC)*, Lecture Notes in Computer Science. Springer, 2006.
→ 3 citations on 2 pages: 133 and 147

R. Hinze, A. Löh, and B. C. d. S. Oliveira. "Scrap your boilerplate" reloaded. In P. Wadler and M. Hagiya, editors, *Proc. 8th Int. Symposium on Functional and Logic Programming (FLOPS)*, pages 13–29, 2006.
→ 3 citations on 2 pages: 133 and 147

S. Horwitz and T. Reps. The use of program dependence graphs in software engineering. In *Proc. 14th Int. Conf. on Software Engineering (ICSE)*, pages 392–411. ACM Press, 1992.
→ 1 citation on page: 75

C++ Standard Draft, N1804=05-0064. ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, ANSI standards for information technology edition, 2003a.
→ 3 citations on 3 pages: 82, 83, and 84

ISO/IEC 14882 Int. Standard for Information Systems – Programming Languages – C++ . ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, 2nd edition, 2003b.
→ 5 citations on 5 pages: 1, 2, 21, 23, and 159

Iss96. C++ Standard Library issue 96: `vector<bool>` is not a container. `http://www.open-std.org/jtc1/sc22/wg21/docs/lwg-active.html#96`, Sept. 2008.
→ 1 citation on page: 4

J. Järvi and G. Powell. Boost Tuple Library (BTL) homepage. `http://www.boost.org/libs/tuple`, Sept. 2008.
→ 1 citation on page: 143

J. Järvi, A. Lumsdaine, J. Siek, and J. Willcock. An analysis of constrained polymorphism for generic programming. In K. Davis and J. Striegnitz, editors, *Proc. Multiparadigm Programming with Object-Oriented Languages (MPOOL) Workshop*, pages 87–107, October 2003.
→ 2 citations on 2 pages: 21 and 119

J. Järvi, G. Powell, and A. Lumsdaine. The Lambda Library: unnamed functions in C++. *Software Practice and Experience*, 33:259–291, 2003.
→ 1 citation on page: 148

J. Järvi, J. Willcock, and A. Lumsdaine. Concept-Controlled Polymorphism. In *Proc. 2nd Int. Conf. Generative Programming and Component Engineering (GPCE)*, volume 2830 of *LNCS*, pages 228–244. Springer, 2003.
→ 2 citations on 2 pages: 119 and 138

J. Järvi, D. Gregor, J. Willcock, A. Lumsdaine, and J. Siek. Algorithm specialization in generic programming: challenges of constrained generics in C++. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, pages 272–282. ACM Press, 2006.
→ 8 citations on 8 pages: 5, 7, 17, 31, 49, 68, 97, and 196

J. Järvi, M. A. Marcus, and J. N. Smith. Library composition and adaptation using C++ concepts. In *Proc. 6th Int. Conf. on Generative Programming and Component Engineering (GPCE)*, pages 73–82. ACM Press, 2007.
→ 2 citations on 2 pages: 22 and 67

M. Jazayeri, R. Loos, and D. Musser, editors. *Generic Programming: International Seminar, Dagstuhl Castle, Germany, 1998, Selected Papers*, volume 1766 of *LNCS*, 2000. Springer.
→ 2 citations on 2 pages: 2 and 209

M. P. Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. In *Proc. Conf. on Functional Programming Languages and Computer Architecture (FPCA)*, pages 52–61. ACM Press, 1993.
→ 1 citation on page: 6

M. P. Jones. *Qualified Types: Theory and Practice*. Distinguished Dissertations in Computer Science. Cambridge University Press, 1994.
→ 1 citation on page: 21

M. P. Jones. Type Classes with Functional Dependencies. In *Proc. 9th European Symp. on Programming Languages and Systems*, pages 230–244, London, UK, 2000. Springer. ISBN 3-540-67262-1.
→ 2 citations on 2 pages: 119 and 122

S. P. Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.
→ 1 citation on page: 114

G. Kahn. Natural semantics. In G. Goos and J. Hartmanis, editors, *Proc. 4th Annual Symposium on Theoretical Aspects of Computer Science (STACS)*, volume 247 of *LNCS*, pages 22–39. Springer, 1987.
→ 6 citations on 2 pages: 24 and 25

S. Kahrs, D. Sannella, and A. Tarlecki. The definition of extended ML: A gentle introduction. *Theoretical Computer Science*, 173(2):445–484, February 1997.
→ 1 citation on page: 20

D. Kapur and D. Musser. Tecton: a framework for specifying and verifying generic system components. Technical Report RPI-92-20, Department of Computer Science, Rensselaer Polytechnic Institute, Troy, New York, 1992.
→ 6 citations on 5 pages: 5, 31, 67, 73, and 114

D. Kapur, D. Musser, and A. Stepanov. Operators and algebraic structures. In *Proc. Conf. on Functional Programming Languages and Computer Architecture*. ACM Press, 1981a.
→ 1 citation on page: 91

D. Kapur, D. R. Musser, and A. A. Stepanov. Tecton: A language for manipulating generic objects. In J. Staunstrup, editor, *Proc. Workshop on Program Specification*, volume 134 of *LNCS*, pages 412–414. Springer, 1981b.
→ 3 citations on 3 pages: 2, 5, and 91

A. Kennedy and D. Syme. Design and implementation of generics for the .NET common language runtime. In *Proc. ACM SIGPLAN 2001 Conf. on Programming Language Design and Implementation (PLDI)*, pages 1–12. ACM Press, 2001.
→ 1 citation on page: 1

S. Kothari and M. Sulzmann. C++ Templates/Traits versus Haskell Type Classes. Technical Report TRB2/05, The National Univ. of Singapore, 2005.
→ 1 citation on page: 119

T. Kühne. Internal iteration externalized. In R. Guerraoui, editor, *ECCOP'99 - Object-Oriented Programming, 13th European Conf (ECOOP)*, volume 1628 of *Lecture Notes in Computer Science*, pages 329–350. Springer, 1999. ISBN 3-540-66156-5.
→ 1 citation on page: 146

D. Kung, J. Gao, P. Hsia, F. Wen, Y. Toyoshima, and C. Chen. Change impact identification in object oriented software maintenance. In *Proc. Int. Conf. on Software Maintenance (ICSM)*, pages 202–211. IEEE Computer Society, 1994.
→ 1 citation on page: 74

R. Lämmel and S. Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. *ACM SIGPLAN Notices*, 38(3):26–37, Mar. 2003. Proc. ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI).
→ 9 citations on 6 pages: 26, 133, 134, 137, 145, and 147

R. Lämmel and S. Peyton Jones. Scrap more boilerplate: reflection, zips, and generalised casts. In C. Okasaki and K. Fisher, editors, *Proc. 9th ACM SIGPLAN Int. Conf. on Functional Programming (ICFP)*, pages 244–255. ACM Press, 2004. ISBN 1-58113-905-5.
→ 5 citations on 4 pages: 133, 145, 146, and 149

R. Lämmel and S. Peyton Jones. Scrap your boilerplate with class: extensible generic functions. In O. Danvy and B. C. Pierce, editors, *Proc. 10th ACM SIGPLAN Int. Conf. on Functional Programming (ICFP)*, pages 204–215. ACM Press, Sept. 2005. ISBN 1-59593-064-7.
→ 4 citations on 4 pages: 133, 140, 145, and 146

F. Lawvere. The category of categories as a foundation for mathematics. In *Proc. Conf. on Categorical Algebra*, pages 1–20. Springer, 1966.
→ 2 citations on 2 pages: 99 and 102

S. MacLane. *Categories for the Working Mathematician*. Springer, 1971.
→ 1 citation on page: 19

J. Matthews, R. B. Findler, M. Flatt, and M. Felleisen. A visual environment for developing context-sensitive term rewriting systems. In *Proc. 15th Int. Conf. on Rewriting Techniques and Applications*, volume 3091 of *LCNS*, pages 310–311. Springer, 2004.
→ 2 citations on 2 pages: 27 and 69

B. McNamara and Y. Smaragdakis. Functional programming with the FC++ library. *J. Functional Programming*, 14:429–472, 2004a.
→ 2 citations on 2 pages: 133 and 148

B. McNamara and Y. Smaragdakis. Functional Programming with the FC++ Library. *Journal of Functional Programming*, 14(4):429–472, July 2004b.
→ 1 citation on page: 120

A. Meredith. State of C++ evolution (pre-antipolis 2008 mailing). Technical Report N2617=08-0127, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, May 2008.
→ 1 citation on page: 31

W. M. Miller. C++ standard core language active issues, revision 58. Technical Report N2714=08-0224, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, Aug. 2008.
→ 1 citation on page: 23

R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990a.
→ 2 citations on 2 pages: 6 and 20

R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990b.
→ 2 citations on 2 pages: 67 and 91

T. Mossakowski, C. Maeder, and K. Lüttich. The Heterogeneous Tool Set. In O. Grumberg and M. Huth, editors, *TACAS 2007*, volume 4424 of *LNCS*, pages 519–522. Springer, 2007.
→ 1 citation on page: 95

P. D. Mosses, editor. *CASL Reference Manual*, volume 2960 of *LNCS*. Springer, 2004.
→ 4 citations on 4 pages: 20, 95, 101, and 102

D. Musser and A. Stepanov. Generic programming. In *Proc. Int. Symopsium on Symbolic and Algebraic Computation*, volume 358 of *LNCS*, pages 13–25, 1998.
→ 1 citation on page: 91

D. Musser and C. Wang. A basis for formal specification and verification of generic algorithms in the C++ standard template library. Technical Report 95-1, Dept. of Computer Science, Rensselaer Polytechnic Inst., Jan. 1995.
→ 1 citation on page: 93

D. Musser, S. Schupp, and R. Loos. Requirements-oriented programming. In Jazayeri et al. (2000).
→ 1 citation on page: 114

D. R. Musser. Tecton description of STL container and iterator concepts. `http://www.cs.rpi.edu/~musser/gp/tecton/container.ps`, Sept. 2008.
→ 1 citation on page: 5

D. R. Musser and A. A. Stepanov. Generic programming. In *Proc. Int. Symposium on Symbolic and Algebraic Computation (ISSAC'88)*, volume 358 of *LNCS*, pages 13–25. Springer, 1989.
→ 1 citation on page: 3

D. R. Musser, S. Schupp, C. Schwarzweller, and R. Loos. Tecton concept library. Technical Report WSI-99-2, Fakultät für Informatik, Universität Tübingen, 1999.
→ 1 citation on page: 5

D. R. Musser, G. J. Derge, and A. Saini. *STL Tutorial and Reference Guide. C++ Programming with the Standard Template Library*. Addison-Wesley, 2nd edition, 2001.
→ 3 citations on 3 pages: 2, 31, and 73

N. Myers. A new and useful template technique: "Traits". *C++ Report*, 7(5):32–35, June 1995.
→ 1 citation on page: 141

E. Niebler. Segmented Fusion - a-ha!, 2006.
`http://article.gmane.org/gmane.comp.parsers.spirit.devel/2765`.
→ 2 citations on 2 pages: 146 and 147

F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, Dec. 2004.
→ 1 citation on page: 25

H. R. Nielson and F. Nielson. *Semantics with Applications: A Formal Introduction*. Wiley, 1992.
→ 2 citations on 2 pages: 23 and 25

P. Nogueira Iglesias. *Polytypic Functional Programming and Data Abstraction*. PhD thesis, School of Comp. Science and Information Technology, The University of Nottingham, UK, Jan. 2006.
→ 1 citation on page: 145

J. O'Neal. Analyzing the impact of changing requirements. In *Proc. of the Int. Conf. on Softw. Maintenance (ICSM)*, pages 190–198, 2001. ISBN 0-7695-1189-9.
→ 1 citation on page: 75

A. Orso, T. Apiwattanapong, and M. Harrold. Leveraging field data for impact analysis and regression testing. In *Proc. of the 9th Europ. Softw. Eng. Conf. & 11th ACM SIGSOFT Int. Symp. Foundations of Softw. Eng.*, pages 128–137, 2003. ISBN 1-58113-743-5. doi: http://doi.acm.org/10.1145/940071.940089.
→ 2 citations on 2 pages: 25 and 75

S. Owens. A Sound Semantics for OCaml$_{light}$. In *Proc. 17th European Symp. on Programming (ESOP)*, volume 4960 of *LNCS*, pages 1–15. Springer, 2008.
→ 1 citation on page: 68

J. Palsberg and C. Jay. The essence of the visitor pattern. In *Proc. 22nd Comp. Software and Applications Conf. (COMPSAC)*, pages 9–15, 1998.
→ 1 citation on page: 147

S. Peyton Jones. *Haskell 98 Language and Libraries: the Revised Report*. Cambridge University Press, 2003.
→ 2 citations on 2 pages: 1 and 148

B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.
→ 2 citations on 2 pages: 6 and 48

P. Pirkelbauer, S. Parent, M. Marcus, and B. Stroustrup. Runtime concepts for the C++Standard Template Library. In *Proc. ACM Symposium on Applied Computing (SAC)*, pages 171–177. ACM, 2008.
→ 1 citation on page: 22

G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981.
→ 4 citations on 2 pages: 24 and 25

G. D. Plotkin. The origins of structural operational semantics. *Journal of Logic and Algebraic Programming*, 60–61:3–15, 2004.
→ 1 citation on page: 25

D. Prawitz. Ideas and results in proof theory. In J. Fenstad, editor, *Proc. 2nd Scandinavian Logic Symposium*, pages 237–309. North-Holland, 1971.
→ 1 citation on page: 24

A. Priesnitz and S. Schupp. From generic invocations to generic implementations. In *6th Workshop on Parallel/High-Performance Object-Oriented Scientific Computing (POOSC)*, July 2006. to appear.
→ 2 citations on 2 pages: 133 and 148

R. Prieto-Diaz and J. Neighbors. Module interconnect language. *Journal of Systems and Software*, 6(4):307–344, Nov. 1986.
→ 1 citation on page: 114

V. Rajlich. A model for change propagation based on graph rewriting. In *Proc. of the Int. Conf. on Softw. Maintenance (ICSM)*, pages 84–91, 1997. ISBN 0-8186-8013-X.
→ 1 citation on page: 75

G. D. Reis and B. Stroustrup. Specifying C++ concepts. In *Proc. 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 295–308. ACM Press, 2006.
→ 2 citations on 2 pages: 74 and 145

G. Dos Reis and B. Stroustrup. A formalism for C++. Technical Report N1885=05-0145, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, Oct. 2005.
→ 1 citation on page: 67

G. Dos Reis and B. Stroustrup. Specifying C++ concepts. In *Proc. 33rd ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL)*, pages 295–308. ACM Press, 2006.
→ 9 citations on 9 pages: 5, 31, 32, 49, 66, 67, 91, 119, and 159

X. Ren, F. Shah, F. Tip, B. Ryder, and O.Chesley. Chianti: A tool for change impact analysis of Java programs. In *Proc. of the 19th annual ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 432–448, 2004.
→ 2 citations on 2 pages: 25 and 75

X. Ren, B. Ryder, M. Stoerzer, and F. Tip. Chianti: a change impact analysis tool for java programs. In *Proc. of the 27th Int. Conf. on Softw. Eng.*, pages 664–665, New York, NY, USA, 2005. ACM Press. ISBN 1-59593-963-2. doi: http://doi.acm.org/10.1145/1062455.1062598.
→ 2 citations on 2 pages: 25 and 75

J. G. Rossie and D. P. Friedman. An algebraic semantics of subobjects. *SIGPLAN Not.*, 30(10):187–199, Oct. 1995.
→ 1 citation on page: 67

G. Rothermel and M. J. Harrold. A safe, efficient regression test selection technique. *ACM Trans. on Softw. Eng. and Methodology*, 6(2):173–210, Apr. 1997.
→ 1 citation on page: 75

B. Ryder and F. Tip. Change impact analysis for object-oriented programs. In *Proc. of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 46–53, 2001. ISBN 1-58113-413-4. doi: http://doi.acm.org/10.1145/379605.379661.
→ 1 citation on page: 75

D. Sannella and A. Tarlecki. Specifications in an arbitrary institution. *Inf. Comput.*, 76(2-3):165–210, 1988.
→ 2 citations on 2 pages: 27 and 100

D. A. Schmidt. *Denotational Semantics: A Methodology for Language Development*. William C. Brown Publishers, 1986.
→ 3 citations on 2 pages: 23 and 24

L. Schröder and T. Mossakowski. HasCASL: towards integrated specification and development of functional programs. In H. Kirchner and C. Ringeissen, editors, *Algebraic Methodology And Software Technology (AMAST 2002)*, volume 2422 of *LNCS*, pages 99–116. Springer, 2002.
→ 2 citations on page: 114

S. Schupp, D. Gregor, and D. Musser. Algebraic concepts represented in C++. Technical Report TR-00-8, Rensselaer Polytechnic Institute, 2000.
→ 1 citation on page: 114

S. Schupp, D. Gregor, D. Musser, and S.-M. Liu. User-extensible simplification-type-based optimizer generators. In R. Wilhelm, editor, *Proc. 10th Int. Conf. on Compiler Construction (CC)*, volume 2027 of *LNCS*, pages 86–101. Springer, 2001.
→ 2 citations on page: 22

C. Schwarzweller. *Towards formal support for generic programming*. Habilitation thesis, Fakultät für Informatik, Universität Tübingen, 2003.
→ 1 citation on page: 5

P. Sewell, F. Z. Nardelli, S. Owens, G. Peskine, T. Ridge, S. Sarkar, and R. Strniša. Ott: effective tool support for the working semanticist. In *Proc. ACM SIGPLAN Int. Conf. on Functional Programming (ICFP)*, pages 1–12. ACM, 2007.
→ 3 citations on 3 pages: 38, 68, and 159

SGISTL. Standard Template Library Programmer's Guide. `http://www.sgi.com/tech/stl/`, Aug. 2008.
→ 5 citations on 3 pages: 3, 4, and 76

J. Siek. Improved iterator categories and requirements. Technical Report J16/01-0011 = WG21 N1297, ISO/IEC JTC1/SC22/WG21 - C++, Mar. 2001.
→ 2 citations on 2 pages: 4 and 82

J. Siek and A. Lumsdaine. Concept checking: Binding parametric polymorphism in C++. In *Proc. 1st Workshop on C++ Template Programming*, Erfurt, Germany, 2000.
→ 1 citation on page: 22

J. Siek and A. Lumsdaine. Essential language support for generic programming. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, pages 73–84. ACM Press, 2005.
→ 7 citations on 6 pages: 5, 6, 67, 73, 91, and 114

J. Siek, L.-Q. Lee, and A. Lumsdaine. *The Boost Graph Library: User Guide and Reference Manual*. Addison-Wesley, 2002.
→ 5 citations on 5 pages: 1, 3, 21, 91, and 147

J. Siek, D. Abrahams, and T. Witt. New iterator concepts. Technical Report N1640=04-0080, ISO/IEC JTC1/SC22/WG21 - C++, Apr. 2004.
→ 4 citations on 4 pages: 5, 74, 82, and 83

J. Siek, D. Gregor, R. Garcia, J. Willcock, J. Järvi, and A. Lumsdaine. Concepts for C++ 0x. Technical Report N1758=05-0018, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, Jan. 2005a.
→ 1 citation on page: 74

J. G. Siek and W. Taha. A semantic analysis of C++ templates. In *Proc. European Conf. on Object-Oriented Programming (ECOOP)*, LNCS, pages 304–327. Springer, 2006.
→ 1 citation on page: 67

J. G. Siek, D. Gregor, R. Garcia, J. Willcock, J. Järvi, and A. Lumsdaine. Concepts for C++0x. Technical Report N1758=05-0018, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, 2005b.
→ 1 citation on page: 66

A. A. Stepanov and M. Lee. The Standard Template Library. Technical Report HPL-94-34, Hewlett-Packard Laboratories, May 1994 (revised in October 1995 as tech. rep. HPL-95-11).
→ 6 citations on 6 pages: 2, 67, 73, 91, 145, and 147

M. Stoerzer and J. Graf. Using pointcut delta analysis to support evolution of aspect-oriented software. In *Proc. 21st Int. Conf. on Software Maintenance (ICSM)*, pages 653–656, 2005.
→ 2 citations on 2 pages: 25 and 75

R. Strniša, P. Sewell, and M. Parkinson. The Java module system: core design and semantic definition. In *Proc. 22nd ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, and Applications (OOPSLA)*, pages 499–514. ACM, 2007.
→ 1 citation on page: 68

B. Stroustrup. *The Design and Evolution of C++* . Addison-Wesley, 1994.
→ 2 citations on page: 7

B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 3rd edition, 1997.
→ 2 citations on 2 pages: 1 and 2

B. Stroustrup. 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++, Oct. 2003.
→ 1 citation on page: 66

B. Stroustrup and G. D. Reis. A concept design (rev.1). Technical Report N1782=05-0042, ISO/IEC JTC1/SC22/WG21 - C++, Apr. 2005.
→ 1 citation on page: 74

H. Sutter. `vector<bool>` is nonconforming, and forces optimization choice. Technical Report J16/99-0008 = WG21 N1185, ISO/IEC JTC1/SC22/WG21 - C++, Feb. 1999.
→ 3 citations on 2 pages: 4 and 82

X. Tang and J. Järvi. Concept-based optimization. In *Proc. ACM SIGPLAN Symp. on Library-Centric Software Design (LCSD)*, 2007. To appear.
→ 3 citations on 2 pages: 22 and 67

A. Tarlecki. Institutions: an abstract framework for formal specifications. In E. Astesiano, H.-J. Kreowski, and B. Krieg-Bruckner, editors, *Algebraic Foundations of System Specification*, pages 105–130. Springer, 1999.
→ 6 citations on page: 101

A. Tarlecki. Abstract specification theory: an overview. In M. Broy and M. Pizka, editors, *Models, Algebras, and Logics of Engineering Software*, volume 191 of *NATO Science Series—Computer and Systems Sciences*, pages 43–79. IOS Press, 2003.
→ 9 citations on 7 pages: 17, 18, 93, 100, 102, 111, and 112

P. Tonella. Using a concept lattice of decomposition slices for program understanding and impact analysis. *IEEE Trans. Software Eng.*, 29(6):495–509, June 2003.
→ 2 citations on 2 pages: 25 and 74

W. Tracz. LILEANNA: a parameterized programming language. In *Proc. 2nd Int. Workshop on Software Reusability*, pages 66–78, Lucca, Italy, Mar. 1993.
→ 2 citations on 2 pages: 20 and 27

R. Turver and M. Munro. An early impact analysis technique for software maintenance. *Journal of Software Maintenance*, 6(1):35–52, 1994.
→ 1 citation on page: 75

J. Visser. Visitor combination and traversal control. In *Proc. 16th ACM SIGPLAN Conf. on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 270–282, 2001.
→ 1 citation on page: 147

P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *Proc. 16th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL)*, pages 60–76. ACM Press, 1989.
→ 4 citations on 4 pages: 21, 91, 119, and 122

J. Walter and M. Koch. The Boost μBLAS Library. `http://www.boost.org/libs/numeric`, Sept. 2008.
→ 1 citation on page: 3

C. Wang and D. R. Musser. Dynamic verification of C++ generic algorithms. *IEEE Trans. Softw. Eng.*, 23(5):314–323, 1997. ISSN 0098-5589.
→ 3 citations on 2 pages: 21 and 93

D. Wasserrab, T. Nipkow, G. Snelting, and F. Tip. An operational semantics and type safety proof for multiple inheritance in C++. In *Proc. 21st ACM SIGPLAN Conf. on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 345–362. ACM, 2006.
→ 1 citation on page: 67

J. Willcock, J. Järvi, A. Lumsdaine, and D. Musser. A formalization of concepts for generic programming. In *Concepts: a Linguistic Foundation of Generic Programming at Adobe Tech Summit*. Adobe Systems, 2004.
→ 7 citations on 5 pages: 5, 6, 67, 73, and 114

E. Winch. Heterogeneous lists of named objects. In *Second Workshop on C++ Template Programming*, Oct. 2001.
→ 2 citations on 2 pages: 143 and 144

M. Wirsing. Algebraic specification. In J. V. Leeuwen, editor, *Handbook of Theoretical Computer Science*, chapter 13, pages 677–788. Elsevier, 1990.
→ 2 citations on 2 pages: 17 and 114

A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Inf. Comput.*, 115(1):38–94, 1994.
→ 1 citation on page: 69

M. Zalewski. A semantic definition of separate type checking in C++ with concepts—abstract syntax and complete semantic definition. Technical Report 2008:12, Department of Computer Science and Engineering, Chalmers University, 2008.
→ 3 citations on 3 pages: 39, 40, and 46

M. Zalewski and S. Schupp. Change impact analysis for generic libraries. *Proc. 22nd IEEE Int. Conf. on Software Maintenance (ICSM)*, pages 35–44, Sept. 2006.
→ 2 citations on 2 pages: 67 and 92

M. Zalewski and S. Schupp. C++ concepts as institutions. a specification view on concepts. In *Proc. ACM SIGPLAN Symp. on Library-Centric Software Design (LCSD)*, 2007. To appear.
→ 1 citation on page: 67

M. Zalewski and S. Schupp. A semantic definition of separate type checking in C++ with concepts. *Journal of Object Technology*, 2009. Accepted.
→ 1 citation on page: 196

A. Zeller. Isolating cause-effect chains from computer programs. In *Proc. 10th ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE)*, pages 1–10. ACM Press, 2002.
→ 1 citation on page: 76

J. Zhao. Change impact analysis for aspect-oriented software evolution. In *Proc. of the Int. Workshop on Principles of Softw. Evolution (IWPSE)*, pages 108–112, New

York, NY, USA, 2002. ACM Press. ISBN 1-58113-545-9. doi: http://doi.acm.org/10.1145/512035.512060.
→ 1 citation on page: 25