Towards Formal Support for Generic Programming

Habilitationsschrift
der Fakultät für Informatik
der Eberhard-Karls-Universität Tübingen

für das Fach Informatik

vorgelegt von
Christoph Schwarzweller

Tübingen, Januar 2003

Contents

1 Introduction **5**

2 Generic Programming **11**
2.1 Overview 11
2.2 Overloading and Polymorphism 13
2.3 Genericity in Programming Languages 18
2.4 Genericity in Computer Algebra 24

3 Generic Libraries **31**
3.1 Introduction 31
3.2 The Standard Template Library 33
3.3 The Boost Library 36
3.4 The Loki Library 40
3.5 The GILF Library 43

4 Representing Requirements **45**
4.1 Introduction 45
4.2 Algebraic Specification Languages 48
4.3 Concepts: Tecton 54
4.4 A Properties-based Approach 61
4.5 Algorithmic Requirements 66

5 Mechanized Reasoning Systems **77**
5.1 Introduction 77
5.2 The Imps System 81
5.3 The Mizar System 85
5.4 The PVS System 90
5.5 The Theorema System 94

6 Applications **99**
6.1 Overview 99

6.2 A Calculus for Deducing Properties 100
6.3 Generic Type Checking 105
6.4 Verification of Generic Algorithms 111
6.5 Design of Libraries 119

7 Conclusion **125**

Bibliography **127**

Chapter 1

Introduction

One of the main purpose of computer science is to develop software. Or, to be more precise, to design and develop methods, programming languages, tools, and software that support application programmers in developing their software. Today's goals of software construction, however, are not restricted to realizing a particular behaviour efficiently. Of course the development of efficient algorithms is still and will stay one prevailing goal, but there are other aspects obtaining more and more importance if high-quality software is striven for [Som01, Mey97]. First, of course, correctness of software is always a subject matter. Due to software employed in safety-critical applications, the desire for doubtless demonstration of correctness has grown over the last decades. Second, software should be developed so that it can be easily adapted and, hence, used for a number of applications. This prevents users from rewriting large parts of a software, or even writing new software, if another, related application is to be realized. Along with this goes the development and the supply of software libraries. A programming language for which commensurable libraries exist is much more convenient for users. They can fall back on the software contained in a library and need not develop their software completely from scratch. In this sense special purpose libraries such for example for sorting and searching or graph algorithms are of particular interest. Both can be seen in the context of reusability of software or program code. A last point we like to mention in some sense is also connected with reusing software. The intention that a piece software is to be applied and refined by a number of different users should be taken into account. In other words using the software should be as clear and easy as possible. This calls for the design of appropriate interfaces for accessing the software and algorithms. Because users often come with strongly varying prerequisites this can lead, in the extreme, to different interfaces for different kinds of users.

Generic programming [JLM00, MDS01, BJJM99] is a growing area that aims at developing high-quality reusable software and software libraries, thus addresses the above mentioned issues. The overall goal of generic programming is to provide methods that allow "to express algorithms and data structures in a broadly adaptable, interoperable form that allows their direct use in software construction" [JLM00]. To achieve this goal generic programming introduces new abstractions: the main observation is that an algorithm's method is often independent of the data structure or its representation being used. An easy example are sequence algorithms; here it does not matter which kind of objects is stored and, to some extent, not even how sequences are represented. Thus generalized algorithms should be provided abstracting from these details and allowing to plug in different data structures, representations or even algorithms. Then the user can fill in the abstract parts according to his needs, in this way adapting and reusing the generic software.

However, it is obviously not completely arbitrary what can be plugged in. Sorting algorithms for example expect that the elements of a sequence can be compared. This leads to the notion of requirements [MSL00]: the code plugged in must come with a number of properties so that the resulting algorithm works properly. In addition, as mentioned above, users should be equipped with efficient algorithms. This raises the question of how far algorithms should be generalized. Often it is possible to "overgeneralize" an algorithm. For example, searching in sequences can be done if the elements can be accessed one after the other only. However, efficient searching, in particular binary searching, needs to access element in the middle of a sequence quickly. Thus in some sense there are two contradicting demands for generic programming: on the one hand algorithms should be generalized as much as possible, that is come with rather weak requirements, so that they are applicable for a wide range of situations. On the other hand the resulting algorithms when instantiated by the user should still be efficient which calls for rather strong requirements ruling out non-efficient solutions. As a consequence developing and using generic algorithms appropriately requires deep inside into the algorithms' underlying methods.

The application of formal methods, that is the use of languages, methods and tools based on rigorous mathematical semantics, in software development has a long tradition. Since the seminal papers [GTWW77] on algebraic specification of data types and [Hoa69] on program correctness the benefits of formal specification and verification have been recognized and numerous languages and tools have been developed using elements from logic, model theory, universal algebra, and category theory [EM85, AKK99, LS87]. A for-

mal specification is a precise mathematically-based description of a system's behaviour. This not only aids in implementing a system, for example by refining the specification towards a program. Formal specification also supports prototyping for early testing of a system's behaviour as well as communication with non-expert customers. In addition, a mathematical description allows to formally prove properties of software; though time-consuming and most often not performed, a complete verification of software is possible in principle. Nevertheless trying to prove correctness can help in finding subtle errors in a software at an early stage. This goes along with the development of high-performance mechanized reasoning systems [Rob01], that is theorem provers and proof checkers. These allow to express and prove necessary theorems with machine assistance, in this way automating the process to a certain extent. Thus formal methods are a means of supporting the development of more reliable software and software systems, despite the growing complexity of software construction.

Consequently, it seems appropriate to apply formal methods to generic programming in order to enhance reliability of generic algorithms and libraries. Correctness of generic algorithms should be formally specified and verified. Here, correctness is understood in a broad sense: it means validity of various properties of generic algorithms and their instances. Note that in general validity of properties depends on the requirements attached to a generic algorithm. In addition, in the field of generic programming the user himself has to deal with some kind of correctness also: the question whether an instantiation is legal for a generic algorithm, that is whether it fulfills the requirements attached, cannot be answered in advance, especially if users themselves provide code to be plugged in. Thus formal requirements and their applications should be made visible to users in order to support them in applying generic algorithms correctly and efficiently.

This thesis aims at inspecting the field of generic programming with respect to both existing and possible future methods and tools supporting formal specification and verification in this area. As mentioned above generic programming introduces new abstractions; thus formal methods are of interest to safely handle the development and the application of generic algorithms. Due to the level of abstraction generic algorithms come with two different facets can be identified. First formal methods and tools for the verification of generic algorithms are of interest. This means proving that if an instance comes with the properties required by a generic algorithm, the resulting instance meets the algorithm's specification. Note however that generic algorithms are usually written for a (possibly infinite) number of applications which actually rules out testing as a "verification method". This

should be reflected in the specification and verification process, that is generic algorithms should be verified for all these applications at once, independent of the particular case in which they are applied. This leads to the question of what requirements are necessary in order that a generic algorithm works correctly for a class of instantiations and how this can be expressed in a formal manner. Second, using generic algorithms can lead to errors even if the algorithm is correct: the intended application may not fall into the ones the generic algorithm can handle correctly. Formal support should also be given to check whether a generic algorithm is applicable in a particular situation and, the other way round, information should be provided to the user if a generic algorithm is not applicable. Thus in generic programming formal methods not only aid in developing correct algorithms provided for the user, but also supports him in correctly applying generic algorithms.

The thesis is organized as follows. In chapter 2 we give an introduction to the field of generic programming. We discuss the ideas and goals of the field and how they contribute to the problems of software development mentioned above. Then we present different facets of generic programming found in the literature focussing on features of programming languages such as overloading and polymorphism. Afterwards we give an overview of techniques for generic programming that can be found in recent programming languages such as e.g. C++ or Haskell. Then we discuss the necessities and possibilities of generic programming in the area of computer algebra. We show that the demands on generic programming are much more involved in this area, in particular semantic properties of the operations involved have to be taken into account. This is due to mathematical results, both definitions and theorems, the development of algebraic algorithms is based on. This leads to the observation that generic algorithms come with both syntactical and semantic requirements on possible instantiations.

Much effort has been spent over the last years to develop libraries with a high degree of reusability, most of them based on the programming language C++ using new techniques such as for example template parameters. Therefore chapter 3 is devoted to libraries of generic algorithms. The ultimate goal of a library is to provide the user with support for developing algorithms and applications. Thus it is quite natural to use generic algorithms because they offer a high degree of reusability. On the other hand due to the requirements generic algorithms come with using the algorithms is more error-prone than in conventional libraries. Thus a well-designed generic library should also come with support to apply and instantiate generic algorithms contained. We present some existing C++ libraries—the Standard

Template Library, excerpts from the Boost Library, the Loki Library, and the GILF Library—and demonstrate to what extent they use and support the paradigms of generic programming according to chapter 2. Thereby we particularly address the role of a library user.

How semantic requirements can be adequately represented for the purpose of generic programming is the topic of chapter 4. We start with a rationale that the description of generic requirements corresponds to considering a set of operations together with associated requirements as an algebra. A natural starting point to describe such requirements thus is the use of algebraic specification languages allowing to exactly state domains and their properties, which are then briefly reviewed. However, generic programming not only calls for such an exact representation of requirements, but also for using them in order to check for valid instantiation. This has been adopted in the concept description language Tecton, in which classes of algebras, called concepts, are specified heavily reusing already existing concept descriptions. Finally we present an approach focussing on the properties of operations, which is therefore well-suited to support checking of generic instantiation. The last section deals with algorithmic requirements, that is with properties of algorithms not concerning their correctness. The need to specify such requirements, too, is discussed and examples how this could be done are given.

The main outcome of a formal specification is the possibility to employ provers or proof checkers to verify properties of specifications and their connected algorithms. Therefore we investigate in chapter 5 mechanized reasoning systems with respect to their possibilities to support generic programming based on properties. We briefly discuss particular characteristics mechanized reasoning systems should meet in order to support generic programming. We in particular take into account the use of the corresponding libraries and the profile of users being more interested in applying theorems for generic programming than in theorem proving itself. Four mechanized reasoning systems—Imps, Mizar, PVS, and Theorema—are considered in detail.

Finally, Chapter 6 presents some applications of the properties-based approach in different areas of generic programming including type checking, verification of generic algorithms and the design of libraries. First, the ideas of chapter 4 are generalized so that, instead of simply checking for inclusion of properties, sets of properties can be deduced. For this a small calculus based on a rule set incorporating knowledge of the application domain is presented. Then generic type checking is discussed in detail: a small programming language is described allowing to state semantic properties of generic type parameters. This allows to check the legality of instantiations based on the calculus just mentioned. The verification of generic algorithms

shows that every instantiation fulfilling requirements attached to the algorithm is legal. Thereby the properties-based approach enables to relax these requirements in a flexible way. We present a case study on Euclid's algorithm for computing greatest common divisors. Finally we briefly explain how the properties-based approach supports not only type checking of generic algorithms, but also the design of libraries, for instance mathematical libraries, by allocating them a more active role in maintaining knowledge. This is achieved by representing knowledge not with respect to domains but based on properties. Then again deduction takes place in order to validate theorems for particular domains.

Chapter 2

Generic Programming

In this chapter we give an introduction to the field of generic programming. We briefly outline the main ideas and goals of this field and give an overview of the techniques for generic programming and their realization in today's programming languages. This includes basic techniques like for example overloading and type coercion as well as more involved ones like (bounded) polymorphism. The inspection of genericity in the area of computer algebra finally leads to the emphasizing of semantic requirements of generic algorithms and their instantiations as a major concern of generic programming.

2.1 Overview

Generic programming is a discipline of computer science that aims at developing methods to write and organize algorithms that are broadly adaptable and interoperable, thus allowing for a maximum of reuse. The main observation is that most algorithms contain parts that are not specific for the behaviour of the algorithm. In other words, the algorithm's underlying method works well for a whole class of specializations, which means that a generic algorithm essentially works for a number of different types of its arguments. For example, sorting a sequence of elements does not depend on whether integers or real numbers or even characters are sorted. Similarly, the way a sequence of elements is stored is of no effect to the method implemented in sorting algorithms, and even the specific order a sorting algorithm works with can be seen as a non-necessary detail of sorting a sequence of elements. Therefore, generic programming introduces another level of abstraction into programming: compared with ordinary algorithms generic algorithms are much more abstract, they only include what is an essential ingredient of the

algorithm's underlying method leaving the rest to be completed later. Thus generic algorithms can be considered as algorithm schemes. Then a generic algorithm should work for every instantiation, that is for every completion of the scheme. This means in particular that a great part of program code can be reused for different types.

The identification of these generic parts leads to the expression of algorithms with minimal assumptions about data abstractions: generic programming deals with finding abstract representations of algorithms with respect to minimal requirements that make such an algorithm work. This includes in particular requirements a possible instantiation has to meet in order to be considered legal. Therefore generic programming is also called *requirement oriented programming* [MSL00].

Requirements of generic algorithms can serve different purposes. Though requirements of course have to ensure the correctness of a generic algorithm and its instances, there are other matters to bear in mind. Consideration of efficiency is a major concern of generic programming. Providing an algorithm that can be correctly used for a large class of specializations is only one side: inside such a class there may be special cases allowing for more efficient solutions. It is not desirable to bring in general application at the expense of an efficiency loss. Different algorithms should be provided for these cases, that is generic programming has to deal with the fact that there may be more than one algorithm to solve a problem. Consider the problem of searching an element in a sequence. Though it is possible to obey the same generic algorithm for unsorted and sorted sequences, this is not appropriate. The additional knowledge that a sequence is sorted leads to a much better way of searching. Therefore requirements for generic algorithms have to be carefully identified so that the efficiency of a method is kept. This means that different generic algorithms should be made available, if a problem can be solved more efficiently in some special cases.

Putting it the other way round: starting with well-known non-generic algorithms a goal of generic programming is to lift these algorithms, that is to identify the class for which the method of the algorithm works [Sch96]. This essentially includes paying attention to whether during generalization the properties responsible for the efficiency of the non-generic algorithm are not lost.

These considerations pose new problems and questions: requirements have to be identified and represented in order to make clear for which class of specialization a generic algorithm is suited. Also, if a generic algorithm is to be instantiated, the question is whether the actual instantiation meets

the requirements connected with the algorithm. Note again that this need not be restricted to the correctness of the instance, but also may concern its efficiency. On the other hand, given an instantiation the question is which generic algorithm should be chosen to solve a particular problem for which there is more than one algorithm available. Note that this in fact implies that there may be not only different generic algorithms for the same problem, but also generic algorithms bearing the same name. Then based on properties the instantiation obeys a decision has to be made. And, last but not least, all this should be organized in forms of generic libraries, so that users are supported when using generic algorithms.

In the literature the term generic programming has been used in a number of different meanings, programming with generic parameters, programming with polymorphic functions, programming with parameterized components, programming by abstraction, or programming with requirements, to name a few. In the rest of this chapter we discuss some of these techniques and clarify how they contribute to our view of generic programming. Genericity in computer algebra is also treated as the problems occurring there are the main motivation of this work.

2.2 Overloading and Polymorphism

The goals generic programming is based on are by no means new. Also techniques enabling current programming languages to handle aspects of generic programming were introduced decades ago. Especially the idea that the same algorithm can work for different types of its arguments was already addressed in the late 1960s. This phenomenon is called *polymorphism* and is in fact the basis of generic programming. In this section we give a short overview of the most important occurrences of polymorphism that can be found in the literature and discuss their impact on generic programming. Thereby a major point of concern is the question of whether the same program code can be used for different types.

Strachey [Str67] already distinguished between two cases of polymorphism. First, an algorithm can work on different types though the behaviour of the algorithm for these different types is not related at all. This means in particular that the code realizing the different variants of the function may be completely different, hence not the same code is used for different types. Strachey refers to this kind of polymorphism as *ad-hoc polymorphism*.

Today we would rather call it overloading as described below. Ad-hoc polymorphism or overloading is in fact a pure syntactic criterion: the same name is used for different algorithms with different argument types. Renaming the function for each combination of argument types allows to eliminate the polymorphism completely and this is what compilers internally usually do.

The second kind of polymorphism, called *parametric polymorphism* in [Str67], appears when an algorithm uniformly behaves on a range of types. Here the idea is that the algorithm actually does the same on all types in this range, except for some details from which can be abstracted away, nowadays usually by employing a type parameter. The name parametric polymorphism stems from the fact that already Strachey thought of type parameters—though in [Str67] he did not use this phrase—which he illustrated with a polymorphic mapping function for lists of arbitrary element type.

In some sense parametric polymorphism is a better way of generic programming as the same code is used to work with different types. Note that, in principle, parametric polymorphism works for infinitely many cases: because a parametric function does the same work independent of its arguments' types, the same code can be used together with code connected to the parameter. In contrast ad-hoc polymorphism is restricted to finitely many cases: each time an additional meaning is given to the algorithm, the complete code realizing this meaning has to be provided. Nevertheless ad-hoc polymorphism is important for generic programming as it allows to deal with the fact that one has usually more than one algorithm realizing a given function.

The view on polymorphism has been further refined by Cardelli and Wegner [CW85]. They split both ad-hoc polymorphism and parametric polymorphism as mentioned above into two facets. According to [CW85] ad-hoc polymorphism comes in two occurrences. *Overloading* corresponds to what Strachey called ad-hoc polymorphism: the same name is used to denote different algorithms and the context, that is the arguments' types, decides what algorithm is applied. Thereby the code for each variant of the algorithm is completely separated from the others. For example, + can denote the addition of integers and real numbers and the conjunction of boolean values as well as the concatenation of strings.

The second kind of ad-hoc polymorphism is *coercion*. Coercion means that the type of a given argument is transformed into a type fitting to the argument type of an algorithm, that is arguments may have more than one type. In other words, not the algorithm works on different types, but the arguments may have more than one type, one of which must match the argument type of the algorithm. Thus coercion allows the user to omit type

conversions necessary to apply a function with fixed argument types.

Both overloading and coercion do not affect the way an algorithm is implemented. In both cases the code of the algorithm itself is not reusable for different types: for overloading different pieces of code get the same name, which as already mentioned above is a purely syntactical phenomenon. Coercion, in contrast, is a semantic operation: the type of an argument is converted, which clearly is not possible for each pair of types. However, when coercion takes place, always the same code for the function is executed, the arguments are transformed so that they fit to the given code. So, here we have no code working for more than one type, thus it is reasonable to subsume them as *ad-hoc polymorphism* as done in [CW85].

Though overloading and coercion do not support reusable code for different types directly, they, nevertheless, are necessary for generic programming languages. Consider for example the addition of numbers, where numbers may be of type `integers` and of type `real`. Then, all of the following four expressions should be legal.

$$\begin{array}{lcl} 3 & + & 4 \\ 3.0 & + & 4 \\ 3 & + & 4.0 \\ 3.0 & + & 4.0 \end{array}$$

There are several possibilities to do so: pure overloading for each of the four cases, pure coercion implementing addition for the real numbers only or a mixture of coercion and overloading where addition is overloaded for integers and real numbers and the other cases are handled via coercion to the real number case. This has been already pointed out in [CW85] and overloading and coercion are rather obvious features of today's programming languages. From the view of generic programming the key point is that there may be several algorithms for the same operation, addition of numbers in the example. This gives the possibility to provide more efficient algorithms if more knowledge of the parameters is present without explicitly stating which one is used. In our example though, if coercion is present, having only one algorithm for addition of real numbers is enough to handle addition of integers and real numbers, a pure integer addition algorithm is probably more efficient and should therefore be used. This is one essential goal of generic programming and would not be possible without overloading.

The second category of polymorphism presented in [CW85] is *universal polymorphism*, which allows to provide code working for more than one type of arguments. It includes parametric polymorphism as explained above:

parametric polymorphism occurs when an algorithm has an explicit or implicit type parameter. Here, the same code is executed for each application given by a realization of the type parameter. In other words, the same code can be applied to a number of types without changes such as e.g. coercions. Typical examples are lists, trees or vectors over arbitrary element type, that is containers for arbitrary elements. We will see in the next section how this idea has been adopted and extended for the design of the Standard Template Library [MDS01].

The other kind of universal polymorphism is *inclusion polymorphism*. Here, similar to coercion, arguments may have different types. However, these types need not be disjoint, so that in contrast to coercion types they do not have to be transformed. Consequently, the same code can be used for different types, which is the reason for considering inclusion polymorphism as a special case of universal polymorphism. Inclusion polymorphism is typical for object-oriented languages. Here, elements of a subclass can be used as arguments for functions defined in the superclass. This means, that a function can be called not only with arguments having the given type, but also with all elements having a subtype of this type without any transformation. Therefore inclusion polymorphism is sometimes also called *subtype polymorphism* [CE00].

Parametric polymorphism is often identified with generic programming; in fact the term *generic function* is used in [CW85] for functions exhibiting parametric polymorphism. However, our point of view is somewhat different: generic programming aims at providing algorithms in a more general form that allows to apply them to a wider range of applications. Parametric polymorphism is one technique that can be used to do so. However, other kinds of polymorphism as indicated by the examples given above also contribute to achieve the goals of generic programming.

In particular, subtype polymorphism allows to provide different algorithms for the same polymorphic function: consider for example polygons and their specialization rectangles. An algorithm for computing the perimeter of polygons can be used to compute the perimeter of rectangles because `rectangle` will be a subtype of `polygon`. However, a more efficient algorithm computing the perimeter of rectangles can be provided for the subtype `rectangle` which is automatically used depending on the argument's type. In contrast to coercion the programmer itself can add new algorithms by introducing new subtypes as we will see in the next section. Note that again overloading is necessary to enable the use of the same name for both algorithms.

Nevertheless, parametric polymorphism is the most involved form of polymorphism as it allows to define functions operating on different types without requiring a relation between these types: a type parameter in fact implicitly states that the part being instantiated should provide both data and some functions operating on that data. Thus a type parameter can be considered as a description of an abstract data type, hence parametric polymorphism allows to develop algorithms working for the whole class of realizations of an abstract data type. Consider, for example, the following piece of C++-style code.

```
template<class T>
T minimum(T x, T y) {
   return x < y ? x : y;
}
```

Here, `T` is the type parameter indicating that the function `minimum` works for arguments of arbitrary type. However, this is not really true: to execute `minimum` a function `<` defined for the actual instantiation of the type `T` is necessary. Unfortunately, this is only specified in the code where `<` is used. The possibility to provide type parameters with properties a type must have to be used as an instantiation, is known as *bounded polymorphism* [AC96, CCH$^+$89] or *constrained genericity* [Mey97].

Properties a type parameter should provide can be roughly considered at two levels: First, functions operating on the elements of the type have to be present, as `<` in the example above. However, this is not enough to ensure that the instance of the algorithm behaves as expected, the instantiated part must behave in a particular way. In other words, in the above example `x < y` for elements `x` and `y` of type `T` must be true if and only if `x` indeed is less than `y`. Thus there is a second level of properties concerning the semantics of the instantiated functions. This nicely fits to viewing type parameters as descriptions of abstract data types: an abstract data type consists of a signature corresponding to the first, syntactic level and a set of axioms corresponding to the second, semantic level. We believe that such semantic requirements of type parameters are crucial for generic programming. This topic will become more important if we consider generic programming in the field of computer algebra.

2.3 Genericity in Programming Languages

In the last section we have presented different facets of polymorphism and discussed their impact on generic programming. In this section we investigate how and to what extent these approaches can be found in recent programming languages. As already stated overloading and coercion have made their way into today's programming languages. Languages like e.g. C++ [Str97], Java [AG97], Lisp [Ste90], or Haskell [Tho99], to name only some of the most used ones, provide overloading and coercion at least for basic data types like numbers. More involved examples of overloading and coercion will be discussed in the next section about computer algebra. Here we concentrate therefore on universal polymorphism, that is parametric, inclusion, and bounded polymorphism.

Object-oriented languages such as e.g. Smalltalk [GR83], Eiffel [Mey92], and to some extend C++ [Str97] and Java [AG97] are the most prominent examples for languages supporting subtype polymorphism, that is inclusion polymorphism restricted to subtypes. Inclusion polymorphism is realized via subclasses: classes are in fact descriptions of new types the user can introduce and afterwards use like built-in types. Thus subclasses derived from a superclass correspond to subtypes. They inherit data and algorithms from its superclass, hence algorithms defined in the superclass are applicable to elements of all its subclasses.

Using the technique of overriding it is also possible to provide different more efficient algorithms for the same problem for subclasses: overriding or redefinition in subclasses allows for replacing the algorithm of the base class with another one. Note that this can be done by the user himself by simply defining a new subclass in which the more efficient algorithm is provided. Consider for example a class `polygon` in which there is an algorithm to compute the area of polygons. For rectangles this can be done much better, so the user defines a new class `rectangles` which is a successor of `polygons`. In the subclass `rectangles` the better algorithm overriding the one in the superclass `polygon` can be defined. However, redefinition of algorithms is the responsibility of the user. Object-oriented programming languages do not check[1] whether the more special algorithm does not change the semantics, that is whether it is correct or it indeed computes the same function as the general algorithm given in the base class.

[1] Eiffel is in some sense an exception as it uses assertions to express pre- and postconditions of algorithms.

The first programming language to include parametric polymorphism was ML [Mil84, Pau96]. ML is a typed functional language based on a type calculus by Milner [Mil78]. This calculus allows to infer the type of expressions, thus provides strong typing without the need of explicitly stating expressions' types. More important from the view of generic programming is that ML types may include type parameters, thus allow to define polymorphic functions. The major example are functions on lists with arbitrary element type such as computing the length of or reversing a list. However, this does not work if operations on the elements are involved, that is if a type parameter is considered as a data type with operations, and not only as a set of elements. The reason is that ML does not allow overloading of functions.[2] So for example, the type of

```
fun sum []      = 0
  | sum (x::xs) = x + sum xs;
```

evaluates to `int list -> int`; in other words `sum` is no polymorphic function. As a consequence `sum([1.2,3.4]);` results in a type error although `1.2 + 3.4` is a valid ML-expression of type `real`.

This problem has been adressed in Haskell[Tho99]. In [WB89] type classes are introduced to enable certain kinds of overlaoding. A type class is a collection of operations with associated types, that is in some sense an abstract prototype for a type. For example, a type class `Num` for numbers can be defined as follows.

```
class Num a where
  +,*    ::  a -> a -> a
  negate ::  a -> a
  0      ::  -> a
```

Then a type belongs to the type class `Num` if it provides functions named `+, *, negate` and `0` of the appropriate types declared by `Num`. Now, if the type of `sum` is derived, Haskell identifies `sum`'s addition with the one in `Num` and uses the type information of the type class as a precondition, that is in Haskell the type of `sum` evaluates to `Num a => [a] -> a`. This means, that for every type `a` belonging to `Num` the type of `sum` results in `[a] -> a`. Instances of `Num` can be built for the integers and reals by providing appropriate binding for the operations occuring in `Num`. This allows to use `sum`

[2] Exceptions are addition and multiplication which are overloaded for `int` and `real`, as well as a few other built-in operators.

to compute sums of lists over both integers and reals, that is in particular `sum([1.2,3.4])` results in `4.6`.

Another interesting approach—also called generic programming—not mentioned so far was realized using functional programming languages: *polytypic programming* [JJ96]. Polytypic programming is a generalization of how functions are defined in abstract data types. There, functions are defined by induction on the constructors of a set of data. A polytypic function is a function that is defined by induction on the structure of data types, that is by induction on type constructors. Consider for example a function `fmap` which gets a function `f` as input and is expected to apply `f` to all elements of the collection given as a second argument. This collection may be a list or a tree or something else. The types of lists and trees, for instance, can be described as follows.

```
List = () + Par x Rec
Tree = Par + Rec x Rec
```

Here, `()` denotes the empty product, `Par` gives an element of the underlying element type and `Rec` is a recursive type constructor. An algorithm defined by case distinction for all the involved type constructors can now be used to handle both lists and trees and every other type that may be generated using these type constructors only. Polytypic programming has been implemented as PolyP [JJ97], an extension of a subset of the programming language Haskell [Tho99].

Polytypic programming allows to provide code working for different types, in fact for a whole class of types given by a set of type constructors. However, the approach is quite different from our view of generic programming: generic programming looks for properties making an algorithm work, that is type parameters are descriptions of abstract data types. In contrast, polytypic programming operates on a fixed set of type constructors. Nevertheless polytypic programming allows to write flexible, reusable code as has been elaborated in particular in [Hin00].

C++ [Str97] probably is the most prominent programming language enabling parametric polymorphism. We already used C++-like code for the maximum example in the last section. C++ feature templates to introduce type parameters based on which algorithms can be written. Note that though not explicitly stated such a type parameter can include operations to be used. This shows again, that a type parameter in fact stand for a collection of data and functions operating on that data, thus for an abstract data type. Maybe

even more important from the view of generic programming is that using templates not only functions but also classes can be parameterized. For example

```
template<class T>
class List {
   ⋮
  T sum(List& l);
   ⋮
};
```

introduces a class for lists over elements of arbitrary type `T`, which among others contains an algorithm `sum` to accumaulate the elements of a list. Inside the class list operations are written with respect to the type parameter `T`. This allows to implement list operations independent of the element type, the same code can be used for all instantiations of the parameter `T`. For example,

```
List<int> l;
List<float> l;
List<my_elements> l;
```

declare lists over integers, reals, and a user-defined type, respectively. However, this will compile only if the instantiation fulfills all the requirements implicitly postulated by the operations in the list class. In our example elements of type `T` must be summable due to algorithm `sum`, that is the class being instantiated for `T` must come with an operator `+`.

In the original design the Java programming language [AG97] did not include type parameters, that is parametric polymorphism. However, as already mentioned Java provides inclusion polymorphism with which some cases of parametric polymorphism can be simulated [CE00] although this is quite cumbersome as it requires wrapping classes and casting of types.

Extensions of Java including parametric polymorphism have been provided, e.g. GJ [BOSW98]. GJ allows the use of type parameters for Java classes, thus introducing parametric polymorphism. GJ code is translated back to Java code. In fact, GJ provides bounded polymorphism using an interface technique: interfaces describe functions that must be present in every class implementing such an interface, for example

```
interface Collection<A> {
  public void add (A x);
  public void Iterator<A> iterator();
};

interface Comparable<A> {
   public int compareTo (A x);
};
```

The interface `Collection` defines some kind of container by requiring an operation `add` and an `iterator`. Note that the interfaces are parameterized with a parameter `A` and that `Iterator<A>` in fact is another interface. Now, GJ allows to state that classes are realizations of interfaces, e.g.

```
class Byte implements Comparable<Byte> {
   ⋮
};

class Collections {
 public static <A implements Comparable<A>>
  A max (Collection<A> xs) {
     ⋮
  };
};
```

Then, the class `Byte` has to provide implementations of at least the functions stated by the interface `Comparable`, that is of the function `compareTo`. A class `List` could be for example an implementation of the second interface `Collection`. In the class `Collections` a static member function `max` is defined. To call `max` successfully the instance of the type parameter `A` must provide a method to compare elements of type `A`. This is ensured by extending the type parameter `A` to `<A implements Comparable<A>>`. This means that only those instances `I` are legal for `A` which in fact are an implementation of `Comparable<I>`, and hence provide a compare algorithm.[3] For instance, `Byte` can be used as a realization of `A` as it implements `Comparable`.

Other programming languages providing bounded polymorphism are Eiffel [Mey92], Ada [Bar95], and CLU [Lis92], the first programming language to

[3]Note the recursive use of the type parameter `A`. This is sometimes called *F-bounded polymorphism* [CCH+89].

include constrained polymorphism. Based on work on OBJ [GM97] Goguen developed a theory for specifying libraries of parameterized modules for Ada, which is known as *parameterized programming* [Gog96]. Similar work has been done for C++. For the development of the Boost Graph Library [SLL02] *concept checking* was employed. Concept checking consists of providing template classes, so-called concept-checking classes [SL00b], in which valid expressions for the instantiations are listed. These classes are then instantiated with the instantiations of the type parameters, thus checking at compile time whether all necessary functions are present. In particular, this leads to much better compiler-generated error messages in case the algorithm's requirements are not matched.

Another approach has been adopted in the language SuchThat [SL00a]. The generic programming language SuchThat [SL00a], originally designed for generic computer algebra algorithms, combines a concept base and a programming language: concepts are abstract descriptions of domains and are expressed in the concept description language Tecton [Mus98]. Algorithms are then written for functions described in concepts using the programming language Aldes [LC92], that is algorithms are written over domains given by concept descriptions. As concepts in fact describe minimal requirements of domains SuchThat allows for bounded polymorphism.

To include an algorithm the user has to define a concept describing the properties necessary for the algorithm or a lifted version of it. As a consequence, the algorithm can be instantiated with every domain that is a realization of this concept. The Tecton language allows for concept inclusion and stating lemmas about concept implications. These can be used to check whether a given domain is a realization of a concept, thus whether a generic algorithm written for a concept can be instantiated with this domain.

The programming language Views [Gas01] goes even further: it extends types by type classes and considers generic parameters of algorithms as type classes. This means that a type `T` is a legal instantiation of a generic algorithm if `T` is an element of the type class describing the algorithm's generic parameter. Then a resolution-based type checker is used to prove that a given type is in a type class, that is whether a generic algorithm can be instantiated with this type.

The realization of bounded polymorphism is important for generic programming as it supports the user in checking whether a particular instantiation fits to a generic algorithm. In [CW95, Cen95] this has been called semantic typing. Here, a calculus for checking semantic properties of instantiations of parameterized algebraic specifications is developed. This is done

in two steps: first a calculus of implementation is presented which allows to check for signatures, thus syntactic requirements. This calculus is then extended to capture semantic requirements by introducing so-called conditions which are interpreted as model class inclusion.

This again shows that the above problem comes with two facets. First, as has been already mentioned above, one has to check whether necessary operations are provided by an instantiation. But this only ensures that the instance of the algorithm is executable, it does not ensure that the instance of the algorithm will behave as expected, that is the instance is correct. The reason is that usually a generic algorithm in addition expects a particular behaviour of the functions being instantiated. In other words, a generic algorithm considers its type parameter as a description of a data type that includes the semantics of the operations. This will become even more important in the next section where we consider a special application area of genericity and generic programming—computer algebra.

2.4 Genericity in Computer Algebra

Computer algebra deals with the development and the implementation of algorithms for solving mathematically formulated problems [GKW02, BCL83]. The basis thereby is a symbolic, exact, and finite representation of the involved finite or infinite mathematical objects, which distinguishes computer algebra from numerical computation. The use of structural mathematical knowledge during the design is one of the major characteristics of computer algebra. This means in particular that theorems about mathematical domains are incorporated leading also to investigations about mathematical structures in general. For example, the question of how greatest common divisors can be computed in polynomial rings over the integers or the real numbers has yielded the notion of pseudo division that can be carried out in every commutative ring. The use of abstract mathematical structures in computer algebra naturally gives raise to genericity by abstracting away from a particular domain. For example, algorithms for polynomial addition and multiplication in principle work well for every polynomial ring no matter which coefficient domain is actually considered. Thus parametric polymorphism is inherent in computer algebra. However, as we will see, parametric polymorphism is not the only kind of polymorphism naturally occurring. In the following section we investigate the appearance of the phenomena described in section 2.2 in the field of computer algebra.

Overloading is omnipresent in computer algebra, just because the operations of mathematical structures themselves are heavily overloaded. The same symbol `*`, for instance, is used for the multiplication in numbers, rings, and vector spaces. In vector spaces it even denotes both the multiplication of vectors and the multiplication of a scalar and a vector. Sometimes `*` is also used for the concatenation in free monoids as well as for conjunction in Boolean algebras. Another example is the symbol `1` used for units in various algebraic structures, for true in Boolean algebras and sometimes for the top element in lattice theory. It would be quite confusing to force a different notation depending on the structure under consideration.

Also coercion plays an important role. In fact, the example concerning integers and real numbers given in section 2.2 occurs in computer algebra, too. However coercion can become much more complicated as in the case of numbers. For instance, an element of an integral domain `I` can be considered as an element of the field of fractions `FF(I)` over `I`; this again can be coerced into a polynomial over `FF(I)`. Also it should be noted that `FF(FF(I))` is isomorphic to `FF(I)`, and thus elements of the former can be naturally considered as elements of the later. The other way round a constant polynomial can be understood as an element of the coefficient domain.

Subtyping in order to reuse algorithms defined for a supertype is a natural feature in computer algebra. This is due to the implicit hierarchy mathematical structures come with. For example a field is a Euclidean domain, a Euclidean domain is an integral domain and so on. In fact, this can be extended even for structures with different sets of operations: a ring, for instance, is a group and a group is a semi group that is a field in particular is a semi group. All these domains provide a set of operations the more specialized ones rely on. In other words, algorithms written for semi groups should be applicable in fields.

The type system of AXIOM [JS92] allows for defining categories which resemble the hierarchy of mathematical structures. Thus it is possible to inherit algorithms from one domain to another. That a special domain is an element of a category is by assertion, that is the user states for example that the integers are a Euclidean domain. Operations and algorithms of the hierarchy can then be used for the integers; AXIOM checks whether this is legal by inspecting the whole hierarchy. However, as category membership is by assertion, it is not guaranteed that a special domain fulfills semantic requirements of a category. These are given in the documentation only, so that the user bears responsibility for that. Thus, for instance, a Boolean algebra could be claimed to be an element of the category of rings though this is certainly not the case.

As already mentioned, parametric polymorphism is quite natural in computer algebra, because algebraic algorithms are based on properties that correspond to or follow from the axioms of mathematical structures. Thus methods developed work well for each realization of such a mathematical structure. However, requirements posed on the instantiation of a generic algebraic algorithm are much more involved. In fact, for such an algorithm an instance is correct only if it fulfills the axioms of the structure the algorithms is based on.

Let us start with a rather pathological, nevertheless illustrating example. Consider again a generic sort algorithm which includes a type parameter `T` for the elements to be sorted. As already mentioned in the last section, to make the algorithm executable the instantiation of `T` has to come with a realization of the operation $\leq$. But this is not sufficient to make the algorithm behave as we expected. Assume that the type parameter `T` is instantiated with the integers and an algorithm that implements the binary predicate $\leq$ on the integers given by $i \leq j$ for all i and j being integers. Clearly, the in this way instantiated generic algorithm will run properly, however given a sequence s of integers, the result of running the algorithm with input s will be s itself. This is not what we expect of sorting integers. But note, that s is a correct answer with respect to our particular choice of $\leq$: trivially, s is a permutation of s and—assuming that s is indexed by integers—for i and j with $i < j$ we have $s_i \leq s_j$, that is s is sorted with respect to $\leq$. The problem is that the relation $\leq$ we have chosen is no order: it does not fulfill antisymmetry, that is $i \leq j$ and $j \leq i$ does not imply $i = j$. In other words, the algorithm will behave as expected only if some semantic requirements on the operation instantiated for $\leq$ are fulfilled. In our example, the relation should be an order—and in addition total to ensure that every sequence can be ordered. Thus the type parameter `T` in fact stands for a mathematical structure $(\mathtt{T}, \leq)$ where $\leq$ is a total order on `T`. This what in [Mus98] is called a concept and can be compared to an abstract data type with loose semantics: only an algebra that fulfills the requirements can serve as a legal instantiation. On the other hand, each algebra that fulfills the requirements is considered as a legal instantiation, which provides the great flexibility of generic programming.

In the area of computer algebra the situation just described is the normal case: algorithms are written for mathematical structures given by a set of axioms, such as fields or polynomial rings. Hence generic algorithms only work as expected if the instantiation indeed is a realization of the underlying structure, that is if it fulfills the axioms required. In addition requirements here are much more involved than in ordinary programming. This is naturally

due to the use of mathematical structures. The same holds for the realization of more efficient algorithms. Efficient algorithms in computer algebra often take advantage of properties specific to a domain. Hence, it is both not easy to generalize such algorithms to generic ones and to check whether a particular instantiation meets these properties. We will see some examples later in this section.

The problem again can be seen in terms of executability and correctness: the changeover from executability to correctness consists of checking whether the operations necessary to ensure executability fulfill some implicit requirements. In computer algebra these requirements are much more involved thus much harder to handle. In other words, the distance between executabilty and correctness is much greater here.

Let us illustrate these issues in detail for multiplication of polynomials. An algorithm for doing so is straightforward, and is not the main concern here. The point is that the algorithm can be naturally parameterized by the underlying coefficient domain. Hence, a generic multiplication algorithm for polynomials is executable if the instantiation of the type parameter provides addition and multiplication of coefficients. However, the instance only computes polynomial multiplication in the usual sense if the instantiation fulfills the axioms of a ring which is much more involved than the axioms of an order. So to say, it is quite safe to use the generic algorithm with obvious instantiations such as the integers or the rational numbers, but using more elaborated structures it is hard to check whether this is legal.

Furthermore, there is quite a number of rings with additional properties: commutative rings, integral domains, principal ideal domains, and fields to name a few basic ones. A generic polynomial multiplication algorithm works correctly for all of them. However, considering a closely related problem, namely the evaluation of polynomials, things change. Obviously, polynomials can be evaluated if the coefficients form a ring. But to establish polynomial evaluation in the mathematical sense, that is as a homomorphism from the ring of polynomials into the coefficient domain, this domain has to be commutative with respect to multiplication. Hence, the class of mathematical structures that may be legally instantiated, may change even if two generic algorithms handle the same class of objects, rings of polynomials in the example.

Another example showing the demands of genericity in computer algebra is the computation of greatest common divisors. The well-known Euclidean algorithm can be used to compute greatest common divisors for integers, and in fact for every Euclidean domain. Given two elements of a Euclidean

domain, the idea is to use division with remainder recursively until one of the input elements equals one so that the other one is the greatest common divisor. Again it quite easy to formulate a generic algorithm working for arbitrary Euclidean domains. However, there are two points of concern. First, what are possible instantiations of such an algorithm? The answer is not straightforward: although there are greatest common divisor algorithms for polynomial rings based on division, that is although the syntactical requirements are fulfilled, this does not lead to a legal instance of the generic Euclidean algorithm. Polynomial rings are not Euclidean in general, thus do not fit to the algorithm's semantic requirements. Second, the efficiency of the Euclidean algorithm may strongly vary for different domains as it is mainly given by the effort spent for division with remainder. Thus the question is to what extent the generic Euclidean algorithm works efficiently for Euclidean domains, or whether different versions should be provided for special Euclidean domains.

Efficiency in computer algebra also strongly depends on the representation of mathematical objects. Thinking again of polynomials, there are so-called sparse and dense representations. Often algorithms are efficient for only one representation of polynomials. Writing efficient generic algorithms in this case thus includes stating requirements on the representation of polynomials. In other words, considered as a description of a concept the generic type parameter `T` does not only include properties of the algebraic structure but also representation details.

The integers modulo n where n is an integer with $n > 1$, denoted by $\mathbb{Z}_n$, are another interesting example. $\mathbb{Z}_n$ is a field if and only if n prime, and algorithms for computing the inverse in such a field are well-known, e.g. based on the extended Euclidean algorithm or Fermat's theorem. Of course the methods are independent of the particular value of n, so these algorithms can be considered as generic for $\mathbb{Z}_n$ with respect to the parameter n. However, again one has to be a bit careful. If n is not prime, $\mathbb{Z}_n$ is no field, indeed not even an integral domain as there are zero divisors. Hence, there are elements having no inverse that is although each instance of the generic inversion algorithm computes something, in some cases the result is not what was expected in the sense that the properties of this result do not fit to the algorithm's description. In other words, the generic inversion algorithm is executable for every instantiation of the parameter n, but correct only if the instantiation fulfills the additional requirement of being a prime number. Note that in this case the requirement is not on the domain, but on a particular element that is instantiated.

Another problem arising in computer algebra is the combination of dif-

ferent concepts such as for example combining rings with orders. Consider again the integers modulo p, where p is prime. The question is, can $\mathbb{Z}_p$ be processed by generic algorithms that include an order as a generic parameter? From a technical point of view the answer is simply "yes", as $\mathbb{Z}_p$ can be considered as the subset $\{0, 1, \ldots, p-1\}$ of the integers, so an order for the integers can be used. On the other hand, from an algebraic point of view it is not that simple: it depends on the specification of the algorithm. If the algorithm expects a domain with an order only then we are in a secure position. But if the algorithm is specified for ordered rings the answer is "no", because $\mathbb{Z}_p$ as a finite field does not allow for orders that are compatible with addition and multiplication.

The key point is again that in all examples the correctness of the instance of an algorithm depends on implicit semantic requirements on the used operations. So, as already said, parameters in generic algorithms should be considered as implicit descriptions of data types: they describe a concept, e.g. a set of operations together with properties the operations have to fulfill, and only those instantiations are legal that fit to this concept. In computer algebra the properties of the operations are a major concern, in fact developing computer algebra algorithms mainly consists of taking advantage of these properties. Hence, genericity in computer algebra much more depends on the semantic requirements of the parameters and therefore much more calls for checking semantic requirements during instantiation.

Chapter 3

Generic Libraries

As we have seen in chapter 2 generic programming aims at providing algorithms that are flexible and easily adaptable to particular applications. This goal is strongly connected with the design and development of libraries. Libraries are the connection between algorithm developers and users. Thus it is not only important how powerful and efficient the algorithms contained in a library are, but also how the user can apply and rely on these algorithms. Genericity can contribute to both.

In this chapter we deal with libraries based on generic programming techniques. Recently work on designing libraries with techniques from generic programming is mostly done based on the programming language C++. We therefore consider three C++ libraries—the Standard Template Library, the Boost Library, and the Loki Library—in detail. We explain the underlying concepts and techniques of these libraries and discuss the impact on users.

3.1 Introduction

Libraries are a major concern in software development. Based on programming languages that well-support intended application areas in both developing programs and efficiency, libraries provide a collection of algorithms the user can build upon. Thus a library together with its underlying programming language serves two purposes. First, a library provides algorithms and data structures so that a user need not start from scratch with the implementation of basic methods. For example, Leda [MN99] and CGAL [FGK+00] supporting geometric computing come with the necessary data structure of graphs and a set of algorithms operating on them. Consequently, a user can focus on the geometric application he is interested in and need not again im-

plement basic graph algorithms. Second, it is not only important to provide a user with (implementations of) algorithms solving certain basic problems. The algorithms contained in a library should furthermore be easily and safely usable, that is incorporated in the user's program development. Thus reusing and combining algorithms is a major point in the development of libraries of algorithms which consequently should be supported to aid users.

Libraries for generic algorithms, therefore are not only collections of generic algorithms, but should also support the user in applying these algorithms. However, the application of generic algorithms introduces new challenges to the user. Generic algorithms have to be instantiated with predefined or self-written code. Thereby, as we have seen, the correctness of the resulting instance strongly depends on semantic properties of the instantiation. Consequently, in particular the instantiation process has to be addressed because here lies the main source for errors as the instantiated piece of code may not fulfill (semantic) requirements of the generic algorithm. The user has to be supported in checking these requirements when algorithms of a generic library are used. In case requirements are not met compilers should generate meaningful error messages [SL00b], or even better should indicate which requirements are not fulfilled by the actual instantiation.

Furthermore, generic programming aims at providing pieces of software for a wide range of applications. This means that algorithms stored in libraries should be both flexible and easily adaptable. Then the user can utilise the library algorithms as algorithm schemes that are in some sense to be refined for a particular application. This for example can be achieved not only by pure instantiation of generic algorithms but also based on iterators and adapters [MDS01] or by providing implementations of design patterns [GHJV95].

In addition, as we have argued especially in the field of computer algebra, in generic programming there is usually more than one algorithm leading to a correct instance. These will differ according to their efficiency and other algorithmic requirements. Though it seems hardly possible to automate the optimal choice completely, we believe that a library of generic algorithms should support the user in deciding which instantiation to choose. This may range from including the specification of properties important for such a decision up to automatically choosing an instantiation based on a knowledge base or heuristics [Kre02].

In the following we present some existing generic libraries. As a matter of fact most work on designing and implementing libraries based on topics from generic programming is done using C++ using and exploring the capabilities of templates. Thus we decided to present three C++ libraries, the Standard

Template Library [MDS01], parts of the Boost Library [SLL02, Jär02] and the Loki Library [Ale01]. We give a brief overview of the main concepts and discuss how these libraries use techniques of generic programming in order to provide more than a collection of algorithms. We analyze in particular how the idioms used support particular applications of users. We close with a new approach based on [Wei03] focussing on the distribution of generic libraries where an XML-based intermediate representation of generic algorithms is used to embrace the demands of generic libraries.

3.2 The Standard Template Library

The Standard Template Library [SL94, MDS01] provides basic algorithms and data structures, mainly for sorting and searching in `list`s, `vector`s, `map`s, and other structures. The outstanding contribution of the Standard Template Library (STL) is the way algorithms and data structures are presented: The STL introduces the concepts of iterators and containers. The goal was to represent algorithms as general as possible without affecting their efficiency. This was achieved by the use of iterators that are a generalization of pointers, that is they provide access and navigation to containers without being restricted to a particular type of container. Thus iterators are a technique to decouple algorithms and containers: Algorithms essentially work on iterators and each container provides iterators used to link algorithms to it. This allows to combine an algorithm with different containers such as e.g. lists, vectors, or maps. Furthermore the STL comes with an iterator hierarchy, that is with different kinds of iterators providing different sets of operations. By choosing an iterator kind for an algorithm thus properties a possible instantiation has to meet can be expressed. Consider for example the binary search algorithm as found in the STL.[1]

```
template<class ForwardIterator, class T>
bool binary_search(ForwardIterator first,
                   ForwardIterator last, const T& value);
```

Besides abstracting away from the type of the elements being searched by the template parameter `T`, this binary search algorithm can be instantiated with every class of type `ForwardIterator`, one of five different kinds of it-

[1] We use C++ code to illustrate STL concepts, because this is most familiar. However, in principle the STL is more a methodology to design a library that, in principle, can be implemented in other programming languages, too.

erators. `ForwardIterator` allows to access elements in a container one by one, but not to go back or to access elements by an index. Note that the type parameter `ForwardIterator` requires the existence of operations with certain properties, that is it again can be considered as a description of an abstract data type. Now, e.g. `list` and `vector` container classes as defined in the STL provide iterators of type `ForwardIterator`[2], hence the algorithm works for both of these containers. Iterators for containers can be accessed using member functions `begin` and `end` giving the range of the container in which its element are stored. So we can call `binary_search` as follows.

```
vector<int> v;
... initialization of v ...
bool binary_search(v.begin(),v.end(),3);
```

Note that in contrast to class instantiation the instantiation need not be given explicitly, it is inferred from the actual parameters' types. The same way the algorithm can be called using `list`s as containers. In fact users can also provide their own iterator classes; these will work well with the algorithm as long as the requirements of a `ForwardIterator` are fulfilled. However, as it stands it is not checked at compile time whether the class being instantiated indeed fits to the semantic `ForwardIterator` requirements. Note that the binary search algorithm of the STL is not a member of a class such as lists. This allows to consider an algorithm as a stand-alone entity which can be called with each triple of arguments where the first two are of type `ForwardIterator` and the third of type `T`. However, again the instantiation of `T` must provide a method to compare elements of type `T`.

The STL also adresses the problem of keeping efficiency while generalizing: A binary search algorithm usually requires logarithmic time. Though a version of binary searching works for iterators of type `ForwardIterator`, the time required is linear in the size of the container, although the number of comparisons is still logarithmic. The reason is that using forward iterators one cannot directly jump to the middle of a container. This operation is available only for iterators of type `RandomAccessIterator`. Now, if the binary search algorithm of the STL is called with such an iterator, the algorithm is indeed logarithmic in the size of the container [MDS01]. Thus the STL provides a more efficient binary search algorithm for the case that more knowledge about the parameter of the algorithm is present. This is one

[2]In fact, both lists and vectors provide iterators with even more properties than `ForwardIterator`, namely lists iterators of type `BiDirectionalIterator` and vectors iterators of type `RandomAccessIterator`.

essential goal of generic programming, realized in the STL by means of the iterator hierarchy.

Algorithm `binary_search` requires a `<`-operator with which elements of type `T` can be compared. Because this operator is not explicitly stated in the algorithm's declaration, solely the operator `<` given by the instantiation of the type parameter `T` is used. However sometimes it is useful to sort and search with a different order, for example one wants to switch from `<` to `>`. Therefore the STL provides another version of most algorithms with an additional template parameter to describe the order to be used.

```
template<class ForwardIterator, class T, class Compare>
bool binary_search(ForwardIterator first,
                   ForwardIterator last, const T& value,
                   Compare comp);
```

Note that overloading is employed to identify the different realizations of `binary_search`. Thus the user need not care about which version will be called. He only has to provide the arguments that shall be taken into account: To use the second version he just defines a class `MyCompare` overloading the function call operator `operator()`. Then this function object is passed to the `binary_search` algorithm to be used for comparisons.

```
vector<int> v;
... initialization of v ...
bool binary_search(v.begin(),v.end(),3,MyCompare());
```

STL containers store collections of objects and are divided into two categories: sequence containers and sorted associative containers. A container defines how objects are stored and provides a number of operations to access and manipulate the objects being stored. Information about the memory model, however, is encapsulated in an `Allocator` class. Thus every container comes with two template parameters, one for the type of the objects and one for the allocator class:

```
template <class T, class Allocator = allocator>
class vector;
{
  ...
};
```

Note the use of a default value for `Allocator` with the predefined STL class `allocator`. Thus STL containers can work in principle with different memory models provided by different allocator classes meeting a set of requirements. This means that on the one hand users can develop and apply their own memory model easily by plugging in such a class instead of using `allocator`. On the other hand as the STL comes with a default implementation of the `Allocator` class, no user is forced to deal with the memory model if he agrees to use `allocator`. This technique in fact can be considered as using a policy class for the memory model as it has been further elaborated in the Loki library (compare section 3.4).

The STL and, in particular its widely accepted implementation in C++, now being part of the C++ Standard Library [C++98], has inspired both further research in generic programming and the development of other libraries based on C++ templates and the iterator concept. For example, the Matrix Template Library [SL99] provides high-performance numerical linear algebra algorithms. Here, two-dimensional iterators are used: They allow to access rows or columns of a matrix which returns an ordinary iterator for dealing with single elements. Special adapters enable reusing the same algorithm in different situations. So, for example, there is a scaling adaptor, thus adding two vectors or a scaled vector and a vector in the MTL is basically the same.

Another example are graph algorithms. Here the underlying data structure is a bit more elaborated: In contrast to STL containers and matrices traversing a graph has to incorporate both edges and vertices. Note, that in particular a graph's adjacency structure has to be taken into account and that graphs may contain circles. Nevertheless it is possible to develop and implement a generic, iterator-based handling of graph algorithms. This have been done, for instance, in the Boost library.

3.3 The Boost Library

The Boost library [Boo02] is the ongoing outcome of an online community. Its goal is to provide portable high-quality C++ libraries working well with the C++ Standard Library. To this end submitted libraries pass through a review process before coming to a decision whether a library should be included in the Boost Library collection. From the viewpoint of generic programming there are two interesting sublibraries: The Boost Graph Library [SLL99, SLL02] implements a number of graph algorithms including

for instance Dijkstra's and Kruskal's algorithm based on templates and iterators similar to the STL. The Boost Lambda Library [JP00, Jär02] provides lambda abstraction for C++, an alternative to the use of function objects when instantiating generic C++ algorithms. In the following these libraries and the techniques used to implement them will be described in more detail.

The Boost Graph Library (BGL) provides both interfaces and their implementation for directed and undirected graphs. Edges and vertices are handled by descriptors with very basic functionality. Additional information can be attached and maintained to vertices and edges by means of so-called property maps. A property map associates values of a particular property with the set of vertex resp. edge descriptors. The BGL provides operations on property maps to set or access the information stored by such a property map. So, for example `get(p_map,key)` gives the value of `key` under the property map `p_map`. This allows to keep graph algorithms generic without shortening the information graphs may carry. Graph traversal is done using edge and vertices iterators based on the adjacency structure of graphs. For example the function

```
std::pair<out_edge_iterator, out_edge_iterator>
                                    out_edges(vertex, graph);
```

returns an iterator range containing the out-edges (for directed graphs) or incident edges (for undirected graphs) of the given vertex.

BGL algorithms provide formal parameters to adapt their behaviour to a particular application. This can be compared to the use of function object parameters in the STL. Here, however, visitors are used which in some sense are an extension of function objects: A visitor defines a number of functions that are called at event points inside the algorithms. Thus a visitor does not only define the behaviour of an algorithm's operator but can also add extra functionality to algorithms.

Two implementations of the graph interface are given. One is based on adjacency lists, the other on adjacency matrices. This reflects efficient handling of spare and dense graphs. Instantiating template parameters allows to maintain details of the implementation, for example which kind of container is used to store vertices and edges or whether a graph is directed or undirected.

Graphs have a number of associated types such as the type of edge and vertices descriptors and iterators or size types. These nested types have to be accessible in order to provide generic graph algorithms. Note that assuming

the graph class being instantiated provides all necessary type definitions is no proper solution. To access nested types, that is types associated with a graph type the BGL employs traits classes. Traits [Mye95, Ale00] are a method where classes containing type definitions only are used to infer type information at compile time. Traits have also been used in the SGI implementation of the STL to determine iterator categories [Aus99]. The idea is to use template specialization to create different type definitions according to the instantiation of template parameters.

```
template <class T>
struct graph_traits {
  typedef typename T::edges_size_type edges_size_type;
};
```

A user working with special graph representations not providing a definition of `edges_size_type` then defines a specialization of the `graph_traits` class in which the type is set accordingly. Note that specializing the `graph_traits` class also allows to use BGL graph algorithms with graph representations as defined in the Stanford GraphBase [Knu94] or the Leda library [MN99].

```
template <>
struct graph_traits<MyGraph> {
  typedef My_edge_size edges_size_type;
};
```

As a consequence generic algorithms need only work with `edges_size_type`. The following type definition guarantees that when instantiating `T` the right type is bound to `edges_size_type` due to instantiation of the `graph_traits` class.

```
typedef typename
      graph_traits<T>::edges_size_type edges_size_type;
```

Note that using partial template specialization it is not necessary to write a traits class for every special graph type. For example, a graph type parameterized by its edge and vertices types such as used in Leda results in a specialization of `graph_traits` with two template parameters `etype` and `vtype`. Thus traits are an appropriate method to encapsulate necessary information about nested types from generic algorithms.

In conjunction with this stands the macro `BOOST_STATIC_ASSERT` [Mad00]. It allows to attach requirements for constants and expressions to class templates. The idea is to use template instantiation to distinguish the positive

from the negative case. Consider for example a template class with a parameter `T` for which only instantiations where `T` is an integral type are expected. This can be expressed by defining a boolean variable `isIntegral` and including `BOOST_STATIC_ASSERT(T::isIntegral)` in the template's definition. Then for an instantiation `isIntegral` must evaluate to true, which in fact means that users writing the instantiation will define this way; otherwise a compile error will occur. The macro is implemented with the help of a class `STATIC_ASSERTION_FAILURE` with a parameter of type `bool`. The class simply does nothing if instantiated with `true`. The case of `false`, however, is not defined which leads to the compile error mentioned. Note that due to this implementation a compiler returns an error message in which the class name `STATIC_ASSERTION_FAILURE` occurs, that is the message indicates that some requirement is not met.

As already mentioned in the last section, function objects are used to modify the behaviour of generic algorithms, for example changing the comparison operator used in search algorithms. However, `struct`s overloading the function call operator have to be defined in order to obtain these function objects. The Boost Lambda Library (BLL) adds some kind of lambda abstraction to C++, thus allowing to create unnamed functions that can be passed directly to the algorithms. The library was designed and implemented so that it in particular works with the C++ Standard Library [C++98]. Thus, for example, let `foo` be a function mapping from `int` to `int`. Then the following code using containers from the Standard Template Library compiles well.

```
vector<int> v;
... initialization of v ...
for_each(v.begin(), v.end(), free1 = bind(foo,free1));
for_each(v.begin(), v.end(), cout << free1 << endl);
```

The last parameter of `for_each` takes an expression that may contain free variables. Then during the loop this expression is evaluated for each element of `v`. Thus in the first loop each element `i` of `v` is replaced by `foo(i)`. The second loop outputs the content of `v` separated by line breaks.

Free variables are introduced by placeholder variables having predefined names, `free1` in the example. Thus the use of placeholders implicitly converts an expression into a lambda expression. `bind` is a BLL function used to turn function calls into lambda expressions, in this way invoking function `foo` with the actual value of `free1`. As the example shows also operators like `=` and `<<` have been overloaded for lambda expressions.

Lambda expressions are implemented using expression templates [Vel95a]: Operators are overloaded to create objects representing the expression and its arguments. In this way the evaluation is delayed so that expression objects can be manipulated, in particular the actual arguments are substituted for the placeholder objects. Note, that at least part of these manipulations can be done at compile time. To summarize the BLL provides a mechanism to include lambda expression into C++ making the use of function objects and binders to adapt STL algorithms unnecessary.

3.4 The Loki Library

Loki [Ale01] pursues a slightly different goal than the libraries presented so far. The goal is not solely to provide a library of generic algorithms but to provide the user with pieces of software that are on the one hand adapted to particular problems and their solutions, however on the other hand general enough so that they can be used as a framework in which only code for details of applications has to be filled in. This is done by implementing a number of generic design patterns [GHJV95] in a way so that the user can easily plug-in the application-connected code into the framework given by Loki. This can be considered as putting genericity onto another level in the sense that not algorithms are generalized, but a set of algorithms together generalizes a solution strategy. This means that the instances the user has to provide are not solely algorithms but can also be some kind of module realizing a particular aspect of the desired application. However, generic programming, here the extensive use of C++ template parameters, is used to implement design patterns in Loki. In the following we present the underlying concepts of this approach that we believe are most important as techniques for generic programming.

The main idea in [Ale01] is to use so-called policy classes as the basis of the design. Using policy-based class design isolates different aspects of the system: A policy class focuses on one behavioral or structural aspect only. Examples are the creation of objects of a type `T` or memory management strategies depending on a type `T` or the length of objects considered. Classes that use a number of policy classes—called host classes—thus are highly adaptable with respect to the aspects covered by the policies: The user just plugs in implementations of policy classes; the overall behaviour of the host class is not affected.

The crucial point is that a policy in fact establishes an interface for the aspect concerned. As a consequence policies can be implemented in various ways emphasizing different specific aspects. As long as the implementation fits to the interface, it can be used in the host class. In other words a policy class can be considered as a generic parameter of the importing host class. This allows the user to decide which strategy he wants to apply. This may be one of the strategies predefined in the library or even a new one for which the user has written a policy class on his own. Note that a policy class can offer more operations than required by the interface so that a user can extend his policy with further functionality.

The idea of policy classes can be easily realized using template template parameters. Consider for example an application in which objects have to be created. This is formulated using a template template parameter `CreatorPolicy`. Instances of `CreatorPolicy` are to define a creation strategy for the objects being instantiated via the template parameter `T`. Then `Application` inherits `CreatorPolicy<T>`; thus the application can create objects based on application data given by the class `T`.

```
template <class T, template <class> class CreatorPolicy
                                          = DefaultCreator>
class Application : public CreatorPolicy<T>
{
  ...
};
```

Then the library user can define his personal application as an instance of `Application`. All he needs to do is to provide `Application` with a class matching the interface of `CreatorPolicy`. This can be one of the policy classes given by the Loki library or a self-written class `MyCreator`. The type parameter `CreatorPolicy` is instantiated with this policy class and inherited by `Application`. Thus the instance of `Application` can create objects according to the strategy given in `MyCreator`. Note the use of partial template specialization in the first example to concretize how objects are created while leaving the objects in class `T` itself still arbitrary.

```
template <class T>
class Application<T, MyCreator>
{
  ...
};
```

```
typedef Application<Int, MyCreator> MyApplication;
```

However, the user is not forced to deal with creation of objects. If a user is not interested in a particular creation strategy he can ignore the template parameter `CreatorPolicy`. This is due to the default value `DefaultCreator` given in the definition of `Application`. Thus, if there is a class `Polynomial` matching `Application`'s requirements on objects of type `T`, the following defines an application for polynomials where polynomials are created following the strategy the library developer has provided as default in the policy class `DefaultCreator`.

```
typedef Application<Polynomial> PolyApplication;
```

Providing more than one policy class in the interface of an application thus allows to abstract from various aspects and strategies used in the application. The user then has the freedom to use particular implementations of these policies in his application. In this way the user can easily configure an application to his needs without even being forced to look to the inside of `Application`. This is achieved by combining template template parameters and inheritance.

Based on policy classes Loki implements, for instance, a generic interface for the abstract factory design pattern [GHJV95]. An abstract factory supports the creation of a family of related objects to be used together. Loki's `AbstractFactory` class template realizes this by providing a template template parameter `Unit` for creating objects; again a default class `AbstractFactoryUnit` is given. To build up a factory, that is a collection of objects determined by the user, typelists are used. Typelists allow for parameterizing a class by a mutable number of types. Type lists and their operations are implemented using recursive template classes and partial template instantiation. As a consequence such computations take place during compile time. This technique is called compile time programming or template metaprogramming [Vel95b, Unr94].

Now, given a typelist holding the types of the objects to be contained in a factory `AbstractFactory` instantiates the instantiation of its second parameter `Unit` with every type in the list. In particular a class hierarchy is generated in which the abstract factory inherits from every class constructed this way. Thus the user automatically gets a uniform interface for the objects of his factory for which implementations can be provided with the help of the `ConcretFactory` class template. Again the user has the possibility to chose between different policy classes for actually creating concrete objects. To conclude with, other patterns, policies and techniques implemented in

the Loki library include small object allocation, smart pointers, singletons, visitors, and multimethods; for details see [Ale01].

3.5 The GILF Library

So far we have concentrated on methodologies and techniques present generic libraries are designed and implemented with. However, there is a another aspect of generic libraries that completely differs from non-generic libraries—their distribution. The standard way of distributing a library is to provide a collection of already compiled object files. These are then linked to the users' application to get a running program. Unfortunately, this concept does not work if generic algorithms come into play. Generic algorithms are non-executable schemes of which parts are left for the user to complete by instantiation. For a generic iterator-based STL algorithm, for example, the memory layout is not known until the user provides an instantiation. In addition operators used may end up in a function call or machine code instructions depending on the instantiation. Thus application programs relying on generic algorithms need to compile these again in order to incorporate properties of actual instantiations. As a consequence object files for generic algorithms cannot be provided at this point of time and the library as a whole must be distributed with the complete source code.

This problem is addressed in [Wei03]. The observation is that the design of common compilation systems aims at the generation of machine level code; this level, however, is not appropriate for generic algorithms which due to their inherent abstraction until instantiation represent a much higher level. Taking a somewhat more abstract view a compiler transforms a source program in some intermediate representation distributed to be executed on different machines. For non generic algorithms object files are an appropriate representation, for generic ones not. Consequently, in [Wei03] a new intermediate representation—called Generic Interface Link Format (GILF)—has been developed in which generic programs should be compiled before being distributed as a library. GILF is an abstract representation of programs that can be compared to a parse tree. Like a parse tree a GILF program represents an analyzed version of the source program on which transformations can be easily performed. In particular, information on the appropriateness of instantiations is included. Therefore when given an instantiation on the actual machine, code for an executable program can be easily generated from a Gilf program.

Note that this in fact results in new definition of front- and back-end. The front-end now performs lexical, syntactic and semantic analysis of the source code and generates a GILF representation. Code generation however is delegated to the back-end, in addition to linking and loading the final machine code. Nevertheless a GILF representation of generic algorithm can be considered as compiled because, in particular, the time-consuming analysis of generic programs has not to be performed on the actual machine again.

To implement a prototype of such a compilation system a version of GILF—XGILF—has been defined using the Extended Markup Language XML [BPSM00]. The structure of an XGILF document is organized in compilation units consisting of an import, a declare, a define, a bind, a store, and an extend section. Thus a compilation units contains all information about a particular piece of code necessary to generate a running instance of the represented code and can be seen as a modularization of the XGILF library.

As of this writing the XGILF core library contains descriptions of basic data structures together with corresponding basic algorithms such as Boolean, Integers, and other machine types as well as for example arrays. In addition some general purpose function and type signatures are provided.

However, the GILF intermediate representation is in principle language-independent, that is it can be seen as an interface for describing precompiled generic algorithms. Thus generic programs from various generic programming languages can be compiled into GILF and then distributed together as a library of generic algorithms.

Chapter 4

Representing Requirements

In the last chapters we have seen that generic programming relies on the abstraction of data requirements. Examples, in particular from the area of computer algebra, have shown that type parameters used to implement this abstraction should be understood as a description of an abstract data type: Types describe data and operations as well as (minimal) requirements on these operations. This means in particular semantic ones, that is properties the operations have to fulfill in order to make the instance of the generic algorithm work properly. However, in most existing programming languages and libraries this aspect is not fully taken into concern. In this chapter we investigate how such requirements can be represented for generic programming and, in particular, for supporting generic type checking. We also address the description of algorithmic requirements.

4.1 Introduction

In generic programming type parameters are implicitly considered as descriptions of an abstract data type, realizations of which are legal instantiations of generic algorithms. That is generic type parameters describe a type class [WB89] in which types are collected that are accepted as a legal instantiation. Important for generic programming is, that descriptions of parameters should also include semantic requirements in order to filter out non-legal instantiations in the sense that though these instantiations provide all necessary operations, their operations do not behave as expected by the generic algorithm. Thus a description for both generic type parameters and possible instantiations is necessary emphasizing the need for checking whether an instantiation meets the requirements given by a generic type parameter.

In the following, this is made explicit. To do so we use a standard terminology from the area of algebraic specification: The semantics of generic type parameters T are interpreted as model classes of algebras $C(\mathtt{T})$. This allows for precise formulation not only of the semantic requirements of generic type parameters T, but also of the possible instantiations $\mathcal{A}$, so that legal instantiation can be described in forms of a predicate $Leg(\mathcal{A}, \mathtt{T})$. We may for now think of the description of a generic type parameter T given by a signature $Sig(\mathtt{T})$ giving carriers and operations and a set of first-order axioms $A(\mathtt{T})$ describing properties of the operations demanded by T. This implicitly defines a class $C(\mathtt{T})$ of algebras that provide the necessary signature $Sig(\mathtt{T})$ and whose operations fulfill the axiom set $A(\mathtt{T})$; thus we have

$$C(\mathtt{T}) := \{\ \mathcal{A}\ \mid Sig(\mathtt{T}) \subseteq \mathrm{S}ig(\mathcal{A})\ \wedge\ \mathcal{A} \models A(\mathtt{T})\ \}.$$

Note that an algebra $\mathcal{A} \in C(\mathtt{T})$ can provide more operations than given by the signature $Sig(\mathtt{T})$. This is quite natural as, for instance, a generic group algorithm should be allowed to be instantiated with a realization of a ring by just forgetting the multiplication of the ring.

Thus a type parameter T in a generic algorithm in fact represents the requirements given by the associated class $C(\mathtt{T})$, and instantiations have to meet these requirements to be legal. Now the instantiated piece of code P can be naturally interpreted as an implementation of an algebra $\mathcal{A}_P$: The set of implemented operations gives the signature, and the (known or expected) properties of the implemented operations constitute the set of axioms. Then P is a legal instantiation for the generic parameter T if and only if the algebra $\mathcal{A}_P$ is an element of the model class of T, thus we have

$$Leg(\mathcal{A}_P, \mathtt{T})\ :\Longleftrightarrow\ \mathcal{A}_P \in C(\mathtt{T}).$$

Note that the definition of the predicate includes in particular the semantic requirements represented by the class $C(\mathtt{T})$. Hence, if an instantiation is legal, the resulting instance of the generic algorithm will not only be executable, but also be correct, that is behave as expected. This can, as already mentioned, be compared to algebraic specifications with loose semantics [AKK99] or to concept descriptions [Mus98]. Subtyping, also necessary for coercion, naturally fits into this approach: A type T' is considered as a subtype of a type T, written $\mathtt{T}' \leq \mathtt{T}$, if every legal instantiation of T' is also legal for T, e.g.

$$\mathtt{T}'\ \leq\ \mathtt{T}\ \ :\Longleftrightarrow\ \ C(\mathtt{T}')\ \subseteq\ C(\mathtt{T}).$$

Thus we have, for example, $\texttt{Ring} \leq \texttt{Group}$. Note that $C(\texttt{Ring}) \subseteq C(\texttt{Group})$ holds due to the fact that an algebra's signature may contain more carriers and operations than given by the description of a type `T`. As a consequence algorithms with `Group` as a type parameter can be legally instantiated with elements of $C(\texttt{Ring})$. In addition types in the usual sense easily fit into this interpretation of generic type parameters: The type `integer`, for instance, can be seen as the type class $C(\texttt{Int})$ in which there are exactly the algebras having a restriction isomorphic to the integers, thus as a type class which basically consists of one algebra only. Then all algebras $\mathcal{A} \in C(\texttt{Int})$ in particular come with the properties of a ring; thus we have $C(\texttt{Int}) \subseteq C(\texttt{Ring})$ and hence $\texttt{Int} \leq \texttt{Ring} \leq \texttt{Group}$.

Thus we have a general framework for describing types and type parameters for generic programming. The framework is based on the notion of algebras thinking of an instantiated piece of code as a realization of an algebra. This allows to describe (minimal) requirements of a generic algorithm by means of giving a signature and a set of axioms specifying the necessary properties; thus in particular semantic requirements are included. Moreover, we have a precise formalization of what a correct instance of a generic algorithm is: An instantiation is correct if viewed as an algebra is an element of the class of algebras described by the generic parameter. Furthermore, the notion of subtype, necessary for coercion especially in the area of computer algebra, is both easy and flexible, it just corresponds to the subset relation of classes of algebras.

Hence, from a theoretical point of view types as classes of algebras are a quite elegant and straightforward description of generic type parameters and their instantiations. However, to include this approach into a generic programming language further issues have to be taken into account. First, of course, the description of generic type parameters `T` should be formal in the sense that proving using mechanized reasoning systems is enabled. This allows to infer properties of the type classes $C(\texttt{T})$ following from the description of `T` and also for generic algorithms themselves: Generic algorithms should be verified with respect to a generic type parameter `T`. This means proving that each piece of code P with $\mathcal{A}_P \in C(\texttt{T})$ can be instantiated resulting in an instance of the algorithm fulfilling the generic algorithm's specification. Finally, it should be proven that particular instantiations are legal, that is whether $\mathcal{A}_P \in C(\texttt{T})$ holds.

Second, as the check for legal instantiation should preferably take place during compilation, it is clear that proving correctness as just described cannot be done at compile time as a whole. This would slow done the compilation process. Thus a representation of type parameters `T` is necessary that on the

one hand is easy enough to enable fast compilation with some kind of instantiation check indicating that $\mathcal{A}_P \in C(\mathtt{T})$ holds. On the other hand it has to be formal enough to allow (off-line) verification of legal instantiation, that is proving that the instantiation check done at compile time is indeed correct, that is indeed implies $\mathcal{A}_P \in C(\mathtt{T})$.

Third, the description method should provide techniques allowing to build up a library of type classes $C(\mathtt{T})$. In particular the subtyping relation $\leq$ on type parameters `T` should be reflected by the library so that algebras $\mathcal{A}_P$ with $\mathcal{A}_P \in C(\mathtt{T}')$ and $\mathtt{T}' \leq \mathtt{T}$ can be easily coerced to $C(\mathtt{T})$. This in fact supports both, the introduction of new type classes $C(\mathtt{T})$ by a user and the instantiation check during compiling. A user only needs to include the new type class into the library at the level he is working on, coercing this class to more general ones then takes place automatically. Note again, that algebras of a type class $C(\mathtt{T}')$ are legal instantiations for all generic algorithms with type parameter `T` with `T'` $\leq$ `T`.

And finally, one should bear in mind that the user of a generic programming language intends to use algorithms for some application. That is, he should not be overwhelmed with formal specification or theorem proving before being able to compile programs. Indeed, a user should be able to just claim that algorithms or instantiations are correct to get programs compiled. Nevertheless, this should be done in a way allowing to later incorporate mechanized reasoning systems to check whether these claims are really true. Also, in the negative case, the user should be provided with information why the program does not compile, that is with hints why the instantiation is not considered legal by the compiler.

In the following sections we consider, starting with algebraic specification languages [SW99], how to represent (semantic) requirements of type parameters `T` in a generic programming language in order to meet the demands just outlined. We particularly take Tecton [KM92, Mus98] into consideration, a concept description language (re-)designed in view of specification of the Standard Template Library. Finally, we present an approach focussing on the properties of operations necessary for being a legal instantiation, thus focussing on checking for legal instantiation of generic algorithms.

4.2 Algebraic Specification Languages

Considering requirements of generic type parameters `T` as a class $C(\mathtt{T})$ of algebras, algebraic specification languages are the natural starting point for

a formal description of $C(\mathtt{T})$. We therefore briefly review the basics of algebraic specification languages. Thereby we in particular focus on techniques for structuring specifications as this is the key to build up a library of specifications in order to use them for generic instantiation checking, that is for checking whether $\mathcal{A}_P \in C(\mathtt{T}')$ holds.

An algebraic specification language combines several ingredients, among them constructs for specifying properties of individual components, structure mechanisms for building larger specifications in a modular way, a definition of the semantics and mechanisms for proving properties of specifications [SW99]. Basic components, that is flat specifications, are usually established by a signature and some kind of axioms, mostly first-order formulae. Out of these new specifications can be constructed using operators on specifications the specification language provides. The most basic structure mechanism is enrichment which allows to add more sorts or operations to a specification. Here, different forms can be distinguished depending on whether new sorts are introduced by the enrichment or whether the enrichment is a conservative extension, that is whether all models of the original specification are still models of the enrichment. The second most basic operation for building specifications incrementally is the union operator. It puts together two specifications which may share some subspecification. The semantics of union can be defined by pushouts based on signature morphisms at the specification level and by amalgamation at the model level; for equational logic these two are compatible [EM85]. Renaming of sorts and operations is needed, for example, to avoid confusion when putting together two specifications, and most specification languages also include a derive operator allowing to hide auxiliary sorts and operations used for the construction of a specification only. There exist quit a number of further operations but most of them can be reduced to a sequence of the operations just presented.

A key concept for structuring and reusing specification is the notion of parameterization, which actually follows the same idea as generic programming with type parameters. Several different approaches have been studied in the literature. The classical approach is to consider a parameterization as an enrichment, where the enriched part describes the admissible arguments for building instances. Again the semantics is defined via signature morphisms and pushouts [EM85]. The second well-known approach is more general by considering any expression over a specification variable as a parameterization. In [Wir86] a λ-notation is chosen, where the type of the specification variable corresponds to the formal parameter specification. In addition, there exist approaches based on the observation that any (adequately structured) specification can be considered parameterized without explicitly defining pa-

rameters [OSC89, Mus98]. Here, instantiation corresponds to substituting parts of a specification by another.

In the following we give a brief overview of the most important existing algebraic specification languages, see [AKK99] or [Wir95] for further details and languages.

Clear [BG77, BG80] was the first algebraic specification language having therefore a significant influence on the development of many other languages. Clear already provided specification-building operators for constructing specifications out of individual components as well as parameterization based on pushouts. *OBJ3* [GWM+92] is an executable specification language which can be seen as an implementation of a restricted version of Clear. OBJ3 is based on order-sorted conditional equational logic, and thus becomes executable employing associative-commutative rewriting. Order-sorted logic is used to handle partial functions, errors and subsort polymorphism. Individual components in OBJ3 come in two versions: There are so-called objects giving the initial semantics approach as well as theories giving loose semantics of specifications. Structuring of specifications is realized by importing both objects and theories. OBJ3 provides different kinds of imports with different meanings, for example that the resulting specification is a conservative extension of the original one. However, as it could require involved theorem proving, OBJ3 does not check whether these import declarations are correct, this is the responsibility of the user. In fact these declarations can be considered as decorated with proof obligations to be proven externally to ensure that the library of specifications indeed is correct.

ASL [SW83, Wir86] is a specification language with the intention to provide a clear and sound semantic basis for "defining more user-friendly higher-level specification languages" [AKK99]. Therefore it provides only a kernel language with very basic nevertheless powerful specification-building operations which can be used to define further operations in a semantically clear way. ASL is based on loose model-class semantics, that is specifications may denote several, non-isomorphic algebras. However, generating constraints are included using a specification-building operator called `reachable`. It includes, in contrast to Clear and OBJ3, parameterization based on λ-calculus. ASL's structuring mechanisms include, besides the basic ones mentioned above, an operation `observe` used to express behavioural abstraction. With respect to behavioural abstraction two algebras are considered equivalent if they cannot be distinguished by a predefined set of observations This has been used to improve the notion of implementation, see [ONS93] for an overview.

CASL [CoF98, Mos97] is a specification language that allows to describe both requirements and designs. It is intended to be the starting point for a family of specification languages; this includes simpler languages obtained by restrictions as well as more advanced languages obtained by extensions. CASL, attempting to initiate a standard specification language, combines what is considered as the most successful features of existing specification languages. Thus it supports both total and partial functions as well as predicates used for instance to handle errors. CASL provides a bunch of specification-building operations, among them not only the basic ones but also for instance translation, reduction, loose extension, and free extension, as well as parameterization based on pushouts. In addition CASL allows for so-called architectural specifications [BST99] which describe how to design a specified software from separately developed pieces with the help of specified interfaces. Thus CASL aims at not only supporting the specification phase, but the whole process of software development.

Specware^{TM} [SJ95, SM96] is both a language and a system for formally specifying and developing software focussing on specification refinements. To this end the notion of morphism is extended to the notion of interpretation. Roughly speaking an interpretation from a specification A to a specification B is a morphism from B into an enrichment of A, where the enrichment is explicitly stated. This allows to construct models of the source specification from models of the target specification by taking reducts after they have been expanded along the enrichment. Thus interpretations are a suitable notion of refinement. When sufficiently refined, specifications can be converted into programs based on the theory of logic morphisms [Mes89]. Certain requirements on a specification must be met so that the specification can be translated following a predefined morphism into the programming language Lisp. By providing further morphisms different programming languages can be incorporated.

Larch [GH93] in fact stands for a family of specification languages. A Larch specification consists of two parts: One specific for a chosen programming language, written in the so-called Larch interface language, and another common for all programming languages, written in the so-called Larch shared language. The Larch shared language is a loose algebraic specification language based on equational axioms to build individual components, called traits. Equational theories are strengthened by providing generating constraints and some structuring operators; in particular Larch traits can import other traits, so that enrichment is available. Other structuring operators are renaming and parameterization. In addition, there is a clause

`partitioned by` indicating that all distinct values of a sort can be distinguished using just the mentioned operators. Furthermore using the keyword `implies` assertions can be stated, that is theorems that are intended to hold. Larch interface languages have been realized for a number of programming languages, among them Ada, Smalltalk and C++.

Extended ML [KST94, ST91] is designed as a framework for the formal development of programs in the Standard ML programming language [Pau96]. It extends (a large subset of) Standard ML by permitting axioms in module interfaces, in this way describing required properties for module components. This allows to attach both input and output interfaces to Standard ML functions. Thus the semantics of Extended ML is based on the semantics of Standard ML, a loose model class based semantics to interpret the specification part of the language is added. Hence, the relationship between Extended and Standard ML is more formal than the rather informal one between the shared and interface language in Larch.

Because the major advantage of formal specification is the possibility to establish correctness in a rigorous sense, theorem proving techniques and proof systems for algebraic specification have been extensively studied. Proof methods for algebraic specifications aim to support proving logical consequences of a specification, that is formulae that hold in all or particular models of a specification. This serves for learning more about the system being specified as well as for controlling whether the specification meets the intended properties. Proving in algebraic specifications can be roughly divided into two forms. Proofs in flat specifications, originating from the verification of data types, concentrate on deducing in basic specifications or "flattened" structured specifications. Thus the starting point here is simply a signature and a set of axioms, mostly first-order equations or Horn formulae. This allows to incorporate theorem provers, or term rewriting systems in the case of equations, easily. More involved approaches split up the function symbols of a signature into constructors, that is functions generating the objects, and defined functions, that is functions completely expressed in terms of constructors.

Structured proof methods [AKK99, Cen94], in contrast, take into account the structuring operators of the language. Based on a proof system for basic specifications, proof rules are given that mirror the individual specification-building operations, in other words the proof system is defined by induction on the structure of the specification. Thus formulae proven in a specification can be lifted to other specifications the construction of which uses this specification. Often a compromise between both proof methods is used where a

structured specification is transformed into some kind of intermediate normal form [AKK99].

Coming back to generic programming now, that is to the specification and verification of generic algorithms and their instantiations, three points are of major concern: First, the individual components, that is the basic specifications, should correspond to both generic algorithms and possible instantiations, that have to be described in order to fix their semantic requirements and properties respectively. Then these specifications have to be compared in the sense that they include each other, that is whether an algebra or a model of an instantiation specification is also a model of the specification of a generic algorithm. This is the major task in generic programming, to decide whether an instantiation is legal and should be reflected in the specification language. In the languages presented above the inclusion of specification is in some sense implicit depending on the structure mechanisms used. However given a particular algebra or specification checking whether it is included in some other specification is not well-supported. Second, one should bear in mind that specifying the requirements of generic algorithms is not only concerned with correctness of algorithms and instantiations, but also with requirements on complexity or performance of the algorithms and their instantiations. Thus a language for the specification of generic algorithms should also provide support to express such criteria formally, so that these can become part of an instantiation check. And third, as already indicated, a library of specifications for generic algorithms together with instantiations is necessary. Thereby, the library should be designed in a way so that the structure of the library supports directly the above mentioned questions of algebra inclusion.

A precise definition of the semantics is in some sense self-evident if we are looking for formal support. The same holds for proving properties of specifications which here corresponds to proving that generic algorithms are correct and that instantiations are legal for them. However, we do not consider this as an intrinsic component of the specification language. The reason is that in generic programming proving should rather be done off-line because complete proofs cannot be established at compile time. Thus the major concern of specification here is to enable for instantiation checking with respect to a given set of facts; in other words with respect to the knowledge given by a library of specifications. Nevertheless, the wish to prove that the check done at compile time is really true of course influences the way of specification: it must be possible to construct formal proof obligations concerning both the correctness of instantiation and the underlying library of specifications, which then are delegated to some mechanized reasoning system.

4.3 Concepts: Tecton

In the field of generic programming a concept stands for a set of requirements [SLL02, Aus99] attached to a generic type parameter `T`. A realization of `T` fulfilling these requirements can then be safely instantiated for `T`. This coincides with the view of type parameters `T` as descriptions of classes of algebras $C(\texttt{T})$ the elements of which correspond to legal instantiations. However, requirements for generic algorithms go beyond the scope usually addressed by the specification of abstract data types. They include guarantees concerning complexity or performance [SL00a]. This allows not only to choose the best realization of `T` with respect to some criteria, but also to decline an instantiation if, for instance, the resulting algorithms does not perform well enough. However, in existing programming languages, if any, both specification and algorithmic requirements are usually given informally. That is they are part of programming language descriptions or standards, as for example in the case of the Standard Template Library [MDS01]. Consequently, algorithmic as well as specification requirements are not part of the entire programming language and therefore cannot be handled explicitly during compilation.

Tecton [Mus98, MS02c] is a language that allows to describe concepts, that is collections of objects that satisfy a common set of requirements. Though also based on algebraic specification—a concept is in fact a class of many-sorted algebras—Tecton was designed to support formal software development and, in particular, specification and verification of generic system components [KM92]. Thus Tecton differs from common algebraic specification languages in some points. First, the signature given by a concept description describes the sort and function that an algebra belonging to the concept must at least provide, that is additional sorts and function may occur. Second, Tecton allows for parameterization without explicitly stating what the parameters are. Nearly all constituents of a concept may be replaced later, this includes also subconcepts. Third, the language includes the definition of legality conditions that must be met when using each construct of the language. Also Tecton allows to state lemmas not only concerning particular properties of a concept, but also concerning relations between concept instances. The Tecton semantics assign to each concept `C` a set of many-sorted algebras sem(`C`). Thereby, the syntax and formal semantics of Tecton are such that a corresponding proof theory requires only first-order and inductive proof methods.

In the following we briefly outline the major constructs of Tecton and provide some examples (see [MS02c] for details and more examples), before we give a discussion on how Tecton's concept descriptions can be used to

formally state semantic requirements of type parameters `T` so that legal instantiation can be checked.

The basic objects of Tecton are simple concept descriptions, that is concepts without inheritance from other concepts. Here, sort and function symbols are introduced using the keyword `introduces` and properties of the concept are defined in a first-order language following the keyword `requires`. The concept of a reflexive relation, for instance, can be described the following way. Actually, the `Reflexive` concept would inherit from a more general concept `Relation` (see [LMSS99]), but for illustration purposes we here give a stand-alone description.

```
Definition: Reflexive
  introduces domain,
       R: domain, domain -> bool;
  requires (for x: domain) x R x.
```

A `generates` clause restricts the algebras belonging to the concept to those in which the elements of a sort are finitely generated by the functions mentioned. This allows to use induction schemes for proving properties of the corresponding sort. For example, the following description defines the concept of natural numbers giving the usual induction scheme.

```
Definition: Natural
  introduces naturals,
  ...
  generates naturals freely using 0, succ;
  ...
```

Note that `generates` clauses can occur without the word `freely`. In this case the terms generated need not be distinct. In our example, thus algebras may identify, for instance, `0` and `succ(succ(0))`, which means that the concept `Natural` would include the residue class rings $\mathbb{Z}_n$ also.

Subsort declarations are possible in Tecton with their obvious meaning; they are given in the `introduces` clause The non-zero natural numbers, for instance, can be introduced in the concept `Natural` as follows.

```
nonzero < zero
```

However, if a subsort is introduced, Tecton requires the definition of a predicate in the `requires` clause describing in which cases the downcast of an element of the supersort to the subsort is possible. This is written with a

special syntax using the `in` operator. For the example concept `Natural` the predicate would look like this:

```
requires (for n: naturals)
     n in nonzero = not(n = 0),
...
```

Now, the semantics sem(`A`) of a simple concept description `A` is just the set of many-sorted algebras providing at least the sorts and functions given by `A`, which means in Tecton notation that `A` belongs to the syntactic concept syn(`A`), and fulfilling all the requirements, subsort relations and `generates` clauses in the usual sense.

To reuse already described concepts Tecton provides `refines` and `uses` clauses. In addition they also allow to identify parts of a definition that can be treated as parameters and can thus be replaced in later concept descriptions. The concept `Quasi-order` for example can be easily derived from the concepts `Reflexive` and `Transitive`.

```
Definition: Quasi-order
 refines Reflexive, Transitive.
```

The meaning of this concept description is that the requirements of the bequested concepts are combined in the new concept, that is roughly speaking we have sem(`C`) = sem(`A`) ∩ sem(`B`) if `C` imports `A` and `B`. Note that in the example this implies that each algebra belonging to the concept `Quasi-order` also belongs to both concepts `Reflexive` and `Transitive`.

Concept descriptions using inheritance, of course, can also come with the postulation of further properties using the `requires` clause. This can be used to build special instances of general concepts, for example to introduce the equivalence relation in commutative rings, saying that two elements are associated to each other, if they only differ by a unit. To do so the concept `Equivalence-relation` is combined with the concepts `Commutative-ring-with-identity` and `Unit` to provide the necessary vocabulary. Then the general equivalence relation `equiv` inherited is specialized in the `requires` clause to give it the particular meaning used in commutative rings.

```
Definition: Unit-equivalence
 refines Equivalence-relation;
 uses Commutative-ring-with-identity, Unit;
 requires (for x, y: domain)
    (x equiv y) = ((for some z: units) x = z * y).
```

Note the difference between a `uses` and a `refines` clause [MS02b]. A `use` clause indicates that the imported concept is a component of the resulting concept, that is the imported concept is preserved in the sense that its semantics are not changed. A `refines` clause in contrast, says that the imported concept is modified. This means that after importing the concept some algebras of the concept will be ruled out. To be more precise, not changing semantics means that each algebra belonging to the original concept `A` must also appear in the new concept `B` importing `A` and vice versa. However, because the new concept `B` may introduce new sorts and function symbols Tecton requires this for the restriction to the original signature. So let $\mathcal{A}|_{\mathtt{A}}$ be the subalgebra of $\mathcal{A}$ whose sorts and function symbols are restricted exactly to those available in `A`, and set

$$\mathtt{B}|_{\mathtt{A}} \;=\; \{\, \mathcal{A}|_{\mathtt{A}} \mid \mathcal{A} \in \mathrm{sem}(\mathtt{B}) \,\}$$

for concepts `A` and `B` with $\mathrm{sem}(\mathtt{B}) \subseteq \mathrm{sem}(\mathtt{A})$. Note that $\mathtt{B}|_{\mathtt{A}} \subseteq \mathtt{A}|_{\mathtt{A}} \subseteq \mathrm{sem}(\mathtt{B})$ holds. Then if a concept `B` imports a concept `A` via a `use` clause the condition that `B` preserves `A` is formally given by

$$\mathrm{sem}(\mathtt{B}) \subset \mathrm{sem}(\mathtt{A}) \;\wedge\; \mathtt{B}|_{\mathtt{A}} \;=\; \mathtt{A}|_{\mathtt{A}}.$$

This allows to treat the inherited concept description as a parameter for which any algebra belonging to it can be safely substituted and can be compared with a conservative extension. This is not the case if a `refines` clause is used, here inconsistencies between the properties of the inherited and bequested concepts may occur. Thus the legality condition in this case is

$$\mathtt{B}|_{\mathtt{A}} \subset \mathtt{A}|_{\mathtt{A}} \;\wedge\; \mathtt{B} \text{ preserves any concept that } \mathtt{A} \text{ preserves.}$$

That is, though `B` may change `A`, this is not allowed arbitrarily: Subconcepts preserved by `A` may not be changed. This guarantees that the modification of `A` is indeed a refinement and does not change `A`'s original intention. To conclude with, in the example concept above the legality condition belonging to the import of concept `Commutative-ring-with-identity` reads

$$\mathtt{Unit\text{-}equivalence}|_{\mathtt{Commutative\text{-}ring\text{-}with\text{-}identity}} = \mathtt{Commutative\text{-}ring\text{-}with\text{-}identity}|_{\mathtt{Commutative\text{-}ring\text{-}with\text{-}identity}}$$

which means that all commutative rings with identity indeed appear in the semantics of `Unit-equivalence`.

Instantiation is the most powerful Tecton construct. As already mentioned, concept descriptions do not have explicit parameters; instead many

constituents of a concept description can be considered as parameters and thus be replaced, for instance subconcepts given by `uses` or `refines` clauses. Instantiations in a concept description are indicated using a `with` clause which appears directly after the concept name. For example, the following renames the function symbol `equiv` to the more usual notation $\sim$.

```
Unit-equivalence [with ∼ as equiv]
```

More important is concept instantiation where a whole part of a concept description, a subconcept, is replaced by another one. We illustrate this by giving the concept description of ample sets in commutative rings. A set of representatives with respect to an equivalence relation is a set that contains exactly one element out of each equivalence class. This has been introduced in Tecton in the concept `Set-of-representatives` for arbitrary sets and arbitrary equivalence relations (see [LMSS99]). Now, an ample set is a special set of representatives where the set is a commutative ring and the equivalence relation is the unit equivalence presented above. That is the concept of ample sets emerges from the concept of set of representatives by restricting both the possible sets and equivalence relations. Instead of repeating all the requirements for commutative rings and unit equivalence, Tecton allows to build a new concept based on `Set-of-representatives` by replacing the subconcept `Equivalence-relation` with the concept `Unit-equivalence`. Thus the concept `Ample-set` is given by

```
Definition: Ample-set is Set-of-representatives
      [with Unit-equivalence as Equivalence-relation].
```

Note that in the concept description of `Set-of-representatives` it was not explicitly stated that `Equivalence-relation` is a parameter that can be replaced; `Equivalence-relation` was just inherited by a `uses` clause. This gives the language user greater flexibility, because it is sometimes difficult to foresee which constituents of a concept may be useful to substitute later.

However, replacement of concepts is not completely arbitrary. As already mentioned there are again legality conditions avoiding incorrectness due to uncontrolled instantiation. Consider for example the concept replacement `D = C[with B as A]`; parallel replacements are allowed, but for illustration it suffices to treat one only; for further details see [MS02c]. First of all, concept `A` must be available in concept `C`, so that the instantiation can take place. Now the intention is that if `B` preserves or modifies `A`, then after instantiating `B` for `A` in `C`, the resulting concept `D` should preserve or modify `B`. So, if $\mathtt{B}|_{\mathtt{A}} = \mathtt{A}|_{\mathtt{A}}$, that is if `A` behaves like a component in `B`, it is required that $\mathtt{C} \subset \mathtt{B}$ and $\mathtt{C}|_{\mathtt{B}} = \mathtt{B}|_{\mathtt{B}}$ holds to ensure that `B` acts like a component of `D`. Similarly,

if we have $\mathtt{B}|_{\mathtt{A}} \subset \mathtt{A}|_{\mathtt{A}}$, we require that $\mathtt{C}|_{\mathtt{B}} \subset \mathtt{B}|_{\mathtt{B}}$ and that `C` preserves every concept that `B` preserves, thus that `B` looks like an ingredient of `D`.

As a consequence each concept inherited by the base concept `C` through a `use` or `refines` clause is actually a formal parameter and can be replaced provided the legality conditions are fulfilled.

Tecton has been incorporated in the programming language SuchThat [Sch96] to formally specify generic algorithms. Tecton concepts are used to describe requirements under which a generic algorithm works correctly. Therefore, the algorithm header does not only include an input/output specification, but also explicitly names some Tecton concept by a `use` clause. These concepts not only provide the sorts and identifiers available in the algorithm, but also specify semantic requirements for legal instantiations of generic parameters `T`, that is $C(\mathtt{T})$ is identified with the semantics of the concepts imported. Thus if an algorithm `A` is specified with a concept `C` then each instantiation `P` with $\mathcal{A}_P \in \mathrm{sem}(\mathtt{C})$ is a legal instance for `A`.

Consider as an example the well-known Euclidean algorithm for computing greatest common divisors. The method works for all Euclidean domains given by the concept `EuclideanRing`. However, because greatest common divisors are only unique up to units, ample sets have to be incorporated to pick a result in a deterministic manner. This corresponds to taking the absolute value in case of integers. Using these two concepts the algorithm can be stated as follows.

```
Algorithm: w := Euclid(u,v)
 uses EuclideanRing,
      Ampleset [with A as set-of-representatives].
 Input:  u,v ∈ domain.
 Output: w ∈ A such that w = gcd(u,v).
while v ≠ 0 do {z := u mod v; u := v; v := z};
w := representative(u).
```

Note that the types used in the input/output specification correspond to sorts introduced in the concepts imported. The same holds for the function descriptor `gcd` describing the function computed by the algorithm. In the algorithm body two function descriptors `mod` and `representative` appear. They must also have a counterpart in the Tecton concepts. However, Tecton concepts only specify properties of the function descriptors for which algorithms must be provided. To find a (generic) algorithm realizing for example `representative` algorithm headers have to be checked whether the output specification states that this function is computed. To avoid this search,

SuchThat therefore allows to replace function descriptors by algorithm descriptors, which are not included in the original Tecton language. A function descriptor corresponds to the name of a (generic) algorithm which then is used as the realization of the function descriptor being substituted. In the example one would thus replace `representative` by an algorithm descriptor `REP` standing for the corresponding algorithm.

```
Algorithm: w := Euclid(u,v)
 uses EuclideanRing,
      Ampleset [with A as set-of-representatives
                      REP as representative].
 Input:  u,v ∈ domain.
 Output: w ∈ A such that w = gcd(u,v).
while v ≠ 0 do {z := u mod v; u := v; v := z};
w := REP(u).
```

As the example shows Tecton is a natural way to specify the application range of generic algorithms in a natural though exact way. In comparison with other algebraic specification languages Tecton possesses a number of advantages for doing that. Signatures of algebras are not restricted to the one of the concept description, they may contain further sorts or function symbols. This is important for generic programming because, for instance, group algorithms should be applicable to rings. As already mentioned concept instantiation and replacement can be done without referring to explicitly stated parameters. Nevertheless correctness of instantiation can be ensured by verifying the corresponding proof obligations. This gives the user greater flexibility when developing concepts and algorithms.

Furthermore, the way Tecton concepts are built naturally supports checking for correct instances. If, for instance, the example algorithm is to be instantiated with the integers, it has to be checked whether the integers, given by the concept `Integer`, form a Euclidean domain, that is belong to the concept `EuclideanRing`. Now, if the concept `Integer` imports `EuclideanRing` by a `use` or `refines` clause this is obvious. Following the inheritance of concepts a hierarchy of concept inclusions can be easily built. But, what is more important, Tecton allows to formulate lemmas about concept inclusion and equality. Thus the following states that every algebra belonging to the concept `Integer` in particular belongs to the concept `EuclideanRing`.

```
Lemma: Integer implies EuclideanRing.
```

Of course Tecton lemmas come with corresponding proof obligations, namely to verify the inclusion. However, as stated explicitly these concept inclu-

sions can be used for checking legal instantiation. Thus the Tecton language provides constructs not only to build a hierarchy of structures, but also to present it in a way supporting both the specification of generic algorithms and the check for legal instantiation of generic algorithms. In particular, Tecton allows to include special domains like the integers into a hierarchy of abstract domains, which is crucial for the instantiation of generic algorithms.

4.4 A Properties-based Approach

In the last section it has been shown how the Tecton language allows to specify generic type parameters `T` by identifying the class of legal instantiations $C(\mathtt{T})$ with the semantics of a concept description $\mathtt{C_T}$ of `T`. This description provides both the set of operations and their semantic properties necessary for a legal instantiation. Thus every instance of the concept, that is every algebra $\mathcal{A}$ with $\mathcal{A} \in \mathrm{sem}(C_{\mathtt{T}})$ is a legal instantiation of `T`. However, though a concept comes as a module, its definition may be distributed in a number of imported concepts or extensions. This makes it sometimes difficult to see which properties are actually required by a generic algorithm, which is in particular necessary if a non-legal instantiation occurs. It may thus be better to give the possibility to describe individual properties of operations in a stand-alone manner. The combination of properties attached to a generic algorithm as the description of a generic type parameter `T` then gives the individual properties that as a whole are necessary for legal instantiations. Though Tecton supports the combination of properties by providing various constructs to extend, combine and refine concept descriptions, the description of the requirements necessary for a generic algorithm and the description of the algorithm itself remain separated.

In the following section we present an approach for the description of generic type parameters `T` focussing on the individual properties of the operators involved, that is on the operators and their properties necessary for an instantiation to be legal. The idea is to define properties of operators independently of domains or concept descriptions [Sch01b]. Properties can then be arbitrarily combined for both the description of requirements for generic algorithms and the description of properties a possible instantiation comes with. These are directly attached to the algorithms allowing to specify requirements of generic algorithms at an arbitrary level of detail. In particular the required operations themselves to be provided by an instantiation are explicitly given in the algorithm's description. This simplifies the definition

of legal instantiation and thus allows for a straightforward check.

Consider for example the property of being associative for an operation + over a domain R. This property is introduced as a predicate named Associative. The arguments correspond to the sort and function symbols necessary to state the property, here a domain R and the operation +.

```
Property: Associative(R,+)
 with +: R x R -> R
 means for all a,b,c ∈ R: a + (b + c) = (a + b) + c.
```

Thereby the (predicate) name of the property can be used to refer to it. Note again that the sorts and operators necessary for the semantic requirement are explicitly given as the arguments of the predicate. As already mentioned combining such properties gives the description of a generic type parameter T. Consequently, the description of a generic type parameter T is given by a signature, and a set of properties defined over this signature, that is

$$\mathtt{T} \;=\; (Sig(\mathtt{T}),\; Props(\mathtt{T}))$$

where all sort and function symbols used in $Props(\mathtt{T})$ must also appear in $Sig(\mathtt{T})$. Note that the signature $Sig(\mathtt{T})$ may include more sort and function symbols than necessary for the properties following. This allows to distinguish between applicability and correctness of an instantiation as mentioned in chapter 2. The signature gives the syntactic requirements, that is which operations an instantiation has to provide; the set of properties describes the semantic requirements leading to a well-behaved instance of the generic algorithm. In addition this enables to specify possible instantiations, that is individual algebras, similarly by a signature giving the operations the instantiation comes with and a set of properties of these operations. Using this representation the class of structures a description of a generic type parameter defines simply is

$$C'(\mathtt{T}) \;:=\; \{\; \mathcal{A} \mid Sig(\mathtt{T}) \subseteq Sig(\mathcal{A}) \;\wedge\; Props(\mathtt{T}) \subseteq Props(\mathcal{A}) \;\}.$$

Note that in contrast to $C(\mathtt{T})$ where all properties of the algebras $\mathcal{A}$ following from a set of axioms were taken into account, here a somewhat different point of view is chosen: Not the properties an algebra $\mathcal{A}$ implicitly obeys are considered, but only the properties explicitly stated about $\mathcal{A}$. It follows that an algebra $\mathcal{A}$ may not belong to $C'(\mathtt{T})$ just because properties $\mathcal{A}$ comes with are not explicitly stated. Thus it may happen that algebras belonging to $C(\mathtt{T})$ are ruled out. The properties attached to an algebra $\mathcal{A}$ may be interpreted as

the amount of knowledge available about $\mathcal{A}$ at a given point of time that may evolve. Thus $C'(\mathtt{T})$ can be seen as an approximation of $C(\mathtt{T})$ with respect to properties explicitly known and stated.

On the other hand, however, the definition of the legality predicate with respect to $C'(\mathtt{T})$ reduces to a simplified version of the one given in section 4.1. Again only properties explicitly given in the description of type parameters and algebras are considered. Thus, an instantiation $\mathcal{A} = (\mathrm{Sig}(\mathcal{A}), \mathrm{Props}(\mathcal{A}))$ is legal, that is an element of $C'(\mathtt{T})$, if it provides the necessary operations given by the signature of the parameter T and all the properties required by T are included in the properties of $\mathcal{A}$. Thus we get

$$Leg(\mathcal{A}, \mathtt{T}) \ :\Longleftrightarrow \ Sig(\mathtt{T}) \subseteq \mathrm{S}ig(\mathcal{A}) \ \wedge \ Props(\mathtt{T}) \subseteq Props(\mathcal{A}).$$

The subtyping problem can be simplified also. A type T' is a subtype of another type T, if $C'(\mathtt{T}') \subseteq C'(\mathtt{T})$ holds, that is if T' explicitly requires more operations symbols or more properties than T. This can now, similarly to the legality predicate, easily be checked by comparing both the signatures and the set of properties of T and T' with respect to inclusion, that is we have the following refinement of the definition given in section 4.1.

$$\mathtt{T}' \ \leq' \ \mathtt{T} \ :\Longleftrightarrow \ Sig(\mathtt{T}) \subseteq Sig(\mathtt{T}') \ \wedge \ Props(\mathtt{T}) \subseteq Props(\mathtt{T}')$$

Again $\leq'$ can be seen as an approximation of $\leq$ where only properties known and explicitly stated about the types T and T' are taken into account. However, the need to compare sets of algebras has been reduced to the need of comparing sets of properties being represented by predicates. This can be seen as a compromise between considering the complete set of algebras the description of a generic type parameter T allows for and the effort necessary to check whether an algebra belongs to this set. Nevertheless formal definitions of properties are provided so that there is the possibility to further properties of generic algorithms and possible instantiations.

Properties-based descriptions of generic type parameters can be used to specify generic algorithms in a way similar to the one used in SuchThat. The description, that is a signature and a set of property predicates over this signature, is attached to the algorithm header and in this way becomes part of the programming language, which can be processed in order to check semantic requirements of generic algorithms. Possible instantiations also come with such a description and thus allow for this check by simply comparing sets of properties with respect to inclusion. For example, the Euclidean property can be formalized as follows.

```
Property: Euclidean(R,+,*,0)
 with +,*: R x R -> R,
      0,1: -> R
 means ex φ: R\{0} -> ℕ:
       for all a,b ∈ R st b ≠ 0:
       ex q,r ∈ R:
             a = q * b + r & φ(r) < φ(b) or r = 0.
```

Note that the definition of this property allows to be used not only as usual for integral domains, but can be applied to every domain obeying the necessary operations. Defining the Euclidean property on top of integral domains implies that every legal instantiation for an algorithm requiring this property must in particular be an integral domain independent of whether this is necessary for the algorithm to work properly. Using a properties-based approach requirements can be easily split up in what is indeed necessary and in what is not. For the Euclidean algorithm we get in particular that multiplication of the underlying domain need not be commutative.

```
Algorithm: w := Euclid(u,v)
 [(R,+,*,0,1,S,rep);
  Group(R,+,0), Associative(R,*), Distributive(R,+,*),
  Euclidean(R,+,*,0),
  AmpleSet(S,R), AmpleFunction(rep,S).]
 Input:  u,v ∈ R
 Output: w ∈ S such that w = gcd(u,v).

while v ≠ 0 do
   {z := u mod v; u := v; v := z};
w := rep(u).
```

Note the use of property predicates at different levels: The three usual properties describing the axioms of a group have been put together in one property. This is nothing else than an abbreviation, of course they could have been also stated themselves. This gives the user the freedom to work on the level of abstraction he considers appropriate. In the example, because not all properties the multiplication of a ring comes with are necessary for the Euclidean algorithm the necessary ones has been attached separately, namely that multiplication is associative and distributes with addition. Nevertheless a user interested in ring only can apply the algorithm. Thus focussing on individual properties of operations supports the attempt to present minimal requirements of generic algorithms without bothering a possible user with every detail.

Legal instantiations can now be found by checking whether properties required by a generic algorithm are attached to them. For example, a description of the integers with an appropriate ample set would include the necessary properties for the above algorithm and thus constitute a legal instantiation for the Euclidean algorithm. The problem of finding appropriate subalgorithms can be also addressed. To find a generic algorithm that can be applied the description attached to it is used. If the generic algorithm does not require more properties than the algorithm to be used as a subalgorithm, then the subalgorithm can be applied. In other words the signature and the properties attached to the subalgorithm must be contained in the generic algorithm's signature and properties, that is the type `T` of the generic algorithm must be a subtype of `T'`, the subalgorithms type. For example, the computation of the `mod` operation does not require an ample set, but only some properties concerning rings, among them of course the Euclidean property to ensure termination. Thus a (non-efficient) generic algorithm for the `mod` operation that can be used in the generic Euclidean algorithm can be specified as follows.

```
Algorithm: w := mod(u,v)
  [(R,+,*,0,1);
    Group(R,+,0), Associative(R,*), Distributive(R,+,*),
    Euclidean(R,+,*,0)]
  Input:  u,v ∈ R such that v ≠ 0
  Output: w ∈ R such that w = u mod v.
w := 0;
while u ≠ 0 do
   {u := u - v; w := w + 1};
w := u.
```

As we see, both the signature and the properties attached to the algorithm are contained in the ones of the generic Euclidean algorithm, thus it is a legal subalgorithm. Note that this can be easily checked due to the representation of properties predicates. Special non-generic algorithms can be easily incorporated, too. Here one has to attach a description of the properties the particular domain comes with, for example for the ring of integers. Note that we have abbreviated the individual properties of the integers with a predicate `EuclideanRing`. Then particular algorithms using special properties or representations of the integers can be described. For example the `mod` operation can be realized by the involved division algorithm following [Knu98], which we have here simply abbreviated by `MOD`.

```
Algorithm: w := mod(u,v)
  [(ℤ, +ℤ, *ℤ, 0ℤ, 1ℤ);
   EuclideanRing(ℤ, +ℤ, *ℤ, 0ℤ, 1ℤ)]
  Input:  u,v ∈ ℤ such that v ≠ 0
  Output: w ∈ ℤ such that w = u mod v.
w := u MOD v.
```

Now, the predicates stated for this algorithms include the ones necessary for the generic `mod` algorithm, thus the algorithm (and every generic algorithm using the `mod` algorithm that is to be instantiated with the integers) can apply this particular integer version.

The same holds for the realization of the ample set and the `rep` subalgorithm in the generic Euclidean algorithm. The non negative integers, for instance, constitute an ample set for the integers and computing a representative means nothing else than taking the absolute value. This can be easily specified as follows.

```
Algorithm: w := rep(u)
  [(ℤ, +ℤ, *ℤ, 0ℤ, 1ℤ);
   EuclideanRing(ℤ, +ℤ, *ℤ, 0ℤ, 1ℤ),
   AmpleSet(ℤ≥0, ℤ), AmpleFunction(rep,ℤ≥0)]
  Input:  u ∈ ℤ
  Output: w ∈ ℤ≥0 such that w = rep(u).
w := |u|.
```

Note that the set of properties here has been extended to describe the chosen ample set for the integers. Of course these properties hold for the integers, and thus are independent of the particular algorithm they are attached to. This suggests to keep properties of special domains as a module separated from the entire algorithms from which they can be accessed if an algorithm with this domain is taken into consideration.

4.5 Algorithmic Requirements

The most important characteristic of an algorithm is its correctness in the sense that the algorithm's formal specification completely and correctly describes what the algorithm computes. This of course holds for both generic and non-generic algorithms, although as we have seen correctness of generic algorithms comes in two facets, correctness of the generic algorithm itself and correctness of its particular instances. However, apart from this more

theoretical issue there are other relevant points, especially if the algorithm is to be stored in a library and applied by users. The main reason is that if an algorithm is stored in a library most users take the algorithm's correctness for granted and are more interested in topics like the algorithm's running time or memory management.

In particular, the algorithm's efficiency is of major interest. However, for generic algorithms this does not only depend on the algorithm itself, but also on the efficiency of the subalgorithms being instantiated. There are generic algorithms where the underlying method is, in principle, efficient, but the efficiency of the final non-generic algorithm depends on the instantiation as well. So for example, computing the greatest common divisor based on Euclid's method is accepted to be sufficiently efficient for the integers and other special domains. However the efficiency of a generic version strongly depends on the efficiency of the instantiated algorithm for computing remainders. That is, the behaviour of an instantiated generic algorithm also depends on properties of the code being instantiated. Thus it seems reasonable to provide the specification of generic algorithms with additional information on such *algorithmic requirements*.

This can be information about the expected complexity of the instantiated code as done in the (informal) specification for the Standard Template Library [MDS01, Aus99]. In the area of computer algebra we find other examples for algorithmic requirements. Algorithms for computing with polynomials are practically efficient only for a bounded degree. Another example are matrices where the matrices' dimension plays the role of the complexity parameter. And, as already mentioned in section 2.4, even the correct behaviour of the instantiated generic algorithm can depend on the instantiation's parameters, for example on the number n of elements when computing inverses in finite fields. Specifying this explicitly does not only support users but also enables automated checking whether instances will meet the expected algorithmic behaviour.

It is straightforward to include algorithmic requirements into the properties-based approach because properties are described symbolically. For example, the fact that the algorithm `rep` computing associated elements of an ample set should be of linear complexity can be easily expressed as follows.

`LinearComplexity(rep)`

Then this predicate can be included in both the generic algorithm's specification with the intention that only linear algorithms should be instantiated for `rep` and in the instantiation's specification expressing that the instantiation

works in linear time. By comparing the sets of properties it can then be ensured that only instances with a linear subalgorithm are legal.

The Tecton concept description language also allows for expressing algorithmic requirements. Here, the necessary concepts have to be introduced expressing the desired properties. After that importing these concepts into the algorithm's header specification restricts possible realizations to those with the desired additional properties.

In the following we give an example for specifying algorithmic requirements in Tecton. Thereby we focus on computational complexity and extend the Tecton Concept Library [LMSS99] so that various classes of complexity can be described easily. Including these Tecton classes in the algorithms' headers as already seen in section 4.3 then restricts the possible instantiated algorithms to the ones of this class in this way expressing algorithmic requirements for complexity.

To describe the complexity of functions usually sets of positive real-valued functions, that is functions returning non negative real numbers, is used. In the Tecton concept library a set of functions is given in the concept `Map` together with a function `apply` assigning each pair consisting of a map `m` and an element `d` of the domain an element of the range, thus the value of `m` under `d`. The concept `Function-set` where different sets of functions are denotable, can therefore be constructed by merging the concepts `Map` and `Set` and identifying the `domain` over which sets are built with `maps` as done in the following definition.

```
Definition: Function-set
 refines Map,
         Set [with maps as domain, set-of-maps as sets];
```

Now sets of positive real-valued functions are simply given by the concept `Function-set` where the `range` is substituted with the positive real numbers. In text books the real numbers are chosen because computing time functions of algorithms are usually counting elementary operations, thus map into the natural numbers. To define the complexity of functions, however, not the real numbers are necessary, but only properties of an ordered ring in order to compare function values. The real numbers thus represent a particular choice of complexity functions which can be substituted if necessary. This is exploited in the definition of the following concept `Complexity` which therefore does not import the concept `Real` but only the concept `Ordered-ring` together with a sort `positive-domain` representing the elements greater than

0. Note that the real numbers Real are a realization of this concept, thus our approach is a slight generalization of the one found in text books.

The definition of the complexity of functions follows [Col74] where a binary relation $\leq$, called "dominated by", on positive real-valued functions is introduced to compare functions with respect to their returned values. Roughly speaking a function f is dominated by a function g, if all values of f are not greater than the ones of g multiplied with a constant c. This is expressed using the function apply inherited from the concept Function-set.

```
Definition: Complexity
 refines Ordered-ring
             [with range as domain,
                   positive-range as positive-domain],
         Function-set;
 introduces ≤: maps x maps -> bool;
 requires (for f,g: maps)
    (f ≤ g) =
     ((for some c: positive-range)
      (for a: domain) apply(f,a) ≤ c * apply(g,a)).
```

The concept Complexity can be extended by introducing some more or less obvious notions derived from the definition of $\leq$ such as $\sim$, $<$ or $\geq$. The most important one here is the notion of a complexity class, that is the set of all positive real-valued functions dominated by a given function. Thus complexity-class is a function from maps into sets of maps. This will be used in the following to define more special complexity classes that can be attached to algorithms.

```
Extension: Complexity
 introduces
  complexity-class: complexity-functions -> set-of-maps,
  ≥: complexity-functions x complexity-functions -> bool,
  ∼: complexity-functions x complexity-functions -> bool,
  <: complexity-functions x complexity-functions -> bool,
  >: complexity-functions x complexity-functions -> bool;
 requires (for f,g: complexity-functions)
  (f ≥ g) = (g ≤ f),
  (f ∼ g) = (f ≤ g and g ≤ f),
  (f < g) = (f ≤ g and not(g ≤ f)),
  (f > g) = (g < f),
  (f in complexity-class(g)) = (f ≤ g).
```

The above concept describes all functions and all complexity classes with respect to $\leq$. To express the complexity of algorithms we in addition need a concept defining a particular complexity class in which the algorithm's computing time function shall lie. Therefore in the following refinement of concept `Complexity` a particular complexity function `g` is introduced and the requirements says that all functions belonging to the concept must be dominated by `g`.

```
Definition: Complexity-class
refines Complexity;
 introduces g: -> complexity-functions;
 requires (for f: complexity-functions)
   f in complexity-class(g).
```

Particular complexity classes can now be easily defined by formulating further requirements on the function `g`. For example attaching the requirement $\mathtt{g(a)} \leq \mathtt{c} * \mathtt{a}$ for all `a` and some positive constant `c` restricts the `maps` of the concept `Complexity` to linear ones, that is all linear functions are part of the new concept. Note however, that for this requirement the domain and the range of `g` must be the same in order to compare `g(a)` and $\mathtt{c} * \mathtt{a}$. This is why the `with` clause in the following definition is necessary, it identifies the `domain` and the `range`.

```
Definition: Linear-complexity
 refines Complexity-class
             [with domain as range,
                   positive-domain as positive-range];
 requires (for some c: positive-domain)
    (for a: domain) apply(g,a) ≤ c * a.
```

In addition more special complexity classes can be built, also based on complexity functions over particular domains like the real numbers. For example, if `g` is intended to be the function that maps every real number r to r^2+r+3, this can be done by substituting the concept `Real` for `Ordered-ring`. Note that the concept `Real` need not be imported, the concept replacement is sufficient.

```
Definition: Special-complexity-class
 refines Complexity-class [with Real as Ordered-ring],
 requires (for r: reals)
    apply(g,r) = 5 * r * r + r + 3.
```

So far we have formalized the basics notions necessary for specifying the complexity of algorithms. The key point is that these descriptions can be combined with the specification of generic algorithms similar to other requirements as done in section 4.3. To do so, we first define an abstract concept for algorithms. Here, a sort `algorithm` is introduced on which functions describing properties of algorithms are working. The functions `valid-input` and `valid-ouput` map an algorithm `a` to the sets of element serving as input and output of the algorithm, respectively. Note, that `valid-input(a)` is a subset of `domain` and `valid-output(a)` a subset of `range`.

The function `sem` maps algorithms `a` onto elements of `algorithm-maps`, that is onto functions describing the input/output behaviour of `a`. Similarly `time` associates with each algorithm `a` an element of `complexity-functions`, that is the computing time function of `a`. Other characteristics of algorithms such as a memory function `space` can be introduced the same way. We also included requirements describing the relationship of `valid-input(a)` and `domain` as well as `valid-ouput` and `range`.

```
Definition: Algorithm
 uses Set [with sets-of-domain as sets],
      Set [with range as domain, sets-of-range as sets],
      Map [with algorithm-maps as maps],
      Map [with complexity-functions as maps,
                complexity-range as range];
 introduces algorithms,
    valid-input: algorithms -> sets-of-domain,
    valid-output: algorithms -> sets-of-range,
    sem: algorithms -> algorithm-maps,
    time: algorithms -> complexity-functions,
    space: algorithms -> complexity-functions.
 requires (for a: algorithms; i: domain; o: range)
    (i in valid-input(a)) =
          (apply(sem(a),i) in valid-output(a)),
    (o in valid-output(a)) =
          ((for some i: domain)
           i in valid-input(a) and apply(sem(a),i) = o).
```

The semantics, that is the function `sem`, of particular algorithms can be described in more detail by refining `domain` and `range`. For example, to describe sorting algorithms both `domain` and `range` are identified with `sequences` imported by the concept `Finite-sequence-with-order`, a refinement of the concept `Finite-sequence` in which the range of `sequences`

is equipped with a partial order. Then `valid-input` is defined to be the set of all `sequences` for an arbitrary sorting algorithm `a`. Finally, the semantics, that is the input/output behaviour, of sorting algorithms are defined as usual applying the functions `permutation` and `ordered` given by the concepts `Finite-sequence` and `Finite-sequence-with-order` respectively.

```
Definition: Sorting-algorithm
 refines Algorithm [with sequences as domain,
                          sequences as range];
 uses Finite-sequence-with-order;
 requires (for a: algorithms; s: sequences)
    s in valid-input(a),
    permutation(apply(sem(a),s),s),
    ordered(apply(sem(a),s).
```

The same can be done for the description of complexity. Note that if `a` is an algorithm, `time(a)` is a function mapping elements of `domain`, thus in particular of `valid-input(a)`, into `range`; that is `time(a)` gathers the amount of computing time for each individual input of the algorithm `a`. In concept extensions `time` can be refined to characterize more common versions of computing time functions. For example, using the concept of equivalence-classes the maximum time function of algorithms can be described as usual. The value of the worst-case complexity function `worst-time` for an equivalence class is just the maximum value `time` returns for the elements of this class.

```
Extension: Algorithm
 uses Equivalence-class,
      Partial-order [with complexity-range as domain],
      Map [with equivalence-classes as domain,
                worst-case-complexity-functions as maps,
                complexity-range as range];
 introduces
    worst-time: algorithms ->
                          worst-case-complexity-functions;
 requires (for a: algorithms; e: equivalence-classes;
               d: domain)
   (for some max: domain)
    (max in e) and
    ((d in e) and (d in valid-input(a))) implies
            (apply(time(a),d) ≤ apply(time(a),max)) and
    (apply(worst-time(a),equivalence-class(max)) =
                                    apply(time(a),max)).
```

Now, the combination of the concepts `Algorithm` and `Complexity` allows to state algorithmic requirements of particular algorithms: The functions such as `time` describing these properties are inherited from the concept `Algorithm`. These functions are required to belong to complexity classes given by the concept `Complexity`. This is realized by introducing complexity functions, here called `g` and `worst case`, generating different complexity classes. Note, that this is done for a newly introduced constant `alg` of sort `algorithms`. This allows to instantiate the concept with various algorithms in this way specifying their complexity.

```
Definition: Algorithm-complexity
 uses Algorithm, Complexity,
      Complexity [with worst-case-complexity-functions as
                                      complexity-functions];
 introduces
    alg: -> algorithms,
    g: -> complexity-functions,
    worst-case: -> worst-case-complexity-functions;
 requires
    time(alg) in complexity-class(g),
    worst-time(alg) in complexity-class(worst case).
```

The key point is that the functions `g` and `worst case` describing the complexity of the algorithm `alg` can be refined so that appropriate requirements for the actual algorithm are given. For example, the following concept defines linear functions. A special map `linear` is introduced whose values are bounded by a constant `c`. This map is then used as an instantiation in the case where linear functions are required.

```
Definition: Linear-function
 refines Map [with domain as range];
 uses Ordered-ring;
 introduces linear: -> maps;
 requires (for some c: positive-domain)
    (for a: domain) apply(lf,a) ≤ c * a.
```

Thus, to specify for example that in the Euclidean algorithm the subalgorithm computing associated elements of an ample set should work in linear time `Linear-function` is additionally imported. Then the function `linear` can be substituted for the function `worst case` defined in concept `Algorithm-complexity` in this way ensuring that the worst-case complexity function of algorithm `alg` identified with the desired subalgorithm `REP` is

dominated by a linear function. Hence we get the following algorithm header for algorithm `Euclid` extending the one from section 4.3.

```
Algorithm:  w := Euclid(u,v)
 uses EuclideanRing,
      Ampleset [with A as set-of-representatives,
                     REP as representative];
      Linear-function,
      Algorithm-complexity [with REP as alg,
                                  linear as worst case];
 Input:  u,v ∈ domain.
 Output: w ∈ A such that w = gcd(u,v).
```

This specification provides the user not only with information with which domain the generic algorithm should be used but also indicates properties of the algorithms to be instantiated in order to result in an efficient non-generic algorithm. Note that the information is attached directly to the algorithm and not given separately in some external description or standard. In addition, using this approach the user can define his own algorithmic requirements by extending the concepts given. He can introduce concepts for further characteristics of algorithms similar to the way complexity functions have been handled and combine these with the given concept for algorithms. Furthermore additional complexity functions can be added easily; they can be introduced in a concept like the one for linear functions in the example. Due to the instantiation mechanism of the Tecton language these concepts can then be incorporated in the algorithms' descriptions without problem.

Including the specification of algorithmic requirements provides the user with information beyond correctness and usability. However, the actual behaviour of an algorithm may still vary from what is expected. For example, the computing time of an algorithm, though in principle acceptable due to static analysis, may cause problems depending on the particular input. The reason is some characteristic not of the algorithm but of the machine the algorithm is run on. This may be, for example, memory management or the compiler used.

Thus there is a gap between the theoretical analysis of algorithms and their actual behaviour on a particular machine. This becomes even more evident in the case of generic programming. Here the situation is more complex because there is usually more than one algorithm realizing the desired function, thus in principle a choice of which algorithm is to be instantiated. The optimal choice does not only depend on the domain the actual instantia-

tion is about, such as for example more efficient algorithms for rings without zero divisors than for ordinary rings. The algorithmic behaviour may also depend on the actual parameters with which an instance is called, for example whether a matrix is sparse or dense. Another example is sorting of sequences, where the quicksort algorithm though better on average may be not the optimal choice for a particular sequence.

Thus integrating algorithmic requirements into the choice of algorithm instantiation in generic programming requires more information than available by classical algorithm analysis. In fact, to choose the optimal instantiation run time information is necessary. This idea has been considered in [Kre02] for the case of run time. Here, the run time of an algorithm for a particular input is measured and stored in a library each time the algorithm is called. This allows on the one hand to include characteristics of different machines and platforms, on the other hand the parameters an algorithm is called with: based on these measurements for particular inputs breakpoints for a set of algorithms computing the same functions are predicted using methods from the field of genetic programming. This information then allows to choose an algorithm for a given input at run time by just looking up the computed prediction. Each use of an algorithm provides more information so that the breakpoint prediction can be improved and updated. Note, that this again is a library approach, that is additional knowledge stored in a library is used to support the process of generic programming.

To summarize, algorithmic requirements allow, first, to restrict the range of legal instantiations by including properties beyond correctness of algorithms such as for example efficiency. By formally specifying such requirements, the ability is given to both prove their correctness and to incorporate them into checking for legal instantiations. Secondly, algorithmic requirements can help to chose the optimal instantiation of generic algorithms with respect to parameters of interest. Again formally specifying possible alternatives allows to at least partially incorporate an automatic check, though indeed the sole presentation of such requirements in direct conjunction to algorithms already supports the user when working with generic algorithms.

Chapter 5

Mechanized Reasoning Systems

Generic programming requires a precise specification of type parameters and instantiations in order to guarantee that algorithms provided in fact work properly, that is are not only executable but also correct. Thus generic programming comes with new challenges for the specification and verification of algorithms for which support by mechanized reasoning systems is wishful. However, it is obvious that the amount of formal reasoning required cannot be done at compile time as we have argued in the preceding sections. Verification and at least parts of the instantiation check, for example rules used to perform this check, have to be delegated to external reasoning systems where the knowledge used can be proven correct to increase reliability of generic programming. Thus generic programming calls for a new kind of mathematical management extending pure theorem proving. In this chapter, therefore, characteristics of mechanized reasoning systems necessary to support specification and verification of generic algorithms are identified. Then some mechanized reasoning systems — Imps, Mizar, PVS, and Theorema — are considered under the viewpoint of how they can support the proof problems occurring in generic programming.

5.1 Introduction

Generic programming introduces new abstractions into programming: the parts of the algorithms not essential for the algorithm's method are instantiated later. To make this work instantiations have to fit both syntactic and semantic requirements. These, as we have seen, can be formally specified and attached to generic algorithms and possible instantiations. Thus correctness of generic algorithms and instantiations in principle is provable and hence a

topic for theorem provers and mechanized reasoning systems in general.

Mechanized reasoning systems and in particular automated theorem provers, have reached an impressive level of success. Various systems exist and involved theorems have been proven with machine assistance or even automatically, as for example the Jordan curve theorem [Lün81] or Robbins' theorem [McC97]. Recently a new direction in the field finds more and more attention: management of mathematical knowledge, that is besides the pure theorem proving capability of a system other topics such as, for example, theory development, management of already proven theorems, communication between different systems and also combination with computer algebra systems [Buc96], are taken into account.

We believe that this coincides with the support of specification and verification problems occurring in generic programming: First, most part of theorem proving for generic programming is done off-line, thus not essential for the standard user of a generic programming system. Furthermore, though users are not required to actually prove theorems they should be able to use the results, that is they should be able to understand which knowledge they use for their algorithms is secure due to some theorem proving and which is not. Hence, the capabilities for proving theorems is not the only characteristic a mechanized reasoning system must come with in order to support generic programming. Generic programming in fact demands a new kind of system for mathematical reasoning—and management.

Now, what are the special features of theorem proving and mathematical knowledge management systems with respect to generic programming? First of course the system must enable building abstract entities in order to model type descriptions of generic algorithms. This is present in nearly all mechanized reasoning systems. However the strength of generic programming comes from combining different properties in various ways. We have already seen that in chapter 4 where we discussed the representation of requirements. This should also be reflected in the reasoning system used: Particular properties should be definable not only together with a particular domain but also on their own merit and then be combinable in order to prove theorems as general as necessary for generic algorithms and their instantiations.

This already addresses the second point, the combination of already defined notions. Generic programming consists to a large extent in identifying, extending and combining domains for which generic algorithms work. This allows to express minimal requirements for algorithms. Thus a mechanized reasoning system for generic programming should mirror the flexibility used for developing generic algorithms in the sense that combining domains is possible so that theorems already proven can be easily lifted into the com-

bination. In addition reusing theorems is a major activity when reasoning about generic algorithms. Domains are refined in order to match application areas of generic algorithms and theorems already proven should be easily reusable without affecting their correctness. In particular to prove instantiations of generic algorithms correct, theorems have to be applied in different, usually more restricted domains.

Another important issue is concerned with library facilities, in particular with the application of the knowledge stored, for example to support instantiation checks during compiling. A proven theorem describing properties of a generic algorithm later used to ensure correctness of the algorithm or its instances is thus to be reused heavily. A library of generic programming system, and thus the underlying mechanized reasoning system, has to take this into account. It has to be organized in such a way that this kind of reuse is maximally supported by allowing to state the assumptions of theorems as general as possible. Hence, on the one hand the organization of such a library should not follow the usual principles of using domain as the major structuring characteristic as this contradicts the principle of flexible combination of properties. On the other hand it should be possible to build sublibraries with respect to particular domains to inform the user what knowledge is available for the domain of his interest.

Along with this goes the question of typing and subtyping. Subtyping supports not only reusing theorems by giving conditions under which a general theorem is valid in a different domain, but also the process of building sublibraries by simply extracting theorems valid for domains given by a particular type. Hence, subtyping can also be a support for a less experienced user, which brings us to another important feature of mechanized reasoning systems for generic programming. Because the major concern here is not to prove theorems, but rather to use theorems to improve application of generic algorithms, we have the fact that people using the output of theorem proving are rather less familiar with this area. However, users should understand why generic algorithms they wrote or used are not properly checked, that is they have to work with the theorem proving component of the system. Thus the results should be presented in such a way that inexperienced users can easily understand. What is important here is that this is in fact necessary not if everything works out fine, but in particular if an error occurs, that is if for example an instantiation for a generic algorithm cannot be shown correct. Then the knowledge of the library must be presented, so that the user can imagine why his algorithm is not accepted, that is what properties are missing in order to get a well-working instance of a generic algorithm. The Tecton concept description language and the properties-based approach of

chapter 4 are a step in this direction in the area of specification. As already mentioned types and in particular subtypes can help the user to identify domains in which generic algorithms work properly; for example if a type `ring` happens to be a subtype of a type `group` the user can conclude that generic algorithms working for `group` can be safely applied to algorithms providing the properties of `ring`.

A last topic is concerned with how proofs in a mechanized reasoning system are done. Thereby the question is not so much whether proofs are constructed automatically or interactively, because also automatic theorem provers usually require some kind of tuning in the sense of setting parameters or formulating a number of lemmas in order to prove the main theorem. The question we consider more important is which proof style underlies a reasoning system because the way in which mathematics is done on the machine influences the acceptance of users focussing on applications.

Roughly spoken there are two somewhat orthogonal approaches to include the user interactively in the process of proving. The first one is a procedural style where the user calls strategies to construct new subgoals in the course of proving a theorem. The HOL system [Gor89], for example follows this approach. The advantage is that these strategies introduce some amount of automation into the proof process by incorporating for instance decision or simplification procedures. On the other hand the user needs a deep understanding of the underlying logic, usually some kind of higher-order logic, in order to effectively use the strategies provided. This holds in particular if existing strategies do not seem to be convenient for an application and the user wants to define his own strategies.

The second proof style adopts a completely different view on proving theorems: The user should just state what he wants to prove, and not how the system shall try to prove it. This declarative proof style, that can be found for instance in the Mizar system [RT01a] allows the user to work rather the same way as he would do without formal reasoning support: He formulates the knowledge he wants to be proven leaving the actual proving to the system. Note that this is not automatic theorem proving as the user is involved in the course of proving a theorem because the users' goal will most often not be accepted from scratch. Then the user has to break up his original proof goal into smaller steps using some predefined proof constructs. Of course this requires some experience with the system also in order to estimate which kind of steps the system will accept. However, the advantage is that this kind of proof style follows the natural mathematical style and thus frees the user from using a second language for proving theorems. This probably leads to a greater acceptance of such a system by users not entirely focussing on

theorem proving itself but rather on application of it. On the other hand a pure declarative proof style tends to require rather long technical decompositions of proofs that can be done automatically using some kind of procedure. [Har90] gives a detailed discussion of the proof styles mentioned.

All the points we just addressed influence the choice of a mechanized reasoning system for generic programming. Thereby, the key point is the use of theorem proving results stored in a library by a generic programming system, that is by users and compilers who want to apply these results in order to improve the correctness of algorithms they use, develop and instantiate. This also emphasizes the fact that the expected user will usually not be interested in theorem proving itself at first sight but rather in understanding and applying the results of it. There are numerous mechanized reasoning systems especially for proving theorems, Coq [HKP02], Isabelle/HOL [Gor89, NPW02], Lego [LP92], OBJ3 [GWM+92], Otter [Wos94], to name a few of the most prominent. In the following sections we present some mechanized reasoning systems we believe are good candidates for supporting generic programming. They all come with at least some of the characteristics we mentioned. However, the way in which mathematics is done in these systems strongly differs, reflecting the fact that the choice of a mechanized reasoning system always mirrors the preferred way of doing mathematics.

5.2 The Imps System

Imps [FGT93] is an Interactive Mathematical Proof System that aims at a general purpose tool for doing mathematics in a traditional style. It consists of a database of mathematics—the Initial Theory Library—and a set of tools for applying and extending the knowledge contained in the database. The underlying logic—called Lutins [Gut91]—is a simple type theory with partial functions [Hin97]. Thus terms may be non-denoting, however formulas always result in a standard truth value. Mathematics in Imps is organized as a network of axiomatic theories following the little theories approach. Theories are translated into each other by using theory interpretations, in this way permitting the reuse of theorems in different contexts. Theorem proving in Imps is a combination of automatically applying simplification routines and user-driven interactive deduction. Reasoning at the formula level is largely done automatically, so for example algebraic simplification of polynomials is done by a simplification routine and a decision procedure for linear inequalities is present. In addition, as Imps allows for partial functions, Imps

includes an involved algorithm for definedness reasoning in order to automate definedness checking as much as possible. The algorithm uses (conditional) totality theorems, (un-) conditional sort coercions and theorems about the range of functions. Reasoning at the proof structure level is done interactively, the user calls strategies operating on deduction graphs in this way producing new subgoals. Thus Imps in essence supports a procedural proof style. From the viewpoint of formal support for generic programming the most interesting parts of Imps are the little theories approach and the Initial Theory Library, which are described in more detail below.

Imps provides both the familiar version of the axiomatic method and the little theories approach. Usually all reasoning is done based on one powerful and highly expressive axiom system such as for example Zermelo-Fraenkel set theory. In the little theories version [FGT92], however, a number of theories are used to establish new results. The key point is that additional theorems are first proven within a particular theory and then exported into some kind of main theory. This allows to mirror the way mathematicians work, for example group and field theory have been developed separately from vector space theory though results from the former are heavily used in developing the latter. Logically, in Imps a theory just consists of a language and a set of axioms. Then a network of mathematical theories can be built. Theorems are proven in different theories depending on the amount and kind of mathematics necessary. If required, theorems are transported into other theories where they are to be reused.

However, in a mechanized reasoning system using one theory in the course of developing another one is not as obvious as in pure mathematics. On a machine, the application, that is the fact that theorems of the first theory indeed can be applied in the second theory, has to be made explicit. It is not enough to say "a theorem for groups also holds for fields", last but not least because a field consists of two groups. This meta information has to be made constructive for the reasoning system. In Imps the transportation of theorems from one theory into another is done using so-called theory interpretations [Sho67]. A theory interpretation is a mapping from one theory into another. Thereby the mapping is a purely syntactical one given by the interpretation of sorts and constants of the source theory in the target theory. Thus, formally a theory interpretation from a theory $\mathcal{T}$ over a language $\mathcal{L}$ to a theory $\mathcal{T}$' over a language $\mathcal{L}$ is a pair $\Phi = (\mu, \nu)$, where μ maps sorts of $\mathcal{L}$ to sorts of $\mathcal{L}$' and ν maps constant symbols of $\mathcal{L}$ to constant symbols of $\mathcal{L}$'. Connected with such a mapping are so-called obligations saying in particular that axioms of $\mathcal{T}$ are mapped to theorems of $\mathcal{T}$'. Then, if all these obligations are theorems in the target theory, then the mapping constitutes

a theory interpretation, that is theorems are always mapped to theorems.

The little theories approach with theory interpretations has a number of advantages and applications. First of all, theorem reuse is supported: Using a theory interpretation a theorem can be safely transported into another theory, that is the system ensures that the translated formula really is a theorem in the new theory. In Imps this is called installing a theorem in a different theory. This can be particularly done for abstract and concrete theories. For example, the binomial theorem can be proven in the abstract theory of fields.[1] In the following **K** is the underlying sort of field element, **Z** the sort for the integers; the printing and formatting is exactly as by the TeX facility of Imps.

for every a, b: **K**, n: **Z** implication

- conjunction
  - $1 \leq n$
  - $\neg(a = o_{\mathbf{K}})$
  - $\neg(b = o_{\mathbf{K}})$
- $(a + b)^n = \sum_{j=0}^{n} \binom{n}{j} \cdot b^j \cdot a^{n-j}$.

Now, the real numbers form a field, and thus a theory interpretation from the theory of field to the theory of the real numbers can be easily constructed. Consequently, the usual binomial theorem for the real numbers can be installed and used as if it would have been proven directly for the real numbers. Note, that this in fact enables to develop parameterized theories whose parameters can be instantiated via theory interpretations.

In addition, theory interpretations allow to formalize arguments involving symmetry and duality. This is achieved by creating a theory interpretation from a theory to itself. As an illustration, let $\mathcal{T}$ be the theory of groups with $*$ denoting group multiplication. Then the translation Φ from $\mathcal{T}$ to $\mathcal{T}$ with $\Phi(x, y) := y * x$ is a theory interpretation. The left cancellation law $(x * y = x * z) \longrightarrow y = z$ is mapped to the right cancellation law $(y * x = z * x) \longrightarrow y = z$. Thus it is only necessary to prove the first one to conclude that both are valid in the theory $\mathcal{T}$.

In mathematics the term theory is usually understood in a much broader sense than used here. In this sense the theory of vector spaces not only refers to the theory of one single vector space, as given by the constituting

[1] In fact the theorem can be proven for an even more general domain (compare section 5.3 on the Mizar system and [Sch00a]). However, in the Imps Initial Theory Library it occurs for fields.

set of axioms. It also includes at the very least a theory of vector spaces and mappings between them. A very basic example theorem, for instance is the fact that the concatenation of vector space homomorphisms again is a vector space homomorphism. Note that to formalize such a theorem a family of vector spaces is necessary. Imps allows to build theory ensembles, that is a collection of copies of a base theory, for which theory interpretations from the base theory to each copy are automatically constructed. The above mentioned theorem about vector spaces, for example can be proven in a theory ensemble with $n = 3$. Once this has been done the theorem can be transported to other theory ensembles again using theory interpretation. For instance, the composition theorem can be applied to a single vector space only, by identifying all three occurring vector spaces. Again, such theorems of abstract theories can be easily reused in more concrete ones. To install the example theorem in the theory of the real numbers $\mathbf{R}$, the user has to do little more than specify that all vector spaces map to $\mathbf{R}$ and that the corresponding distance functions in `R` are given by $|x - y|$.

Imps' Initial Theory Library is a collection of theories, theory interpretations and theory constituents, that is definitions and theorems. It is organized in sections, each of which holds a particular amount of mathematical knowledge. These sections can then be loaded into the Imps system. The knowledge available in the library includes real numbers, objects like sets and sequences, abstract mathematical structures and some theories supporting applications of Imps in computer science.

The real numbers are formalized two times in Imps, the first theory being the one of complete ordered fields, the second the one of real arithmetic considered as the working theory of the real numbers used for example as a subtheory of graphs if weighted graphs are to be introduced. The theories are equivalent in the sense that there are two theory interpretations translating one theory into the other; furthermore composition of these interpretations is the identity interpretation. The algebra library is not much developed yet, it consists of monoids, groups (and group actions) and fields. In contrast the analysis library comes with involved structures like metric or normed spaces. In the library for computer science three significant facilities exist: state machine theories, a domain theory for denotational semantics and a facility for defining free recursive data types via model conservative extensions.

Recently based on ideas and mechanisms of Imps a formal framework for managing mathematics [Far01, FvM03] has been proposed. The central notion here is the one of a biform theory which is both an axiomatic theory and an algorithmic theory. Thus a biform theory represents knowledge about

its models both declaratively and procedurally. Expression manipulating algorithms are represented by so-called transformers in this way incorporating algorithmic knowledge into theories. Transformers can include for example normal form algorithms as well as proof rules of a deduction calculus. Thus biform theories allow to argue about both theorems and algorithms thereby using knowledge about one kind to infer new knowledge about the other kind. This approach can be considered as a step towards the integration of deduction and symbolic computation.

5.3 The Mizar System

Mizar stands for both a language for the formalization of mathematics [RT99] as well as for the system for developing and storing mathematics based on the Mizar language [RT01a]. The main goal of its original design was a formal language close to the mathematical jargon supporting the process of writing and reviewing mathematical papers. Therefore the logical basis of Mizar is the system of natural deduction following [Jaś34] which is considered as an adequate reconstruction of proof styles used in mathematical publications.

Furthermore, the Mizar system provides a proof checker for validating scripts written in the Mizar language—so-called Mizar articles—and a large library of mathematical knowledge—the Mizar Mathematical Library (MML)—in which accepted articles are included. The knowledge of the library can be imported into articles easily by a referencing mechanism and thus reused to build new theories in the style mathematicians are used to. The design of the Mizar language as a formal counterpart of mathematical vernacular allows to automatically convert articles into other representations such as LaTeX or Html files better accessible by users searching the library. In the following we describe the most important features of the Mizar language and checker, and the Mizar Mathematical Library in more detail as they are most interesting from the viewpoint of formal support for generic programming.

The Mizar language provides the standard set of first order logical connectives for constructing formulae and syntactic constructs for presenting proofs based on natural deduction. These constructs are connected with corresponding (English) natural language phrases so that definitions, theorems and proofs are given in textbook style. In addition Mizar provides means for second order formulae, so-called schemes, with which for instance various induction schemes can be expressed. The axiomatic basis used for Mizar is Tarski-Grothendieck set theory, a variant of Zermelo-Fraenkel set

theory using Tarski's axiom on arbitrarily large, strongly inaccessible cardinals [Tar38, Tar39] instead of the axiom of choice. However, the Mizar language in principle is independent of the underlying axioms so that other axiom systems such as for example Peano axioms or Barnay-Goedel set theory could be chosen.

The Mizar proof checker verifies the individual proof steps given in a Mizar article. To do so, the notion of an obvious inference [Dav81] is employed. However, using only the basic inference rules of natural deduction as obvious would result in very long proofs even for simple theorems. Therefore the notion of obvious inference has been extended in Mizar: each proof step can be decorated with labels referencing definitions, proof steps or completed proofs occurring in the actual article or in other parts of the Mizar Mathematical Library that have been imported into the article. The definitions and theorems referenced this way are considered as additional premises for the proof step they are attached to. Note that this referencing mechanism allows to incorporate every definition or theorem contained in the Mizar Mathematical Library. The checker then tries to refute the conjunction of these premises and the negated proof goal. The emphasis of the checker is on processing speed rather than on proof power. As a consequence, logical valid proof goals may be not accepted. Then the user has to add more premises or to split up the proof goal. For this purpose Mizar provides a number of proof constructs such as for instance, case distinction, reasoning by contradiction and induction using schemes.

The most remarkable feature of the Mizar language, however, is its type system and the mechanisms to enrich and automate this type system. In Mizar types are introduced using type constructors, so-called modes; for example `set`, `Function` or `FinSequence` are modes. Also structures, that is collections of carriers and operations can be introduced using so-called structure modes. Thereby the user has to ensure that a mode defined is non empty by giving an existence proof. Mizar types can be parameterized, thus defining for example the mode `Subset of S`, where the type of `S` must widen to `set`, or the mode `VectorSpace over R`, where the type of `R` can be required to widen to `Ring` or even `Field`. The key point is that modes can be extended using attributes: An attribute defines a property an object of an already defined mode may have or not. By combining such an attribute with the mode over which it is defined a new mode is constructed, and a hierarchy of types (with `set` as its root) is built which the proof checker can automatically use. So for example abelian groups are constructed by introducing an attribute `Abelian` and combining this with the mode `Group` of all groups into the new one `AbelianGroup`. Then in particular `Group` becomes an an-

cestor of `AbelianGroup`. As a consequence, all theorems (and definitions) for groups in particular are automatically valid for abelian groups because the mode `AbelianGroup` widens to the mode `Group`. For an illustration consider again the binomial theorem. It can be proven in general for a domain providing addition, multiplication, a zero and a unit (given by the structure mode `doubleLoopStr`) fulfilling a number of attributes [Sch00a]:

```
theorem T1:
for n being Nat,
    L being Abelian add-associative left_zeroed
            add-cancelable associative commutative
            unital distributive (non empty doubleLoopStr),
    a,b being Element of L
holds (a+b)|^n = Sum((a,b) In_Power n);
```

To define finite sums in Mizar finite sequences are employed, whose elements are then summed up using the functor `Sum`. Here `(a,b) In_Power n` is the finite sequence of length `n+1` where the i-th element corresponds to i-th element of the binomial sum. Note that in the theorem the property of providing inverses with respect to addition has been replaced with the weaker requirement of cancelability. Hence the theorem is not only valid for the real but also for the natural numbers. Now to apply theorem `T1` in a different domain, it is only necessary that the domain's attributed type widens to the one used in `T1`. For example, a field comes with all the properties mentioned in `T1`. In Mizar the mode `Field` can be introduced by combining the attributes describing the usual axioms of a field as follows.

```
definition
cluster Abelian add-associative right_zeroed well-unital
    right_complementable associative commutative
    distributive Field-like (non empty doubleLoopStr);
end;

definition
mode Field is
    Abelian add-associative right_zeroed well-unital
    right_complementable associative commutative
    distributive Field-like (non empty doubleLoopStr);
end;
```

The (existential) cluster definition is necessary here because in Mizar non empty modes are not allowed; the user has to prove that an object with all

claimed properties indeed exists. Now, theorem `T1` calls for the attribute `add-cancelable` not mentioned in the definition of `Field`. It would not be convenient if the user had to include this attribute (which is implied by the ones already given) in the definition of `Field`. Instead Mizar offers the possibility to teach the type checker that this implication holds by using (conditional) cluster definitions. Note that such a cluster is not restricted to a particular definition, but holds in general. Thus once such a cluster is proven, Mizar's type hierarchy is extended and automatically used by the checker when trying to apply theorems.

```
definition
cluster add-associative right_zeroed right_complementable
    -> add-right-cancelable (non empty LoopStr);
cluster Abelian add-right-cancelable
    -> add-left-cancelable (non empty LoopStr);
cluster add-left-cancelable add-right-cancelable
    -> add-cancelable (non empty LoopStr);
end;
```

Of course these clusters also require a (coherence) proof showing the stated implication. Now, the following theorem is automatically accepted by just referencing theorem `T1` because using the above cluster definitions the type `Field` widens to the one used in `T1`.

```
theorem
for n being Nat,
    L being Field,
    a,b being Element of L
holds (a+b)|^n = Sum((a,b) In_Power n) by T1;
```

More concrete domains like for example the natural numbers can be handled similarly. We consider the natural numbers as an example. Here first the structure of the natural numbers—called `Nat.Ring`—has to be defined as a `doubleLoopStr` by providing the set of natural numbers `NAT` and defining natural addition and multiplication. Note the difference between the set of natural numbers `NAT`[2] and the algebraic structure of the natural numbers given by `Nat.Ring` where the usual operations on the natural numbers are introduced. Then attributes, that is properties, the natural numbers fulfill are proven in functorial clusters. Again using these clusters the Mizar checker automatically infers that the attributes mentioned hold for `Nat.Ring` in this way extending `Nat.Ring`'s type.

[2]The type `Nat` actually is an abbreviation for the type `Element of NAT`.

```
definition
cluster Nat.Ring ->
    Abelian add-associative right_zeroed well-unital
    add-right-cancelable associative commutative
    distributive (non empty doubleLoopStr);
end;
```

Then the binomial theorem for natural numbers can be inferred from the general theorem `T1` by just referencing it. Note that the clusters concerning cancelability from above are again involved in the course of checking that that the type of the following theorem widens to the one of `T1`

```
theorem
for n being Nat,
    a,b being Element of <NAT,+,*,0,1>,
holds (a+b)|^n = Sum((a,b) In_Power n) by T1;
```

This mechanism of type widening based on attributes allows to easily reuse theorems in different domains because Mizar's checker can automatically apply them if the type properly widens. Note that no translation from one theory into another is necessary as in Imps. Also attributes can be combined in various ways, thus allowing to formulate theorems using only properties necessary o prove them. On the other hand, however, similar attributes have to be defined for different operations separately; so in the example we have `add-associative` for `+` and `associative` for `*`.

The Mizar Mathematical Library (MML) is a collection of Mizar articles.[3] As of this writing the library consists of more than 700 articles with about 40,000 theorems. Currently, the development of the library is the main activity in the Mizar project. Efforts have been concentrated on the theory of continuous lattices [GHK$^+$80], the proof of the Jordan curve theorem [Lün81], abstract reasoning about computations [NT92] and algebra towards symbolic computation following [BW93].

The knowledge stored in the library can be used as the basis for writing new articles. Therefore a local environment for each article is constructed. This is realized by referencing (names of) articles contained in the library in which definitions, theorems and schemes the user wants to rely on are included. Different environment directives such as `vocabulary`, `constructors`,

[3]Before an article is included in the library it is reviewed by the Mizar Library Committee. The committee also revises and updates the library if a new version of the Mizar software is published.

`theorems` or `clusters` import different kinds of knowledge into the article. The building of a local environment allows to restrict the knowledge of the library to those parts necessary for the actual article without forcing the user to import every necessary notion individually. Note that the checker only accesses the local environment, it does not communicate with the library itself. This addresses the problem of how to reuse particular knowledge of a huge data base without bothering the user with too many details when importing the knowledge.

In the same direction goes a recent development within the Mizar project. There are a number of examples for reasonings, for instance calculating with the real numbers, that based on natural deduction and obvious inferences tend to be long and tedious. To overcome this problem a new environment directive `requirements` has been added. Possible requirements are for example `BOOLE`, `REAL` or `SUBSET`. If imported, these requirements enable the checker to deduce a number of facts automatically, that is without referencing a theorem. For example, `x + 0 = x` for a real number `x` can be automatically inferred, if the requirement `REAL` is included. This can be seen as providing a general purpose checker that can be strengthened if necessary, that is if a particular application area is considered.

5.4 The PVS System

PVS [ORS92] is a prototype system for writing specifications and constructing proofs. Based on simply typed higher-order logic [Hin97] its type system has been extended with subtypes and dependent types. However, type checking for PVS is undecidable in general. Therefore proof obligations implying correct typing are automatically generated which then have to be proven with the PVS proof checker. The proof checker is a combination of decision procedures and primitive inference steps. Decision procedures include for example the theory of linear arithmetic, arrays and tuples. Based on primitive inference steps high-level strategies similar to tactics in LCF [GMW79] can be defined. The PVS system comes with an initial library containing besides basic notions like sets, relations, functions and numbers, theories from different areas, for example graphs and digraphs, basic algebraic structures, an introduction to analysis and even formalizations of arrays, bitvectors and fixpoint theory. In the following we consider predicate subtyping and dependent typing in more detail, in particular their impact on proving theorems. We also have a look at how the PVS system deals with theories and theory instantiation.

Predicate subtyping allows to naturally handle partial functions in a framework containing total functions only. For example, the division operator / for rational numbers is introduced by first defining a predicate subtype posrat of the natural numbers serving as the type for the second argument of /. Then each time the operator / occurs, PVS automatically generates a proof obligation stating that the second argument of / is not equal to 0 taking into account the logical context in which this argument appears. Thereby, quite a number of these proof obligations are discharged automatically by subsumption or applying basic inference steps.

In addition, predicate subtyping is a powerful method to encode knowledge in an object's definition used later for proving. For example, injective functions can be defined as a subtype of functions from D to R using a higher-order predicate injective?. Note that functions in fact is a theory parameterized by the types D and R.

```
functions [D,R: TYPE]: THEORY
BEGIN
 f: VAR [D->R]
 x, y: VAR D
 injective?(f): bool =
                 (FORALL x, y: (f(x) = f(y) => (x = y)))
 injection: TYPE = (injective?)
END functions
```

Then, using the (predicate) subtype even of even numbers, the function double can be declared as an injective function from the natural to the even numbers:

```
even: TYPE = {i : nat | EXISTS (j : nat): i = 2 *j }

double : injection[nat, even] = (LAMBDA (i : nat): 2 * i)
```

Now, when double is type checked two proof obligations are generated. The first one states that the result computed by double is even, the second one that double fulfills the predicate injective?, e.g. is an injective function. Both are proven automatically by the PVS proof checker, however more complicated proof obligations may require some interaction with the user. Conversely, this type information is used in PVS theorem proving: Constraints given by predicate subtypes are automatically asserted to decision procedures, thus using the additional type information such as injective? to support theorem proving.

Another advantage of predicate subtypes is that they enable the definition of dependent types, that is a type can depend on the value of a parameter.[4] This allows to capture relationships existing between the input and the output of a function. So for example `upto(n)` is a predicate subtype of the natural numbers giving the starting segment up to `n`.

```
upto(n) : TYPE = {s: nat | s <= n}
```

Then binomial coefficients $\binom{n}{k}$ can be properly defined: Possible values of the second argument `k` depend on the actual value of `n` as we usually have $0 \leq k \leq n$. This is expressed by using the type `upto(n)` for the second argument of the function `chooses`:

```
chooses(n, (k: upto(n))) : posnat =
   factorial(n)/(factorial(k) * factorial(n-k))
```

The type of `chooses` is a dependent tuple type where the type, and hence the possible values, of the second component `k` depends on the first component `n`. Note that this typing corresponds exactly to the informal restriction for `k` given in textbook definitions. The definition of `chooses`, when type checked, generates a proof obligation concerning the return type `posnat`. To prove that `choose` indeed returns a non-negative integer requires the proof of the well-known recurrence for binomial coefficients as a lemma.

Predicate subtypes and dependent types are especially useful in the area of computer algebra where for example an inversion function in residue class rings $\mathbb{Z}_n$ exists if and only if $\mathbb{Z}_n$ is a field if and only if n is prime. Such constraints are naturally expressed using these mechanisms.

Theories in PVS group together type declarations, axioms, definitions, and theorems. Theories may be parameterized in this way allowing to import an instance of a theory, for example the integer instance of a general theory group, into another. However, when a theory is imported actual parameters for either all or none formal parameters have to be supplied. Assumptions about the formal parameters can be included, then each time the theory is imported proof obligations for the (instantiated) assumptions are generated. This for example allows to state that the theory of residue class rings $\mathbb{Z}_n$ should only be imported, if n is prime, that is if $\mathbb{Z}_n$ is a field. Consequently, a general theory of groups can be defined in two ways. Firstly, axioms can

[4] And, vice versa, using predicate subtypes is the only way to define dependent types in PVS.

be given in the theory body. Then importing an instance of the theory adds additional axioms, namely the group axioms, to the particular instantiation. In particular, no proof obligations are generated, that is there is no need to prove that the instantiated axioms are indeed valid. Secondly, the group axioms can be introduced as assumptions on the theory's parameters. Then, when instantiation takes place, proof obligations are generated, that is the user has to prove that the instance indeed constitutes a group. Note, that in this case the group axioms do not explicitly appear in the instance.

Using a mechanism called theory declaration, PVS enables to define theories with theories as parameters. This allows to develop theories in which different copies of the same theory are used. Thus, for example a theory of group homomorphisms can be introduced with `G1` and `G2` as parameters of "type" `group`. Note that `group` actually is a theory name as indicated by the keyword `THEORY`, and not a type.

```
group_homomorphism[G1, G2 THEORY group]: THEORY
BEGIN
 x, y: VAR G1.G
 f: VAR [G1.G -> G2.G]
 homomorphism?(f): bool = FORALL x,y: f(x+y) = f(x) + f(y)
END group_homomorphism
```

Then `G1` and `G2` are parameters that may be instantiated with two different groups, for instance `G1` with the integers and `G2` with the set of nonzero real numbers, that is with two distinct versions of the theory `group`. Note that without theory declarations it would have been necessary to duplicate the group specification, because just importing a theory twice is the same as importing it once, thus the above definition would change into an automorphism.

Theory interpretations which have already been described in section 5.2 are also available in PVS [OS01]. They are realized as mappings for uninterpreted types and constants of the source theory into the current theory. To actually constitute a theory interpretation such a mapping must fulfill two further requirements. First, the mapping must be consistent, that is if a type `T` is mapped to a type expression `E`, then a constant `t` of type `T` must be mapped to an expression `e` of type `E`. Second, the translation of the source theory's axioms must result in theorems of the target theory. For these, the PVS checker generates proof obligations when processing the target theory.

Note that theory interpretations can be used to improve instantiation of theories. Here, the actual parameters are given by means of the interpreta-

tion. However, using mappings not all entities must be interpreted in contrast to parameterized theories where either all or none of the parameters must be instantiated. Thus a theory interpretation allows to instantiate part of a theory leaving some types or constants uninterpreted. Consider for example, a theory containing a maximal `max` element for a subset `S` of elements of a type `T`. It may be desirable to instantiate this theory with the real numbers for `T` without being forced to say what the maximal element (or the subset of the reals) is in detail, thus leaving `max` and `S` as parameters. This can be done by using a theory interpretation mapping only `T` to the real numbers while leaving `max` and `S` uninterpreted.

5.5 The Theorema System

Theorema [BJK+97, Buc01b] is a system that aims at supporting proving, solving and simplification of theorems, or more generally formulas. Proving, solving and simplification are identified as the three basic activities of formal mathematics and, hence, symbolic computation in [Buc96]. Solvers try to find substitutions that make a formula true, whereas simplifiers transform a formula into an equivalent simpler one. Thus the Theorema system is an environment in which computing and reasoning is combined, that is in particular algorithmic mathematics is taken into account. This is also reflected in the Theorema language, a version of higher-order predicate logic with a syntax resembling common mathematical style. As a consequence, Theorema formulas can be likewise processed by provers, solvers or simplifiers.

The current version of Theorema is implemented in the Mathematica programming language, which means that Theorema is available if Mathematica [Wol96] is installed. Therefore the Theorema system can be considered as an extension of a current mathematical software system by facilities for logical and mathematical reasoning and for mathematical knowledge management in general.

The Theorema system comes with a collection of provers and proof strategies. In addition to a general higher-order theorem prover Theorema provides a number of special provers, solvers, and simplifiers that can be explicitly called by the user during a proof attempt. For instance, the so-called PCS proof method [Buc01a] can be applied. PCS actually is a proof heuristic that, roughly spoken, iterates through a proving, a computing (simplifying), and a solving phase. There is evidence that the PCS proof method is particularly helpful for formulas with alternating quantifiers, such as for instance

the notion of limits in analysis. The following theorem about addition of limits for example is proven automatically by Theorema if PCS is chosen as the proof method (and if the basic definitions and some lemmas concerning limits are included by referring to the appropriate theory).

Proposition["limit of sum", any[f,a,g,b],
(limit[f,a] ∧ limit[g,b]) ⇒ limit[f+g,a+b]]

Thereby a proof is constructed and returned consisting of the formulas generated during the proofs enhanced by descriptive English text. Thus Theorema proofs, though tending to be quite long, are easily human readable. Other special internal provers of Theorema include a set theory prover, a Gröbner bases prover, and induction provers for equalities. In addition other provers such as for instance Otter [Wos94] can be incorporated in Theorema; if such an external prover is recorded in Theorema's library of provers, a call to it is nothing else then selecting it as a proof method. Thus strong special purpose provers can be embedded into the Theorema system. However, proofs originating from external provers are not as human readable as the ones from Theorema's internal provers which emphasize besides proof power also readability and naturalness.

Theories can be built in Theorema easily by grouping together definitions, lemmas and propositions using the construct **Theory**. Each theory is given a name so that it can be included as a backbone when calling a proof method. For example, a basic theory for limits is given by

Theory["limit",
Definition["limit"]
Definition["+"]
Lemma["| + |"]
Lemma["max"]]

Note, that the definition of a theory may again include theories. Thus hierarchical theories can be developed, in this way building up a knowledge base for mathematics. Theories can be included into proof attempts in this way extending the chosen proof strategy. So, for example using theory "limit" together with the PCS proof method as follows

Prove[Proposition["limit of sum"], using → Theory["limit"], by → PCS]

results in a completely automatic proof of the example proposition "limit of sum" from above with a detailed proof description in common mathematical textbook style. Note again that external provers contained in Theorema's

prover library can be called the same way as PCS in our example. Also computational knowledge, that is algorithms, from the underlying Mathematica system can be incorporated using the **Built-in** construct.

From our point of view the most interesting part of the theorema system is the use of functors. Functors are essentially descriptions of how to uniformly construct new domains out of given ones. Theorema functors are based on the functor construct of ML [Pau96] and concern not only carriers and operations but also predicates. So, for instance, the well-known construction of the field of fractions can be formulated as a Theorema functor as follows.

Definition["fractionfield", any[C],
fractionfield[C] = Functor[D, any[r,i,xr,xi,yr,yi],
$\epsilon_{\mathrm{D}}[\langle r, i\rangle] \Leftrightarrow \epsilon_{\mathrm{C}}[\mathrm{r}] \wedge \epsilon_{\mathrm{C}}[\mathrm{i}]$
$0_{\mathrm{D}} = \langle\ 0_{\mathrm{C}},\ 0_{\mathrm{C}}\ \rangle$
$\langle$ xr,xi $\rangle +_{\mathrm{D}} \langle$ yr,yi $\rangle = \langle$ xr$+_{\mathrm{C}}$yr, xi$+_{\mathrm{C}}$yi $\rangle$;
$\vdots$
]];

Definition["$\mathbb{Q}$",
$\mathbb{Q}$ = fractionfield[$\mathbb{Z}$]]

Given a domain `C` the functor "fractionfield" constructs the field of fractions `D` of `C`, provided that `C` comes with the necessary operations. Once the functor "fractionfield" is installed, it can be applied to particular domains to generate fields of fractions in tuple representation. In the example this application is shown for the domain of integers $\mathbb{Z}$; which of course requires that $\mathbb{Z}$ is already defined. Note, how the operations (and the ϵ predicate) of the domain `D` are reduced to the corresponding operations of `C`. Because Theorema also supports algorithms, mainly by importing them from Mathematica, "fractionfield" can be instantiated with a domain `C` for which algorithms are provided. Then the domain `D` constructed by "fractionfield" also comes with algorithms for its operations by just invoking the corresponding algorithms of `C`. As already mentioned importing from Mathematica is done using the **Built-in** construct. Thus, because algorithms for $\mathbb{Z}$ are provided, the following imports a computable version of the rational numbers $\mathbb{Q}$.

Use[$\langle$Built-in["Numbers"], Built-in["Tuples"], ..., Definition["$\mathbb{Q}$"]$\rangle$]

Note, that among others also algorithms for basic operations of tuples have to be imported. After this declaration, for example

$$\text{Compute}[\langle 1,2\rangle +_{\mathbb{Q}} \langle 4,3\rangle]$$

is evaluated to $\langle 5,2\rangle$ by invoking the functor's definitions followed by applications of the imported algorithms for tuples and the integers.

What makes Theorema's functor concept attractive is the possibility to prove theorems about functors. To be more precise, properties of the domains constructed can be shown assuming properties of the instantiated domain. Note that in the definition of "fractionfield" no algebraic properties of `C` are assumed. Thus the functor works for arbitrary domains coming with the operations required; nevertheless the resulting domain `D` is a field only if `C` is an integral domain. This can be stated in Theorema as a theorem about the functor: Theorema allows to define predicates over domains; these predicates in particular may use operations defined for the domain involved. Then, for example, such a predicate can describe the axioms of an integral domain or a field, or even individual algebraic properties. Using such predicates theorems can be stated and proven. For example, the just mentioned relationship between `D` an `C` reads as follows.

Proposition["ff", any[C],
isIntegralDomain(C) → isField(fractionfield[C])]

Thus Theorema functors support proving based on properties of the underlying domain: For a given construction or algorithm theorems can be shown using as less assumptions of the domains involved as possible. Checking whether such a theorem holds for a particular domain then consists of checking whether these assumptions are fulfilled. For example because $\mathbb{Z}$ is an integral domain $\mathbb{Q}$, the field of fractions for $\mathbb{Z}$, is indeed a field, in which hence inverses can be computed. Again this can be done algorithmically based on algorithms for the integers $\mathbb{Z}$.

Chapter 6

Applications

In chapter 4 when discussing the representation of semantic requirements we presented an approach focussing on the individual properties of operations involved and argued that this approach can be used to reduce the check for legal instantiations of generic algorithms to a comparison of such properties with respect to inclusion. In this chapter we will both further elaborate the approach's use in generic type checking and illustrate further applications such as the verification of generic algorithms and the design of libraries in general.

6.1 Overview

Focussing on individual properties of operations allows to state minimal requirements necessary for generic algorithms in a very flexible way. However, checking for legal instantiations of generic algorithms based on inclusion of properties only turns out to be too restrictive. Thus we first present a generalization of this idea incorporating symbolic deduction so that logical consequences of sets of properties can be computed (see [Sch02]). As we will see, this can be used not only for legal instantiation checking in generic programming but also in general for developing libraries to give them a more active role in managing the knowledge included.

Then we present generic type checking based on sets of properties in more detail. We provide a small programming language relying on the discussion in section 4.4 (compare [GS03]). Generic type checking for this language is based on the calculus just mentioned and is realized using the system for generating type checkers presented in [Gas03]. We give some running examples illustrating the ideas.

Verification of generic algorithms is also addressed. Using properties as assumptions about possible instantiations generic algorithms can be verified on a very abstract level (compare [Sch00b, Sch03]). This allows to prove correctness of an algorithm with respect to its minimal requirements, thus for a whole class of possible instantiations. Adding more assumptions, also more special versions of the algorithm incorporating particular knowledge about the now restricted class of instantiations can be shown correct.

Finally we briefly explain how the properties-based approach supports the design of libraries in general, that is not only libraries of algorithms but also for instance mathematical libraries, by adding a component based on the calculus presented (compare [Sch01b, Sch02]). This allows to state facts in a general manner and the library to conclude that these facts hold in particular cases the user wants to consider. Thus the library itself is given a more active role than only holding static knowledge.

6.2 A Calculus for Deducing Properties

Let $P_{\mathtt{I}}$ and $P_{\mathtt{A}}$ be two sets of properties, that is sets of predicates describing semantic properties of operations. In section 4.4 such sets were used to check whether particular instances are legal for a generic algorithm by comparing sets of properties, that is if $P_{\mathtt{I}}$ are the properties of the instantiation `I` and $P_{\mathtt{A}}$ the properties attached to the generic algorithm `A` describing the generic type parameter `T` of `A` we have

$$Leg(\mathtt{I}, \mathtt{A}) \ :\Longleftrightarrow \ Sig(\mathtt{A}) \subseteq \mathrm{S}ig(\mathtt{I}) \ \wedge \ P_{\mathtt{A}} \subseteq P_{\mathtt{I}}.$$

Thus, the distinction between the set of operations a type parameter requires and properties of these operations necessary to make the algorithm work as expected allows for checking legality of instantiations. Sets of properties must only be compared with respect to inclusion, namely the set of properties describing the semantic requirements of `A` with the set of properties `I` comes with. However, this setting is too restricted. For example, if a generic algorithm `A` requires the property `left-distributive`, and an instantiation `I` obeys the property `distributive`, it is not desirable that `left-distributive` has to be added to the instantiation's properties. It should rather be possible to conclude that `I` is legal for `A`, although the sets of properties involved are not related by inclusion.

Following [Sch02] we now replace the subset relation between two sets P_1 and P_2 of properties used in section 4.4 by a relation $P_1 \Longrightarrow P_2$ with the meaning that every domain D that fulfills the properties of P_1 also fulfills the ones in P_2, or more formally

$$\models P_1 \Longrightarrow P_2 \quad :\Longleftrightarrow \quad \forall D : D \models P_1 \text{ implies } D \models P_2$$

where $\models$ on the right-hand side is the model operator well-known from first-order logic. Consequently an instantiation $\mathtt{I} = (Sig(\mathtt{I}), Props(\mathtt{I}))$ is legal for a given generic algorithm $\mathtt{A} = (Sig(\mathtt{A}), Prop(\mathtt{A}))$ if both $Sig(\mathtt{A}) \subseteq Sig(\mathtt{I})$ and $\models Props(\mathtt{I}) \Longrightarrow Props(\mathtt{A})$ are valid. Note that $\models P_1 \Longrightarrow P_2$ corresponds to the usual semantic implication. However, in this special case the formulas occurring in P_1 and P_2 are given by a set of predicates only, whose arguments are either variables or constants.

Obviously, an implication $P_1 \Longrightarrow P_2$ cannot be checked in this generality, because this would require arbitrary difficult theorem proving. Therefore we incorporate a set of rules L describing basic relations between sets of properties. For example, the following rule

$$\begin{array}{l} \{\mathtt{distributive}(R, +, *)\} \longrightarrow \\ \{\mathtt{left\text{-}distributive}(R, +, *),\ \mathtt{right\text{-}distributive}(R, +, *)\} \end{array} \qquad (\mathbf{A})$$

states that structures $(R, +, *)$ that are `distributive` are also both `left-` and `right-distributive`. Thereby the arguments of the predicates are interpreted as variable symbols. This allows to include properties of particular domains, for example

$$\mathtt{distributive}(\mathbb{Z}, +_{\mathbb{Z}}, *_{\mathbb{Z}})$$

where the arguments are now interpreted as constant symbols. Note however, that the formal definition of the predicates is not incorporated in the rules; thus correctness of rules and particular properties of special domains is delegated to some external reasoning system. In this way additional knowledge about the problem domain is provided. The deduction of $\models P_1 \Longrightarrow P_2$ is then performed relative to such a set of rules.

The calculus has two axioms. The first mirrors the fact that an implication $P_1 \Longrightarrow P_2$ trivially holds, if $P_2 \subseteq P_1$. The second axiom allows to incorporate the external rules: if $l \longrightarrow r \in L$, then $\sigma(l)$ implies $\sigma(r)$ where σ is an arbitrary substitution compatible with the signature. Further on, there

is a rule allowing to combine different implications P_2 and P_3 both made from P_1 and a rule for concatenating implications $P_1 \Longrightarrow P_2$ and $P_2 \Longrightarrow P_3$.

$$\frac{P_2 \subseteq P_1}{\vdash P_1 \Longrightarrow P_2} \qquad (\mathbf{AX1})$$

$$\frac{l \longrightarrow r \in L}{\vdash \sigma(l) \Longrightarrow \sigma(r)} \qquad (\mathbf{AX2})$$

$$\frac{\vdash P_1 \Longrightarrow P_2, \ \vdash P_1 \Longrightarrow P_3}{\vdash P_1 \Longrightarrow P_2 \cup P_3} \qquad (\mathbf{R1})$$

$$\frac{\vdash P_1 \Longrightarrow P_2, \ \vdash P_2 \Longrightarrow P_3}{\vdash P_1 \Longrightarrow P_3} \qquad (\mathbf{R2})$$

Provided that the rules in L are correct, that is if from $l \longrightarrow r \in L$ indeed follows $\models \sigma(l) \Longrightarrow \sigma(r)$ for the substitutions σ used in a deduction, it is straightforward to see that the calculus is correct. In other words, we have that (relative to L) $\vdash P_1 \Longrightarrow P_2$ implies $\models P_1 \Longrightarrow P_2$. However, if no deduction sequence is found this does not necessarily mean that $\models P_1 \Longrightarrow P_2$ is not valid. The reason is that the calculus checks for implications with respect to the rule set L only. In other words, if L does not contain enough knowledge about the problem domain, the deduction of an implication may fail, although this implication is true.

The calculus can be extended with some straightforward derived rules, among them

$$\frac{\vdash P_1 \Longrightarrow P_2 \cup P_3}{\vdash P_1 \Longrightarrow P_2} \qquad (\mathbf{L1})$$

$$\frac{\vdash P_1 \Longrightarrow P_2, \quad P_1 \subseteq P_3}{\vdash P_3 \Longrightarrow P_2} \qquad (\mathbf{L2})$$

These rules can be easily proven correct in the sense that their consequences can be deduced from their premises in the original calculus. To see how the calculus works let us deduce the following implication.

$$\begin{aligned} \vdash \ & \{\texttt{associative}(\mathbb{Z}, +_{\mathbb{Z}}), \ \texttt{distributive}(\mathbb{Z}, +_{\mathbb{Z}}, *_{\mathbb{Z}})\} \Longrightarrow \\ & \{\texttt{associative}(\mathbb{Z}, +_{\mathbb{Z}}), \\ & \ \texttt{left-distributive}(\mathbb{Z}, +_{\mathbb{Z}}, *_{\mathbb{Z}}), \ \texttt{right-distributive}(\mathbb{Z}, +_{\mathbb{Z}}, *_{\mathbb{Z}})\} \end{aligned}$$

To do so, we assume that the distributivity rule **A** from above is present in the set of rules L. Then, using Lemma **L2**, we get the following deduction

sequence. Note that the actual definition of the properties involved has no influence on the deduction, that is the deduction is purely symbolic.

(1) $\vdash \{\texttt{associative}(\mathbb{Z}, +_{\mathbb{Z}}),\ \texttt{distributive}(\mathbb{Z}, +_{\mathbb{Z}}, *_{\mathbb{Z}})\} \implies \{\texttt{associative}(\mathbb{Z}, +_{\mathbb{Z}})\}$
by **AX1**

(2) $\vdash \{\texttt{distributive}(\mathbb{Z}, +_{\mathbb{Z}}, *_{\mathbb{Z}})\} \implies \{\texttt{left-distributive}(\mathbb{Z}, +_{\mathbb{Z}}, *_{\mathbb{Z}}),\ \texttt{right-distributive}(\mathbb{Z}, +_{\mathbb{Z}}, *_{\mathbb{Z}})\}$
by **AX2** with **A** and $\sigma(R) = \mathbb{Z},\ \sigma(+) = +_{\mathbb{Z}},\ \sigma(*) = *_{\mathbb{Z}}$

(3) $\vdash \{\texttt{associative}(\mathbb{Z}, +_{\mathbb{Z}}),\ \texttt{distributive}(\mathbb{Z}, +_{\mathbb{Z}}, *_{\mathbb{Z}})\} \implies \{\texttt{left-distributive}(\mathbb{Z}, +_{\mathbb{Z}}, *_{\mathbb{Z}}),\ \texttt{right-distributive}(\mathbb{Z}, +_{\mathbb{Z}}, *_{\mathbb{Z}})\}$
by **L2**(2)

(4) $\vdash \{\texttt{associative}(\mathbb{Z}, +_{\mathbb{Z}},\ \texttt{distributive}(\mathbb{Z}, +_{\mathbb{Z}}, *_{\mathbb{Z}})\} \implies \{\texttt{associative}(\mathbb{Z}, +_{\mathbb{Z}}),\ \texttt{left-distributive}(\mathbb{Z}, +_{\mathbb{Z}}, *_{\mathbb{Z}}),\ \texttt{right-distributive}(\mathbb{Z}, +_{\mathbb{Z}}, *_{\mathbb{Z}})\}$
by **R1**(1,3)

Thus a generic algorithm requiring the properties $\texttt{associative}(R, +)$ and $\texttt{right-distributive}(R, +, *)$ can in particular be instantiated with the integers $\mathbb{Z}$. Though for $\mathbb{Z}$ only the property $\texttt{distributive}(\mathbb{Z}, +_{\mathbb{Z}}, *_{\mathbb{Z}})$ has been stated, this can be validated by a type checker using the calculus. Note that, by just taking the identity substitution for σ, the above sequence can be easily transformed in a deduction sequence using $R, +$ and $*$ instead of $\mathbb{Z}, +_{\mathbb{Z}}$ and $*_{\mathbb{Z}}$, that is general lemmas can be shown.

Finding a deduction sequence for an implication $P_1 \implies P_2$ requires some amount of guessing in which way the set on the left-hand side of an implication has to be extended, that is which property should be additionally considered in order to combine already deduced implications. This can be seen, for example, in step (3) of the deduction above where using Lemma **L2** the set on the left-hand side is extended from $\{\texttt{distributive}(\mathbb{Z}, +_{\mathbb{Z}}, *_{\mathbb{Z}})\}$ to $\{\texttt{associative}(\mathbb{Z}, +_{\mathbb{Z}}),\ \texttt{distributive}(\mathbb{Z}, +_{\mathbb{Z}}, *_{\mathbb{Z}})\}$; any other extension would have been a correct application of **L2**, too. Fortunately, this problem can be avoided using backward propagation similar to Prolog [CM87, Col85]. The idea is, given an implication $P_1 \implies P_2$, to successively remove properties from P_2 that are implied by the ones from P_1 until P_2 has been transformed into the empty set. We use the following three rules.

(B1) Replace $\vdash P_1 \implies P_2$ by $\vdash P_1 \implies P_2 \backslash (P_1 \cap P_2)$.

(B2) Replace $\vdash P_1 \Longrightarrow P_2$ by $\vdash P_1 \Longrightarrow (P_2 \backslash (\sigma(r) \cap P_2)) \cup \sigma(l)$ if there are a rule $l \longrightarrow r \in L$ and a substitution σ with $\sigma(r) \cap P_2 \neq \emptyset$.

(B3) Accept $\vdash P_1 \Longrightarrow \emptyset$.

Thus an implication $P_1 \Longrightarrow P_2$ is accepted, if it can be transformed into an implication of the form $P_1 \Longrightarrow \emptyset$. The rules presented are correct with respect to the above calculus in the sense that every deduction starting with $P_1 \Longrightarrow P_2$ and ending with $P_1 \Longrightarrow \emptyset$ using **B1** - **B3** can be translated into a correct sequence of the original calculus. For the example deduction from above we get

$\vdash$ $\{\texttt{associative}(\mathbb{Z}, +_{\mathbb{Z}}),\ \texttt{distributive}(\mathbb{Z}, +_{\mathbb{Z}}, *_{\mathbb{Z}})\} \Longrightarrow$
$\{\texttt{associative}(\mathbb{Z}, +_{\mathbb{Z}}),$
$\texttt{left-distributive}(\mathbb{Z}, +_{\mathbb{Z}}, *_{\mathbb{Z}}),\ \texttt{right-distributive}(\mathbb{Z}, +_{\mathbb{Z}}, *_{\mathbb{Z}})\}$

$\vdash$ $\{\texttt{associative}(\mathbb{Z}, +_{\mathbb{Z}}),\ \texttt{distributive}(\mathbb{Z}, +_{\mathbb{Z}}, *_{\mathbb{Z}})\} \Longrightarrow$
$\{\texttt{left-distributive}(\mathbb{Z}, +_{\mathbb{Z}}, *_{\mathbb{Z}}),\ \texttt{right-distributive}(\mathbb{Z}, +_{\mathbb{Z}}, *_{\mathbb{Z}})\}$
by **B1**

$\vdash$ $\{\texttt{associative}(\mathbb{Z}, +_{\mathbb{Z}}),\ \texttt{distributive}(\mathbb{Z}, +_{\mathbb{Z}}, *_{\mathbb{Z}})\} \Longrightarrow$
$\{\texttt{distributive}(\mathbb{Z}, +_{\mathbb{Z}}, *_{\mathbb{Z}})\}$
by **B2** with **A** and $\sigma(R) = \mathbb{Z},\ \sigma(+) = +_{\mathbb{Z}},\ \sigma(*) = *_{\mathbb{Z}}$

$\vdash$ $\{\texttt{associative}(\mathbb{Z}, +_{\mathbb{Z}}),\ \texttt{distributive}(\mathbb{Z}, +_{\mathbb{Z}}, *_{\mathbb{Z}})\} \Longrightarrow \emptyset$
by **B1**

which is accepted by **B3**. Note, that the only choice throughout the deduction consists of determining which rule of L should be applied. Also the left-hand side P_1 of the goal does not change throughout the whole deduction, so that keeping track of the changes occurring in P_2 is sufficient. Finally it may be worth mentioning that for rules $l \longrightarrow r$ with a right-hand side r consisting of one property predicate only, the rule **B2** from above can be simplified to

(B2') Replace $\vdash P_1 \Longrightarrow P_2$ by $\vdash P_1 \Longrightarrow (P_2 \backslash \sigma(r)) \cup \sigma(l)$ if there are a rule $l \longrightarrow r \in L$ and a substitution σ with $\sigma(r) \in P_2$.

Thus no intersection has to be computed in this case. Note, that this kind of rule can be easily obtained by splitting up given rules as for example the distributivity rule from above into the following two ones.

$\{\texttt{distributive}(R, +, *)\} \longrightarrow \{\texttt{left-distributive}(R, +, *)\}$
$\{\texttt{distributive}(R, +, *)\} \longrightarrow \{\texttt{right-distributive}(R, +, *)\}$

However, transforming all the rules of L this way would heavily increase the number of steps in a deduction and only detailed experiments will show which method is to prefer.

6.3 Generic Type Checking

As we have seen generic programming introduces new abstractions into programming: Type parameters describe a class of possible instantiations. Thus a type parameter implicitly determines syntactic and (most often implicitly) semantic requirements an instantiation has to meet in order to make the generic algorithm run correctly. In addition, the use of subalgorithms comes with the same problem: in order to work correctly, the (semantic) requirements of the subalgorithm must be implied by the ones of the calling algorithm. However, especially semantic requirements are usually not taken into consideration during instantiation, they can be found in language descriptions only and are not included in the language itself. Thus they are not checked during the compilation process leading to errors hard to find or remove; also because error messages given are not able to give hints referring to semantic requirements. In the following we present a small programming language Saga — *S*ignatures and *A*djectives for *G*eneric *A*lgorithms — which using the properties-based approach of section 6.2 allows for checking symbolically both whether a proposed instantiation indeed fulfills the requirements given by the generic instantiation and whether an algorithm is legal as a subalgorithm of another. Furthermore, the properties-based approach supports the generation of meaningful error messages in case this check fails simply by using the properties' definitions. A prototype checker and compiler that translates into C++ is implemented allowing to experiment with legality checks. Here we focus on a conceptional presentation of the language, details and formal definitions can be found in [GS03].

Saga essentially is an imperative language similar to C/C++ extended with the notion of signatures and adjectives in order to exactly describe the input/output parameters of generic algorithms. Using this description and the calculus of the last section it is then possible to check for correctness of subalgorithms and legal instantiation of algorithms during compile time. The calculus has been implemented in the system presented in [Gas03] that allows to generate type checkers based on on a given set of typing rules and type implications. The language provides the usual statements **while**, **do** and **for** loops; **if** conditionals and compound statements are built using curly

brackets. The basic type system of the language includes **Int**, **Char**, **String** and **Bool** type constructors **Pair**, **Cell**, a reference type **&**, as well as a function type constructor.

At the core of SAGA is the possibility to define adjectives over signatures describing particular properties of various operators. A signature is a sequence of type names and operators with their arities, whereby naming in a signature must be unique. An adjective basically introduces a name for a property. The meaning of the property being introduced can be described informally or by a formula. In particular the precise shape of the formula given depends on the logic being used for verification. Thus checking done during compiling is purely symbolical. However, using the formulae given in the definition of adjectives a mechanized reasoning system can be connected in order to formally prove the correctness of the implications used and properties stated for particular domains.

So for example an adjective `strict_weak_order` is given by the signature consisting of a carrier `T` and two binary operations `<` and $\sim$ together with a description of the intended meaning of the property defined following the keyword `means`. Here, we use standard first-order logic to formulate the properties of a weak strict order. A more STL-like adjective `assignable` for a type `T` is similarly given by the signature consisting of `T` and an assignment operator `=`. Note the use of the reference operator `&` for describing the arity of `=`. The description of the adjective's meaning in this case uses the informal option which is indicated by the keyword `informal`. In both cases the description of the property can be used to generate expanded error messages in order to support the user in case type legality checks during compilation have failed.

```
Adjective strict_weak_order
 for (T, < : (T,T)->Bool, ∼ : (T,T)->Bool)
 means all(x:T) not(x < x) ∧
       all(x,y:T) x < y ==> not(y < x) ∧
       all(x,y,z:T) (x < y ∧ y < z) ==> x < z ∧
       all(x,y:T) (x ∼ y) = (not(x < y) ∧ not(y < x)) ∧
       all(x,y,z:T) (x ∼ y ∧ y ∼ z) ==> x ∼ z

Adjective assignable
 for (T, = : (&T,T)->&T )
 informal "Operator" = " is an assignment on " T;
```

Note that instead of defining the adjective `strict_weak_order` as a whole,

it is also possible to define adjectives for the constituting properties such as `irreflexive` or `antisymmetric` separately and combine these to get the requirements for `strict_weak_order`. Thus the user has the possibility to describe requirements for SAGA algorithms as fine-grained as he considers necessary for his generic algorithms.

The same way adjectives describing properties of algebraic domains such as `Group` or `Euclidean` as well as more involved properties used for the description of containers in the STL such as `bidirectional_iterator` or `random_access_iterator` can be defined. Then adjectives are used to specify requirements of generic algorithms: Together with a signature they are included in algorithm headers and are thus part of the programming language. The signature indicates which operations are to be instantiated and can thus be used in the algorithm body. Based on the calculus of section 6.2 the signature and adjective declarations connected with algorithms, subalgorithms, and instantiations are used to check whether an instantiation or a subalgorithm is legal for a generic algorithm.

As an example for a SAGA algorithm we consider the sorting algorithm `qsort` realizing quicksort. Sorting a sequence of elements can be made generic in two ways: First, the type of the elements to be sorted is irrelevant as long as they can be compared. Second, the way in which such a sequence is stored is irrelevant also, as long as some particular operation on sequences can be done. Both abstractions have been elaborated in the STL where iterators are used to implement this abstraction; we adopt these notions here. Quicksort splits up the sequence to be sorted using a `partition` subalgorithm for which a pivot element is chosen using another subalgorithm `select_pivot`. However, in the following we only consider the subalgortihm `partition`. A partition of a sequence `T` into two subsequences can be built using a bidirectional iterator where the elements stored in the container must be assignable [Aus99]. The elements are partitioned with respect to some boolean predicate on the element set `VT`. The algorithm then returns an iterator indicating the border between the two subsequences. Thus the algorithm header in our language looks as follows.

```
Algorithm partition
  [ (T,VT,...)  with bidirectional_iterator(T,VT,...);
    (VT,...)  with assignable(VT,...) ]
  ( fst : T;
    lst : T;
    pred : (VT)->Bool )
  return T
```

We feel that the use of ... in the signature and adjective declarations needs some explanation. It would be tedious to always repeat the whole signature in all declarations. Therefore operators and their arity can be left out using ... to indicate that something has to be restored; however, domain names have to be given in order to identify domains of different adjectives. The operators of an algorithm's signature are just the union of the ones used in the adjectives, and can be thus constructed from the adjectives' definitions by unification.

Now following [MDS01, Aus99] a generic version `qsort` of quicksort can be formulated in our language: To realize sequences `random_access_iterator`s are employed because the subalgorithm `select_pivot` uses the median to select an element of the container. This can be done efficiently only with a `random_access_iterator`. However, other pivot selection strategies may work well for weaker iterators. In addition, the elements being stored need to `assignable` and `equality_comparable` and must be equipped with a `strict_weak_order`. Thus we get the following algorithm. Note that operations on iterators, such as for example `==`, occur in the definition of the adjective `random_access_iterator` and are not stated again in the algorithm declaration. We also included the algorithm body here to show that SAGA indeed can be considered as a usual declarative programming language extended with possibilities to specify type parameters.

```
Algorithm qsort
  [ (RAI,VT,...)
     with random_access_iterator(RAI,VT,...);
     (VT,...)
     with equality_comparable(VT,...),
          strict_weak_order(VT,...),
          assignable(VT,...) ]
  ( first : RAI;
    last : RAI )
begin
 if (first == last) return;
 var pivot : VT;
 pivot = select_pivot(first,last);
 var split : RAI;
 split = partition(first, last, (lambda x . x < !pivot));
 qsort(first,!split);
 qsort(!split,last)
end
```

Now, how is it checked whether `partition` is a legal subalgorithm for `qsort`? First, the actual arguments and the return type of `partition` are matched with `partition`'s formal parameters. Note that this may include overload resolution in case there is more than one `partition` algorithm. Then, as already mentioned, the adjectives attached to `partition` are instantiated with the types given by `qsort`. Now it has to be checked whether the adjectives of `qsort` imply the ones of `partition`. Following the calculus from section 6.2 this is obvious for `assignable(VT,=)`. The adjective `bidirectional_iterator` is not included in `qsort`'s adjectives. However, this is easily deduced using the following rule; note again the use of `..` as an abbreviation for the adjectives' operators.

```
Rules:
  for (T,VT,...)
     random_access_iterator(T,VT,...)
 ==> bidirectional_iterator(T,VT,...);
```

Thus `partition` is a legal subalgorithm for `qsort`. In particular, this means that if `qsort` is instantiated adjectives of the subalgorithms need not be considered again; it suffices to check for the adjectives directly attached to `qsort`. Note that the recursive call of `qsort` is also checked like other subalgorithms. In case of errors, for example if the above rule is not present, error messages like "cannot deduce `bidirectional_iterator(T,VT,...)` from the given set of adjectives" can be easily generated. So for example if a user has written an algorithm based on `bidirectional_iterator` and it appears that `random_access_iterator` should be deduced, this gives evidence that the user's algorithm is not well-designed, that is other subalgorithms must be provided or even the algorithm probably cannot be formulated using `bidirectional_iterator` only due to the algorithm's requirements.

Checking for the correctness of particular instances is basically the same as checking for the correctness of subalgorithms: The instantiation's properties must imply the generic algorithm's ones. This requires providing the necessary knowledge about the instantiation in order to make it available for the checker. This is done using rules with an empty premise set. For example, the following rules state basic properties of the integers used to deduce them as a correct instance of `qsort`.

```
Rules: for () { } ==> assignable(Int,=);
Rules: for () { } ==> equality_comparable(Int,==,!=);
Rules: for () { } ==> strict_weak_order(Int,<,==);
```

Thus, we check whether instantiations are legal for a generic algorithm with respect to a knowledge base, here about the integers. Note, that if an instantiation is not accepted, the reason might be that the knowledge stored is not sufficient although this particular instance is correct.

Now, the generic algorithm `qsort` can be used to sort for example integer arrays. `Array(T)` is a built-in type providing the usual set of operators. It is parameterized by a type `T` giving the type `VT` of the elements being stored in the array. Using this we can provide the following algorithm.

```
Algorithm main
 return Int
begin
  var a : Array(Int);
  a = new_array(random(30),0);
  // ... initialize the array ...
  qsort(ibegin(!a), iend(!a));
  print_array(!a)
end
```

Note that `qsort` does not need a third argument to specify the order. The adjective `strict_weak_order` requires the existence of a binary operator `<` and thus it is checked whether the instantiation tt Int provides an operator with the properties necessary. In case there is more than one algorithm for a binary operator fulfilling all requirements Saga will not instantiate but collect all possible legal instantiations. It is planned to include a `with` clause with which the user can resolve such ambiguities. This allows for instance the user to call `qsort` with both `<` and `>` for the integers by just stating which instance he desires.

The requirements of `qsort` for `VT` are deduced using the three rules for the integers `Int` from above. `ibegin(!a)` and `iend(!a)` return iterators of type `ArrayIterator(T)` where `T` in our example is instantiated with `Int`. Thus it has to be checked whether `ArrayIterator(Int)` indeed is a `random_access_iterator`. This is deduced with the help of the following rule. Note, that in this case the complete signature has to be provided because the operators denote algorithms connected to `ArrayIterator(T)`.

```
Rules: for (T)
 {} ==> random_access_iterator(ArrayIterator(T),T,=,==,
                      !=,++_,_++,--_,_--,*_^^v,*_^^r,+,-);
```

In this way the instantiation of `qsort` with the integers is accepted. However, as already mentioned, the resulting instance is correct only relatively

to the correctness of the rules used. For example, the last rule postulates that the implementation of `ArrayIterator` fulfills the requirements of a `random_access_iterator`. This may be proven externally with a mechanized reasoning system, but is not considered during type checking. Note again, that requirements of the subalgorithms used need not be taken into account during instantiation.

6.4 Verification of Generic Algorithms

A generic algorithm can be considered as an algorithmic scheme: Parts of the algorithm are left abstract using some kind of type parameter. These abstract parts may be concerned with data handling [MDS01] or even with domains the algorithm is dealing with [SL00a]. To get a running algorithm the abstract parts have to be instantiated with concrete pieces of code. However, as argued in chapter 2, it is usually not guaranteed that after instantiation the resulting non-generic algorithm works correctly: the instantiation may lack necessary operations or the operations do not fulfill all requirements implicitly given by the generic algorithm. Consider for example again a sorting algorithm; a generic version can easily be formulated for an arbitrary domain with a binary relation. However, to actually sort sequences over a particular domain in the usual sense this is not enough: it is necessary that the binary relation is a total order. In other words, if the instantiation for the generic sorting algorithm does not fulfill the requirement that its associated binary relation is a total order, the instance of the generic sorting algorithm will not work correctly.

The other way round these requirements for instantiations in fact describe conditions under which the generic algorithm works as expected, that is assuming these conditions the generic algorithm is correct with respect to its input/output specification independent from the particular instance [Sch00b]. Thus the correctness of a generic algorithm does not depend on the underlying domain, but in some sense more on whether the operations used come with particular properties. Consequently, the correctness of a generic algorithm can be formulated with respect to a set of properties of the algorithm's operations, to say it in other words with respect to a set of minimal requirements: Provided that a particular instantiation fulfills these properties, the resulting (non-generic) algorithm will work correctly. Thus the problem of correct instantiation is nothing else than checking whether a properties based theorem about an abstract algorithm of method is valid in a particular domain implicitly given by the instantiation. In our exam-

ple, properties necessary for the correctness of the generic sorting algorithm are reflexivity, antisymmetry, transitivity and totality of the binary relation. The integers enriched with their usual order provide these properties, hence the integers constitute a legal instantiation for a generic sorting algorithm.

Following [Sch03] we now present a case study on properties based verification of generic algorithms: we use the Mizar system [RT01a] to formalize and prove the correctness of Euclid's algorithm for computing greatest common divisors. It is well-known that Euclid's method works for arbitrary Euclidean domains [BW93]. However, our proofs will show that Euclidean domains provide more properties than necessary to make the method work.

We model the Euclidean algorithm by the sequence `e-seq` of remainders computed.[1] However, at this point it is irrelevant how—or even whether—remainders are computed. Thus we can define `e-seq` as a function that, based on two initial values `a` and `b`, computes the next value by just applying a given function `g` to the two preceding values. The computation proceeds as long as the second argument of `g` does not equal `0.R`.

```
definition
let R be non empty ZeroStr,
    g be Function of [:R,R:],R,
    a,b be Element of R such that a <> 0.R;
func e-seq(a,b,g) -> Function of NAT,R means
  it.0 = a &
  it.1 = b &
  for i being Nat holds
  it.(i+1) = 0.R or it.(i+2) = g.(it.(i),it.(i+1));
end;
```

Next we define two requirements describing the correctness of the Euclidean method. First, the computation should terminate, in other words `e-seq` should eventually yield the value 0. Second, the greatest common divisor of the initial values `a` = `e-seq.0` and `b` = `e-seq.1` should be invariant throughout the computation, that is two consecutive pairs of values in `e-seq` should possess the same greatest common divisor. Note however, that the attributes are defined for arbitrary functions `f` and not for `e-seq` only.

```
definition
let R be non empty ZeroStr,
    f be Function of NAT,R;
```

[1] see [Sch00b] for another approach based on Hoare's calculus.

```
attr f is terminating means
  ex t being Nat st t > 0 & f.t = 0.R;
end;

definition
let R be non empty doubleLoopStr,
    f be Function of NAT,R;
attr f is gcd_computing means
  for c being Element of R, i being Nat holds
  f.(i+1) = 0.R or
  (c is_gcd_of f.i,f.(i+1) implies
                    c is_gcd_of f.(i+1),f.(i+2));
end;
```

Thus proving that `e-seq(a,b,g)` is `terminating` and `gcd_computing` for all initial values `a` and `b` shows the correctness of Euclid's method for arbitrary domains `R` with arbitrary degree function `d` and arbitrary function `g`. However, to do so further properties of the domain `R` (and of the function `g`) are necessary. Essential for termination is the existence of a degree function well-known from Euclidean rings. Here we define degree functions isolated from other ring properties by first introducing a corresponding attribute. We use a somewhat unusual form of the Euclidean property which we called `Left-Euclidean`: `a` is decomposed into `r + q * b`, which differs from `q * b + r` if addition is not commutative. This allows to prove the requirements from above without assuming commutativity of addition.[2] Note however, that in case addition is commutative our definition coincides with the usual one for Euclidean domains.

```
definition
let R be non empty doubleLoopStr;
attr R is Left-Euclidean means
  ex f being Function of the carrier of R,NAT st
  for a,b being Element of R st b <> 0.R holds
  ex q,r being Element of R st
  a = r + q * b & (r = 0.R or f.r < f.b);
end;
```

[2]We generalized the well-known correctness proof found in text books (see for example [BW93]). This proof requires commutative addition if `a=q*b+r` instead of `a=r+q*b` is used. However, there may exist a different proof showing the correctness of Euclid's method without assuming commutativity of addition in this case.

```
definition
let R be Left-Euclidean (non empty doubleLoopStr);
mode DegreeFunction of R
         -> Function of the carrier of R,NAT means
  for a,b being Element of R st b <> 0.R holds
  ex q,r being Element of R st
  a = r + q * b & (r = 0.R or it.r < it.b);
end;
```

Finally, we need to formalize that the function `g` used by `e-seq` computes remainders. This can be easily done for arbitrary degree functions of `R`, hence for arbitrary structures that are `Left-Euclidean` as the following definition shows. Note however, that it is possible to require other properties for `g`. Crucial is that the properties required allow to show that the resulting `e-seq` is `gcd_computing`.

```
definition
let R be Left-Euclidean (non empty doubleLoopStr),
    d be DegreeFunction of R,
    g be Function of [:R,R:],R;
pred g computes_mod_wrt d means
  for a,b being Element of R st b <> 0.R holds
  ex q being Element of R st
           a = g.(a,b) + q * b &
           (g.(a,b) = 0.R or d.(g.(a,b)) < d.b);
end;
```

After these preparations we can prove the following theorems describing the correctness of Euclid's method for computing greatest common divisors. Not surprisingly the property `Left-Euclidean` is sufficient to prove termination of `e-seq` (provided of course that the function used to compute the sequence fulfills the property `computes_mod_wrt`). Theorem `T2` is more interesting: the properties used to prove the theorem show that greatest common divisors can be calculated[3] in domains `R` where neither addition nor multiplication is commutative. Furthermore `R` need not be an integral domain, that is `R` may contain zero divisors. Note also, that the (proofs of the) theorems hold for arbitrary degree functions `d` of `R`.

[3]To actually compute greatest common divisors it is also necessary that `a` is a greatest common divisor of `a` and `0.R`. To prove this, addition of `R` must be cancelable and `1.R` must be a left unity with respect to multiplication.

```
theorem T1:
for R being Left-Euclidean (non empty doubleLoopStr),
    d being DegreeFunction of R,
    g being Function of [:R,R:],R st g computes_mod_wrt d
for a,b being Element of R
holds e-seq(a,b,g) is terminating;

theorem T2:
for R being add-associative associative right-zeroed
            right_complementable left-distributive
            Left-Euclidean (non empty doubleLoopStr),
    d being DegreeFunction of R,
    g being Function of [:R,R:],R st g computes_mod_wrt d
for a,b being Element of R
holds e-seq(a,b,g) is gcd_computing;
```

The theorems show in particular that Euclid's method can compute greatest common divisors in rings where neither addition nor multiplication is commutative. However, as the greatest common divisor is not unique algorithms usually return one particular characterized with respect to an ample set. This is done by employing an ample function to transform the computed greatest common divisor into an element of the given ample set.

Therefore, another function `res-gcd` is introduced which takes the last element of an `e-seq`, that is the one right before the first zero occurs, and returns its corresponding element in the ample set using the function `NF` which coincides with an ample function for `A`. Note that the ample set `A` is an explicit parameter of the function `res-gcd`, and thus multiplication of `R` has to be `associative` and `well-unital` just because these properties are necessary to show the existence of ample sets.

```
definition
let R be associative well-unital (non empty doubleLoopStr),
    A be AmpleSet of R,
    f be Function of [:R,R:],R,
    a,b be Element of R;
func res-gcd(a,b,f,A) -> Element of A means
  ex i being Nat st
    e-seq(a,b,f).(i+1) = 0.R &
    it = NF(e-seq(a,b,f).i,A) &
    for k being Nat st k <= i holds e-seq(a,b,f).k <> 0.R;
end;
```

Then it can be easily shown that the result of `res-gcd` is both an element of the given ample set `A`, which holds basically by definition, and a greatest common divisor of the start values `a` and `b` of `e-seq`, which holds because greatest common divisors are invariant with respect to the associated relation. Thus `res-gcd` computes the greatest common divisor of `a` and `b` following Euclid's method:

```
theorem T3:
for R being add-associative right_zeroed right_complementable
            associative well-unital distributive domRing-like
            Left-Euclidean (non empty doubleLoopStr),
    A being AmpleSet of R,
    d being DegreeFunction of R,
    g being Function of [:R,R:],R st g computes_mod_wrt d
for a,b being Element of R st a <> 0.R
holds res-gcd(a,b,g,A) is_gcd_of a,b &
      res-gcd(a,b,g,A) in A;
```

Note again the reduced requirements on the underlying domain `R`. Now, similar to section 4.4 and 6.2 we can use theorem `T1`, `T2`, and `T3` to infer (generic and non-generic) domains with which the instantiation of the generic Euclidean algorithm is legal. Assuming, for example, that the type `EuclideanRing` is available in Mizar as a `doubleLoopStr` and that the appropriate attributes have been clustered, we get the following theorems.

```
theorem
for R being EuclideanRing,
    d being DegreeFunction of R,
    g being Function of [:R,R:],R st g computes_mod_wrt d
for a,b being Element of R
holds e-seq(a,b,g) is gcd_computing &
      e-seq(a,b,g) is terminating by T1,T2;

theorem
for R being EuclideanRing (non empty doubleLoopStr),
    A being AmpleSet of R,
    d being DegreeFunction of R,
    g being Function of [:R,R:],R st g computes_mod_wrt d
for a,b being Element of R st a <> 0.R
holds res-gcd(a,b,g,A) is_gcd_of a,b &
      res-gcd(a,b,g,A) in A by T3;
```

Please note again that these theorems are accepted by just referencing theorems `T1` and `T2` resp. `T3`, because the attributes of an `EuclideanRing` are a superset of the ones used in the theorems.

Consider the integers as a second example. The ring of integers in particular establishes a Euclidean domain. This is again formalized in a conditional cluster definition where it is shown that `INT.Ring` fulfills the attributes not proven so far.[4] After the cluster is registered the theorem that Euclid's algorithm instantiated with the integers is correct can be proven by just referencing `T1` and `T2`.

```
definition
cluster INT.Ring -> Euclidean;
end;

theorem
for A being AmpleSet of INT.Ring,
    d being DegreeFunction of INT.Ring,
    g being Function of [:INT.Ring,INT.Ring:],INT.Ring
      st g computes_mod_wrt d
for a,b being Element of INT.Ring st a <> 0.(INT.Ring)
holds res-gcd(a,b,g,A) is_gcd_of a,b &
      res-gcd(a,b,g,A) in A by T3;
```

The same way further parameters occurring in the theorem can be specialized easily. For instance, the usual absolute value function `absint` is a degree function of the ring of integers. Now, if `absint` is defined as a `DegreeFunction of INT.Ring`—which in Mizar of course includes proving that `absint` indeed is a degree function—the following is accepted by the Mizar checker.

```
theorem
for A being AmpleSet of INT.Ring,
    g being Function of [:INT.Ring,INT.Ring:],INT.Ring
      st g computes_mod_wrt absint
for a,b being Element of INT.Ring st a <> 0.(INT.Ring)
holds res-gcd(a,b,g,A) is_gcd_of a,b &
      res-gcd(a,b,g,A) in A by T3;
```

[4]Note that in the cluster the usual property `Euclidean` and not `Left-Euclidean` is proven. This is sufficient due to a conditional cluster definition stating that for commutative addition `Euclidean` implies `Left-Euclidean`. The same holds for the attributes `distributive` and `left-distributive`.

The Euclidean algorithm also works for polynomials over one variable if the coefficients form a field, which is our final example. In Mizar polynomial rings have been introduced for an arbitrary number of variables [RT01b, MS02a], that is we have a type `Polynomial-Ring(n,R)` where `n` is an ordinal giving the number of variables and `R` is the coefficient domain. Now if we prove in a cluster definition the necessary attributes[5] for `Polynom-Ring(1,K)` where `K` is a field, we can infer the correctness of the generic Euclidean algorithm for polynomial rings with one variable over fields without revising or giving a new proof.

```
definition
let K be non trivial Field;
cluster Polynom-Ring(1,K) -> Euclidean;
end;

theorem
for K being non trivial Field,
    A being AmpleSet of Polynom-Ring(1,K),
    d being DegreeFunction of Polynom-Ring(1,K),
    g being Function of
      [:Polynom-Ring(1,K),Polynom-Ring(1,K):],
                                    Polynom-Ring(1,K)
      st g computes_mod_wrt d
for a,b being Element of Polynom-Ring(1,K)
        st b <> 0.(Polynom-Ring(1,K))
holds res-gcd(a,b,g,A) is_gcd_of a,b &
      res-gcd(a,b,g,A) in A by T3;
```

To summarize, a properties based approach allows to state the correctness of generic algorithms at a very abstract level, namely by formalizing requirements so that the algorithm works. Once the correctness or other properties of a generic algorithm have been shown with respect to these requirements, it is then possible to identify correct instantiations by just comparing properties of theorems and domains.

In addition using properties based theorems can result not only in generalization of domains but also of methods. For instance, the properties of the domain `R` used to show the correctness of Euclid's algorithm in the cases study were in fact necessary only to prove that $\gcd(a,b) = \gcd(b,\mathsf{g}(a,b))$

[5] A number of clusters for polynomial rings with weaker domains than fields as coefficients has already been defined in [RT01b]. These clusters are of course also applicable in the case of fields.

where g is the function yielding the remainder of a and b. The rest of the proofs constructed is not affected by properties of R—exept for the Euclidean property of course. Consequently, greatest common divisors can be computed in a (left-Euclidean) domain R, if there is a function g with $\gcd(a, b) = \gcd(b, \mathtt{g}(a, b))$ and a well-founded relation $<$ with $\mathtt{g}(a, b) < b$ for all $a, b \in$ R. Properties of R allowing to prove this property of g are then sufficient to compute greatest common divisors in R— independent of the function g realizes.

6.5 Design of Libraries

The properties-based approach used in the previous sections to check for legal instantiation in generic algorithms can also be used for the development of libraries. Here we focus on libraries of mathematical theorems, though in principle other libraries can be similarly designed. By using properties to describe requirements under which a theorem holds, the knowledge is presented in a more general way leading to improved support for reusing theorems. Furthermore, because not only theorems but also individual domains can be represented this way, the calculus of section 6.2 can be used to deduce whether a theorem holds in a given domain. Thus a properties-based library gets a more active role by processing the knowledge and, furthermore, supports the user by extracting theorems of a particular theory.

The key idea to design mathematical libraries this way is to separate the content of a theorem from the properties necessary to prove a theorem correct [Sch01b]. The main observation is that the correctness of theorems essentially does not depend on domains but rather on sets of properties. This can be seen in corresponding proofs where not all properties of a domain are actually used. Consequently a theorem falls into several constituents: The content $Cont(T)$ of a theorem T states the proposition the theorem is about. The way propositions are represented is of minor concern here, one may think of some kind of logical formulae allowing to delegate proofs to some external mechanized reasoning system. However, the domain and the operations necessary to express $Cont(T)$ are given separately in a signature $Sig(T)$. This allows to distinguish between the proposition of the theorem and conditions under which it holds. This is further elaborated in the third component of a theorem T. Here, a set of properties $Props(T)$ is given. The intended meaning is that using these properties $Cont(T)$ can be proven correct. Properties again are represented by predicate symbols. Thereby, the

arity of these symbols corresponds to the carriers and operations necessary to formulate the property. Summarized we consider a theorem T as a triple

$$T \;=\; (Sig(T), Cont(T), Props(T))$$

where the statements of $Cont(T)$ and $Props(T)$ fit to the given signature $Sig(T)$, that is use only symbols introduced there. The other way round $Sig(T)$ should not include more than necessary for the statements given in $Cont(T)$ and $Props(T)$. Note, that we do not use a formal definition of properties in the sense of first-order logic here. We assume that the meaning of a property is indicated by its name, that is by the chosen predicate symbol, the formal definition given elsewhere and not directly processed by the library. Consider, for example, the following well-known theorem T from ideal theory.

Let R be a (commutative) ring. Then $\{0\}$ is an ideal in R.

We now transform the theorem into our representation thereby generalizing it to due to the properties necessary to prove it. We start by just mentioning that we get for the content of T

$$Cont(T) \;=\; \{0\} \textit{ is an ideal in } R$$

and that it is a straightforward task to expand the description of the content into, for instance, a first-order formula. More importantly, the signature necessary to formulate this proposition is

$$Sig(T) \;=\; (R, +, *, 0),$$

that is the symbol 1 usually part of the ring signature is not included. Furthermore, in order to prove that $\{0\}$ is an ideal in R it is only necessary that $+$ is associative, provides a right zero as well as right inverses and that $+$ and $*$ fulfill the distributivity law [BRS01]. So we get

$$\begin{aligned} Props(T) \;=\; \{&\texttt{associative}(R, +),\ \texttt{right-zero}(R, +, 0),\\ &\texttt{right-inverse}(R, +, 0),\ \texttt{distributive}(R, +, *)\}, \end{aligned}$$

that is the properties connected with T are much weaker than the properties of a ring required in the original version of the theorem. Note that the arguments $R, +, *$ and 0 can be interpreted as variable symbols since they represent arbitrary carriers and operations respectively.

Domains D can be represented in a similar way. They also consist of a signature $Sig(D)$ giving carriers and operations of the domain and a set $Prop(D)$ containing properties the domain fulfills, thus

$$D \;=\; (Sig(D), Prop(D)).$$

This of course works for both abstract domains such as rings or fields and individual domains. For example, the description of the ring of integers $\mathbb{Z}$ would include the following.

$$\begin{array}{rcl} Sig(\mathbb{Z}) & \supseteq & (\mathbb{Z}, +_{\mathbb{Z}}, *_{\mathbb{Z}}, 0_{\mathbb{Z}}, 1_{\mathbb{Z}}) \\ Props(\mathbb{Z}) & \supseteq & \{\texttt{associative}(\mathbb{Z}, +_{\mathbb{Z}}), \texttt{distributive}(\mathbb{Z}, +_{\mathbb{Z}}, *_{\mathbb{Z}}), \\ & & \texttt{commutative}(\mathbb{Z}, +_{\mathbb{Z}}), \texttt{commutative}(\mathbb{Z}, *_{\mathbb{Z}}), \\ & & \texttt{Euclidean}(\mathbb{Z}, +_{\mathbb{Z}}, *_{\mathbb{Z}}, 0_{\mathbb{Z}})\} \end{array}$$

where, in contrast to the description of theorems and abstract domains, $\mathbb{Z}, +_{\mathbb{Z}}, *_{\mathbb{Z}}, 0_{\mathbb{Z}}$ and $1_{\mathbb{Z}}$ are now constant symbols. Thus the approach allows for the description of both domains and theorems with regard to properties of domains and the properties making theorems correct. Similar to the correctness of instantiations of generic algorithms, this gives rise to a straightforward criterion whether a theorem T holds in a domain D where D may be both an abstract or a concrete domain. It has only to be checked whether the domain D provides both the necessary signature and the properties connected with the theorem T, thus

$$\begin{array}{l} Cont(T) \text{ is valid in } D \;:\Longleftrightarrow \\ \qquad\qquad Sig_D(T) \subseteq Sig(D) \wedge Props_D(T) \subseteq Prop(D) \end{array}$$

The notations $Sig_D(T)$ and $Props_D(T)$ resp. mean that the variable symbols occurring in the theorem T, actually in $Sig(T)$, are replaced by the corresponding symbols of the domain D. This in fact is a match of the symbols given in the domain's definition with those of the theorem. Thus, for example, it can be easily checked that the theorem from above is valid for both abstract rings and the ring of integers, just because the signature and the properties attached to the content of the theorem are included in the properties of an abstract ring and the ring of integers, respectively. This nicely corresponds to checking of legal instantiation for generic algorithms presented in section 6.3.

Note however, that the failure of this check does not necessarily imply that a theorem does not hold in a domain; the check is relative to the properties stated about the theorem and the domain. Theorem proving in its original

sense is not necessary here, just checking whether certain properties, that is predicates connected with domains and theorems, are present. Thus, the validity of a theorem in a particular domain is reduced to just comparing the theorem's and the domain's properties. Nevertheless the deduced results are correct provided that the meaning of the properties and rules was defined properly, which actually can be ensured by proving the rules externally with some mechanized reasoning system. Note, that this method also allows for straightforward error messages by collecting the properties of the theorem T not included in the ones of the domain D.

Again, the properties-based approach can be generalized to implications of sets of properties based on the calculus of section 6.2. This results in an active library that uses deduction to check validity of stored general theorems in both abstract and individual domains. Thus theorems can be kept in the library in a more general fashion . Nevertheless their usual connection to well-known domains found in textbooks is still maintained by means of deduction.

In fact well-known domains such as groups, fields or vector spaces, can be easily incorporated into the library's predicates to shorten the description of theorems, in this way serving as abbreviations for sets of predicates. On the one hand, they are of course domains having a representation by their signatures and sets of predicates. Thus we have for example

$$\begin{array}{rcl} \mathit{Sig}(\texttt{Group}) & \supseteq & (R, +, 0) \\ \mathit{Props}(\texttt{Group}) & \supseteq & \{\texttt{associative}(R, +), \texttt{right-zero}(R, +, 0), \\ & & \ \texttt{right-inverse}(R, +, 0)\} \end{array}$$

There may be more signature symbols or predicates for `Group`, for example $^{-1} : R \longrightarrow R$ which together with a predicate $\texttt{inverse}(R, +, 0, ^{-1})$ gives the inverse function. Thus this representation of domains can be easily extended by just adding more knowledge to the signature and predicate set. Note that general theorems stored in the library holding for the original group specification, that is theorems requiring at most the predicates $\texttt{associative}(R, +)$, $\texttt{right-zero}(R, +, 0)$ and $\texttt{right-inverse}(R, +, 0)$, still hold after the extension of `Group` just because the calculus is monotone with respect to sets of predicates. Thus a properties-based library supports an incremental approach to specifying domains.

On the other hand, though the basis for the flexibility of the approach, it seems somewhat tedious to have to repeat each individual property for each theorem, that is all the three properties of the `Group` example have to be stated although it is clear that together these properties constitute a group. Thus abbreviation rules, that is in fact the definition of a new predicate as an

abbreviation for a set of predicates, can be given by the user. For example, the following rule

$$\{\texttt{Group}(R,+,0)\} \longrightarrow \{\texttt{associative}(R,+),\ \texttt{right-zero}(R,+,0),\ \texttt{right-inverse}(R,+,0)\}$$

allows to state theorems about groups without giving the three constituting properties. However such theorems may hold in more general domains also which is then not taken into consideration. Thus the user can select the level of abstraction he considers best for his problem. To this end he can also filter out theorems holding in special theories easily by just collecting general theorems the properties of which are implied by the one of the considered theory. It is also possible to consider different views of a particular theory. So, for instance, the following rule

$$\{\texttt{Group}(R,+,0,^{-1})\} \longrightarrow \{\texttt{associative}(R,+),\ \texttt{right-zero}(R,+,0),\ \texttt{right-inverse}(R,+,0),\ \texttt{inverse}(R,+,0,^{-1})\}$$

introduces the theory of groups where the inverse function is explicitly denoted in the signature. Though this is in principle the same theory as the original group theory, it allows the user to present statements about this function explicitly which is not possible with the original group specification.

Summarized the properties-based approach allows to design flexible more active libraries in two ways. First, theorems can be formulated in a very general setting by just stating properties necessary to prove them. Nevertheless, deduction on sets of properties gives the possibility to validate such general theorems for domains in the usual sense. Second, the user is enabled to develop and work in his own individual theory by introducing new domains or rules. General theorems included in the library that hold in his special theory are available automatically due to the deduction mechanisms incorporated in the library.

Chapter 7

Conclusion

In this thesis we have made a tour through the field of generic programming and formal methods and tools supporting it. After a general introduction to generic programming we have presented a number of existing libraries relying on principles and method of generic programming either way. The main outcome of such libraries is that users have great flexibility in applying algorithms contained. They can plug in data structures, domains or even method and policies in this way adapting software to their personal needs. We believe that this way of reusing and adapting software will become even more important in the future. In particular well-designed interfaces and methods supporting users in doing so are of great interest.

Then we focused on semantic requirements for generic algorithms. Semantic requirements appear as a consequence of the additional abstraction generic algorithms introduce: Instances of generic algorithms behave as expected only if instantiations come with a number of (semantic) properties. In addition algorithmic requirements such as the complexity of algorithms are of interest. The specification of such requirements is thus of major concern in order to provide reliable generic algorithms. We argued that it is of particular interest to provide mechanisms allowing to represent requirements in a flexible combinable way. This enables both developers and users to specify generic algorithms and their requirements as general as possible. On the other hand users can strengthen requirements to work in domains they are familiar with. These considerations have resulted in a properties-based approach for describing requirements.

Closely connected with specification is the formal verification of generic algorithms. We argued that both the verification of generic algorithms in its original sense as well as that particular instantiations will result in correct instances of generic algorithms should be supported mechanically. We

presented some mechanized reasoning systems we consider suited for the particular needs of proving generic algorithms correct. We believe that existing mechanized reasoning systems are capable of formalizing mathematical knowledge and proving theorems occurring when specifying and verifying generic algorithms. Nevertheless further improvement, especially of mathematical databases holding and accessing knowledge for this purpose, seems preferable. Imps and Theorema are steps into this direction, especially because they aim at combining reasoning and computing, that is non-algorithmic and algorithmic mathematical knowledge.

Both specification and verification of generic algorithms has been addressed in the applications: A properties-based approach has been presented that allows to describe semantic requirements of generic algorithms adequately. A small programming language—Saga—has been described that based on this approach allows checking for legal instantiations of generic algorithms and subalgorithms. The appropriateness of the properties-based approach for the verification of generic algorithms has been demonstrated by a case study on Euclid's algorithm. In addition, the design of libraries dealing with requirements and in this way supporting storing and reusing more general knowledge has been outlined.

Generic programming offers methodologies and techniques that allow to develop highly adaptable, reusable software and software libraries. On the other hand generic software and algorithms demand a much more abstract view on programming in order to achieve its goals. As a consequence generic software also comes with new demands for both developers and users. These can and should be supported by a rigorous use of formal methods for specification and verification of generic algorithms.

Bibliography

[AC96] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer Verlag, 1996.

[ACC93] M. Abadi, L. Cardelli, and P. Curien. Formal Parametric Polymorphism. *Theoretical Computer Science*, 121(1-2):9–58, 1993.

[AG97] K. Arnold and J. Gosling. *The Java Programming Language*. Addison-Wesley, 2nd edition, 1997.

[AKK99] E. Astesiano, H.-J. Kreowski, and B. Krieg-Brückner, editors. *Algebraic Foundations of Systems Specification*. Springer Verlag, 1999.

[Ale00] A. Alexandrescu. Traits — The Else-If-Then of Types. *C++ Report*, 12(4), 2000.

[Ale01] A. Alexandrescu. *Modern C++ Design*. Addison-Wesley, 2001.

[And86] P. Andrews. *An Introduction to Mathematical Logic and Type Theory — To Truth through Proof*. Academic Press, 1986.

[Aus99] M. Austern. *Generic Programming and the STL — Using and Extending the C++ Standard Template Library*. Addison-Wesley, 1999.

[Bar95] J. Barnes. *Programming in Ada 95*. Addison-Wesley, 1995.

[BCL83] B. Buchberger, G. Collins, and R. Loos, editors. *Computer Algebra — Symbolic and Algebraic Computation*. Springer Verlag, 2nd edition, 1983.

[BdMH96] R. Bird, O. de Moor, and P. Hoogendijk. Generic Functional Programming with Types and Relations. *Journal of Functional Programming*, 6(1):1–28, 1996.

[BG77] R. Burstall and J. Goguen. Putting Theories Together to Make Specifications. In *Proceedings of the 5th International Joint Conference on Artificial Intelligence*, pages 1045–1058, 1977.

[BG80] R. Burstall and J. Goguen. The Semantics of Clear, a Specification Language. In D. Björner, editor, *Proceedings of the Copenhagen Winter School on Abstract Software Specification*, volume 86 of *Lecture Notes in Computer Science (LNCS)*, pages 292–332. Springer Verlag, 1980.

[BJJM99] R. Backhouse, P. Jansson, J. Jeuring, and L. Meertens. Generic Programming — An Introduction. In S. Swierstra, P. Henriques, and J. Oliveira, editors, *3rd International Summer School on Advanced Functional Programming*, volume 1608 of *Lecture Notes in Computer Science (LNCS)*, pages 28–115. Springer Verlag, 1999.

[BJK+97] B. Buchberger, T. Jebelean, F. Kriftner, M. Marin, E. Tomuta, and D. Vasaru. A Survey on the Theorema Project. In W. Küchlin, editor, *Procceedings of the International Symposium on Symbolic and Algebraic Computation (ISSAC'97)*, pages 384–391. ACM Press, 1997.

[BN94] J. Barton and L. Nackman. *Scientific and Engineering C++ — An Introduction with Advanced Techniques and Examples*. Addison-Wesley, 1994.

[Boo02] Boost. *Boost C++ Libraries*. 2002. available at www.boost.org.

[BOSW98] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the Future Safe for the Past — Adding Genericity to the Java Programming Language. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '98)*, 1998.

[BPSM00] T. Bray, J. Paoli, C. Sperberg-McQueen, and E. Maler, editors. *Extensible Markup Language (XML) 1.0, W3C Recommendation*. 2nd edition, 2000. available by anonymous ftp from http://www.w3.org/TR.

[BRS01] J. Backer, P. Rudnicki, and C. Schwarzweller. Ring Ideals. *Formalized Mathematics*, 9(3):565–582, 2001. available by anonymous ftp from http://mizar.uwb.edu.pl/JFM/Vol12/ideal_1.html.

[BST99] M. Bidoit, D. Sannella, and A. Tarlecki. Architectural Specifications in Casl. In A. Haeberer, editor, *Algebraic Methodology and Software Technology, Proceedings of the 7th International Conference (AMAST '98)*, volume 1548 of *Lecture Notes in Computer Science*. Springer Verlag, 1999.

[Buc96] B. Buchberger. Symbolic Computation — Computer Algebra and Logic. In F. Bader and K. Schulz, editors, *1st International Workshop on Frontiers of Combining Systems (FROCOS96)*, volume 3 of *Applied Logic Series*, pages 193–220. Kluwer Academic Publisher, 1996.

[Buc01a] B. Buchberger. The PCS Prover in Theorema. In R. Moreno-Diaz, B. Buchberger, and J. Freire, editors, *Proceedings of the 8th International Conference on Computer Aided Systems — Formal Methods and Tools for Computer Science (EUROCAST 2001)*, volume 2178 of *Lecture Notes in Computer Science (LNCS)*, pages 469–478. Springer Verlag, 2001.

[Buc01b] B. Buchberger. *Theorema — Extending Mathematica by Automated Proving*. Invited talk at the conference on "The Programming System Mathematica in Science, Technology and Teaching", Zagreb, 2001.

[BW93] T. Becker and V. Weispfenning. *Gröbner bases: A Computational Approach to Commutative Algebra*. Springer Verlag, 1993.

[C++98] *JTC1/SC22 — Programming Languages, their Environments and System Software Interfaces. Programming Languages — C++*. International Organization for Standardization, ISO/IEC 14882, 1998.

[Car87] L. Cardelli. Basic Polymorphic Typechecking. *Science of Computer Programming*, 8(2):147–142, 1987.

[Car97] L. Cardelli. Type Systems. In *Handbook of Computer Science and Engineering*, pages 2208–2236. CRC Press, 1997.

[CCH+89] P. Canning, W. Cook, W. Hill, W. Olthoff, and J. Mitchell. F-Bounded Polymorphism for Object-Oriented Programming Languages. In *Functional Programming Languages and Computer Architecture*, pages 273–280. ACM, 1989.

[CE00] K. Czarnecki and U. Eisenecker. *Generative Programming.* Addison-Wesley, 2000.

[Cen94] M. Cengarle. *Formal Specifications with Higer-Order Parameterization.* Ph.D. Thesis, Ludwig-Maximilians-Universität München, Germany, 1994.

[Cen95] M. Cengarle. Semantic Typing for Parameterized Algebraic Specifications. In V. Alagar and M. Nivat, editors, *Algebraic Methodology and Software Technology (AMAST'95)*, volume 936 of *Lecture Notes in Computer Science (LNCS)*, pages 261–276. Springer Verlag, 1995.

[CL90] G. Collins and R. Loos. *Specification and Index of SAC-2 Algorithms.* Technical Report WSI-90-04, University of Tübingen, Germany, 1990.

[CM87] W. Clocksin and C. Mellish. *Programming in Prolog.* Springer Verlag, 1987.

[CoF98] The CoFI Task Group on Language Design. *Casl: The Common Algebraic Specification Language, Summary (Version 1.0)*, available by anonymous ftp from http://www.brics.dk/Projects/CoFI/Documents/CASL/Summary, 1998.

[Col74] G. Collins. *Algebraic Algorithms.* Lecture Manuscript, 1974.

[Col85] A. Colmerauer. Prolog in 10 Figures. *Communication of the ACM*, 28(12):1296–1310, 1985.

[CW85] L. Cardelli and P. Wegner. On Understanding Types, Data Abstraction and Polymorphism. *Computing Surveys*, 17(4):471–523, 1985.

[CW95] M. Cengarle and M. Wirsing. A Calculus of Higher-Order Parameterization for Algebraic Specification. *Bulletin of the Interest Group in Pure and Applied Logics (IGPL)*, 3(4):615–641, 1995.

[CW96] E. Clarke and J. Wing. Formal Methods: State of the Art and Future Directions. *ACM Computing Surveys*, 28(4):636–643, 1996.

[Dav81] M. Davies. Obvious Logical Inferences. In *Proceedings of the 7th International Joint Conference on Artificial Intelligence*, pages 530–531, 1981.

[DCG+89] P. Denning, D. Corner, D. Gries, M. Mulder, A. Tucker, A. Turner, and P. Young. Computing as a Discipline. *Communications of the ACM*, 32(1):9–23, 1989.

[EM85] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specifications 1: Equations and Initial Semantics*, volume 6 of *EATCS Monographs on Theoretical Computer Science*. Springer Verlag, 1985.

[EM90] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specifications 2: Module Specification and Constraints*, volume 21 of *EATCS Monographs on Theoretical Computer Science*. Springer Verlag, 1990.

[End72] H. Enderton. *A Mathematical Introduction to Logic*. Academic Press, 1972.

[Far01] W. Farmer. STMM: A Set Theory for Mechanized Mathematics. *Journal of Automated Reasoning*, 26:269–289, 2001.

[FGK+96] A. Fabri, G. Giezeman, L. Kettner, S. Schirra, and S. Schönherr. The CGAL Kernel — A Basis for Geometric Computing. In *Proceedings of the Workshop on Applied Computational Geometry, Philadelphia, Pennsylvania*, 1996.

[FGK+00] A. Fabri, G. Giezeman, L. Kettner, S. Schirra, and S. Schönherr. On the Design of CGAL, the Computational Geometry Algeorithms Library. *Software — Practice and Experience*, 30:1167–1202, 2000.

[FGT92] W. Farmer, J. Guttman, and F. Thayer. Little Theories. In D. Kapur, editor, *Automated Deduction — CADE-11*, volume 607 of *Lecture Notes in Computer Science (LNCS)*, pages 567–581. Springer Verlag, 1992.

[FGT93] W. Farmer, J. Guttman, and F. Thayer. IMPS — An Interactive Mathematical Proof System. *Journal of Automated Reasoning*, 11:213–248, 1993.

[FGT95] W. Farmer, J. Guttman, and F. Thayer. Contexts in Mathematical Reasoning and Computation. *Journal of Symbolic Computation*, 19:201–216, 1995.

[FvM03] W. Farmer and M. v. Mohrenschildt. A Formal Framework for Managing Mathematics. *Annals of Mathematics and Artificial Intelligence*, 2003. to appear.

[Gas01] H. Gast. *Generic Programming with VIEWS: Type- and Class-Inference with Polymorphic Subsumption by Resolution Theorem Proving.* Technical Report WSI-2001-17, University of Tübingen, Germany, 2001.

[Gas03] H. Gast. *A Type Check Generator.* Ph.D. Thesis, University of Tübingen, Germany, 2003. in preparation.

[GB92] J. Goguen and R. Burstall. Institutions: Abstract Model Theory for Specification and Programming. *Journal of the Association for Computing Machinery*, 39(1):94–146, 1992.

[GH93] J. Guttag and J. Horning. *Larch: Languages and Tools for Formal Specification.* Springer Verlag, 1993.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns — Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[GHK+80] G. Gierz, K. Hofmann, K. Keimel, M. Lawson, J. Mislove, and D. Scott. *A Compendium of Continuous Lattices.* Springer Verlag, 1980.

[GKW02] J. Grabmeier, E. Kaltofen, and V. Weispfenning, editors. *Computer Algebra Handbook — Foundations, Applications, Systems.* Springer Verlag, 2002.

[GM92] J. Goguen and J. Meseguer. Order-Sorted Agebra I: Equational Deduction for Multiple Inheritance, Overloading, Exceptions and Partial Operations. *Theoretical Computer Science*, 105(2):217–273, 1992.

[GM93] M. Gordon and T. Melheam. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic.* Camebridge University Press, 1993.

[GM97] J. Goguen and G. Malcom. *Algebraic Semantics of Imperative Programs*. MIT Press, 1997.

[GMW79] M. Gordon, R. Milner, and C. Wadsworth. Edinburgh LCF — A Mechanized Logic of Computation. volume 78 of *Lecture Notes in Computer Science*. Springer Verlag, 1979.

[Gog96] J. Goguen. Parameterized Progamming and Software Architectures. In *Proceedings of the 4th International Conference on Software Reuse*, IEEE Computer Society, pages 2–11, 1996.

[Gor89] M. Gordon. HOL — A Proof Generating System for Higher-Order Logic. In G. Birtwistle and P. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automated Theorem Proving*, pages 73–128. Springer Verlag, 1989.

[GR83] A. Goldberg and D. Robson. *Smalltalk-80 — The Language and its Implementation*. Addison-Wesely, 1983.

[GS03] H. Gast and C. Schwarzweller. *Local Checks for Semantic Requirements of Generic Algorithms*. in preparation, 2003.

[GTWW77] J. Goguen, J. Thatcher, E. Wagner, and J. Wright. Initial Algebra Semantics and Continuous Algebras. *Journal of the Association for Computing Machinery*, 24(1):68–95, 1977.

[Gut91] J. Guttman. *A Proposed Interface Logic for Verfication Environments*. Technical Report M91-19, The Mitre Corporation, 1991.

[GW98] B. Ganter and R. Wille. *Formal Concept Analysis — Mathematical Foundations*. Springer Verlag, 1998.

[GWM+92] J. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J.-P. Jouannaud. *Introducing Obj3*. Technical Report SRI-CSL-92-03, SRI International, 1992.

[Har90] J. Harrison. *Proof Style*. Technical Report 410, University of Camebridge, U.K., 1990.

[Hin97] J. Hindley. *Basic Simple Type Theory*. Camebridge University Press, 1997.

[Hin00] R. Hinze. *Generic Programs and Proofs*. Habilitation Thesis, University of Bonn, Germany, 2000.

[HKP02] G. Huet, G. Kahn, and C. Pauline-Mohring. *The Coq Proof Assistant — A Tutorial.* available by anonymous ftp from http://coq.inria.fr/doc, 2002.

[Hoa69] C. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12:576–583, 1969.

[Jär02] J. Järvi. The Boost Lambda Library. available by anonymous ftp from http://www.boost.org, 2002.

[Jaś34] S. Jaśkowski. On the Rules of Supposition in Formal Logic. *Studia Logica*, 1, 1934.

[JJ96] P. Jansson and J. Jeuring. Polytypic Programming. In J. Launchbury, E. Meijer, and T. Sheard, editors, *Advanced Functional Programming*, volume 1129 of *Lecture Notes in Computer Science (LNCS)*, pages 68–115. Springer Verlag, 1996.

[JJ97] P. Jansson and J. Jeuring. PolyP — A Polytypic Programming Language Extension. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 470–482, 1997.

[JLM00] M. Jazayeri, R. Loos, and D. Musser, editors. *Generic Programming*, volume 1766 of *Lecture Notes in Computer Science (LNCS)*. Springer Verlag, 2000.

[JP00] J. Järvi and G. Powell. *The Lambda Library — Lambda Abstraction in C++.* Technical Report 378, Turku Centre for Computer Science, 2000.

[JS92] R. Jenks and R. Sutor. *Axiom: The Scientific Computation System.* Springer Verlag, 1992.

[Kam96] F. Kammüller. *Comparison of Imps, PVS and Larch with Respect to Theory Treatment and Modularization.* Technical report, Computer Laboratory, University of Camebridge, 1996.

[Kla83] H. Klaeren. *Algebraische Spezifikation: Eine Einführung.* Springer Verlag, 1983. in German.

[KM92] D. Kapur and D. Musser. *Tecton: A Framework for Specifying and Verifying Generic System Components.* Technical Report 92-20, Rensselaer Polytechnic Institute, 1992.

[KMN94] D. Kapur, D. Musser, and X. Nie. An Overview of the Tecton Proof System. *Theoretical Computer Science*, 133:307–339, 1994.

[Knu94] D. Knuth. *Stanford GraphBase — A Platform for Combinatorial Computing*. ACM Press, 1994.

[Knu97] D. Knuth. *The Art of Computer Programming, Vol. 1: Fundamental Algrithms*. Addison-Wesley, 3rd edition, 1997.

[Knu98] D. Knuth. *The Art of Computer Programming, Vol. 2: Seminumerical Algrithms*. Addison-Wesley, 3rd edition, 1998.

[Kow74] R. Kowalski. Logic as a Computer Language. In J. Rosenfeld, editor, *Information Processing 74 — Proceeding of theIFIP Congress*. North-Holland Publishing, 1974.

[Kre02] A. Kreppel. *Algorithm Selection based on Empirical Data*. Ph.D. Thesis, University of Tübingen, Germany, 2002.

[KS98] S. Kahrs and D. Sannella. Reflections on the Design of a Specification Language. In *Proceedings of the International Colloquium on Fundamental Approaches to Software Engineering. European Joint Conference on Theory and Practice of Software (ETAPS'98)*, volume 1382 of *Lecture Notes on Computer Science*, pages 154–170. Springer Verlag, 1998.

[KST94] S. Kahrs, D. Sannella, and A. Tarlecki. *The Definition of Extended ML*. Technical Report ECS-LFCS-94-300, University of Edinburgh, 1994.

[LC92] R. Loos and G. Collins. *Revised Report on the Algorithm Description Language ALDES*. Technical Report WSI-92-14, University of Tübingen, Germany, 1992.

[Lip98] S. Lippman, editor. *C++ Gems*. Camebridge University Press, 1998.

[Lis92] B. Liskov. *A History of CLU*. Technical Report TR-561, Massachusetts Institute of Technology, Camebridge, 1992.

[LMSS99] R. Loos, D. Musser, S. Schupp, and C. Schwarzweller. *The Tecton Concept Library*. Technical Report WSI-99-02, University of Tübingen, Germany, 1999.

[LP92] Z. Luo and R. Pollack. *LEGO Proof Development System — Users Manual*. Technical Report ES-LFCS-92-211, Laboratory for Foundations of Computer Science, University of Edinburgh, 1992.

[LP99] L. Lamport and L. Paulson. Should Your Specification Language be Typed? *ACM Transactions on Programming Languages and Systems*, 21(3):133–169, 1999.

[LS87] J. Loeckx and K. Sieber. *The Foundations of Program Verification*. Wiley-Teubner Series in Computer Science. 2nd edition, 1987.

[LSL99] L. Lee, J. Siek, and A. Lumsdaine. The Generic Graph Component Library. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '98)*, 1999.

[Lün81] H. Lüneburg. *Vorlesungen über Analysis*. B.I. Wissenschaftsverlag, 1981. in German.

[Mad00] J. Maddock. *Static Assertions*. available by anonymous ftp from http://www.boost.org/libs/static_assert/static_assert.htm, 2000.

[Mar99] U. Martin. Computers, Reasoning and Mathematical Practice. In *Proceedings of the 1997 NATO ASI Summer School on Logic and Computation*. Springer Verlag, 1999.

[McC97] W. McCune. Solution of the Robbins Problem. *Journal of Automated Reasoning*, 19:263–276, 1997.

[MDS01] D. Musser, G. Derge, and A. Saini. *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library*. Addison-Wesley, 2nd edition, 2001.

[Mes89] J. Meseguer. General Logics. In H. Ebbinghaus et al., editor, *Logic Colloquium'87*, pages 275–329. North-Holland, 1989.

[Mey92] B. Meyer. *Eiffel — The Language*. Prentice Hall, 2nd edition, 1992.

[Mey97] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 2nd edition, 1997.

[Mil78] R. Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.

[Mil84] R. Milner. A Proposal for Standard ML. In *Proceedings of the Symposium on Lisp and Functional Programming*, pages 184–197. ACM, 1984.

[Mit98] J. Mitchell. *Foundations for Programming Languages*. MIT Press, 1998. 2nd printing.

[MN99] K. Mehlhorn and S. Näher. *Leda — A Platform for Combinatorial and Geometric Computing*. Camebridge University Press, 1999.

[Mos97] P. Mosses. CoFI: The Common Framework Initiative for Algebraic Specification and Development. In M. Bidoit and M. Dauchet, editors, *Proceeding of TAPSOFT'97*, volume 1214 of *Lecture Notes in Computer Science (LNCS)*. Springer Verlag, 1997.

[MS87] D. Musser and A. Stepanov. A Library of Generic Algorithms in Ada. In *Using Ada (1987 International Ada Conference)*, pages 216–225. ACM SIG Ada, 1987.

[MS94] D. Musser and A. Stepanov. *The ADA Generic Library: Linear List Processing Packages*. Springer Verlag, 1994.

[MS02a] R. Milewski and C. Schwarzweller. Algebraic Requirements for the Construction of Polynomial Rings. *Mechanized Mathematics and its Applications*, 2:1–8, 2002.

[MS02b] D. Musser and Z. Shao. Concept Use or Concept Refinement: An Important Distinction in Building Generic Specifications. In C. George and H. Miao, editors, *Proceedings of the 4th International Conference on Formal Engineering Methods (ICFEM2002)*, volume 2495 of *Lecture Notes in Computer Science (LNCS)*, pages 132–143. Springer Verlag, 2002.

[MS02c] D. Musser and Z. Shao. *The Tecton Concept Description Language* (Revised Version). Technical Report 02-2, Rensselaer Polytechnic Institute, 2002.

[MSL00] D. Musser, S. Schupp, and R. Loos. Requirement Oriented Programming. In M. Jazayeri, R. Loos, and D. Musser, editors,

Generic Programming — International Seminar on Generic Programming, volume 1766 of *Lecture Notes in Computer Science (LNCS)*, pages 12–24. Springer Verlag, 2000.

[Mus98] D. Musser. *The Tecton Concept Description Language*. available by anonymous ftp from http://www.cs.rpi.edu/∼musser/gp/tecton, September 1998.

[Mye95] N. Myers. Traits — A New and Useful Template Technique. *C++ Report*, 7(5):32–35, 1995. reprinted in [Lip98].

[NPW02] T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science (LNCS)*. Springer Verlag, 2002.

[NT92] Y. Nakamura and A. Trybulec. A Mathematical Model of CPU. *Formalized Mathematics*, 1992. available by anonymous ftp from http://mizar.uwb.edu.pl/JFM/Vol4/ami_1.html.

[ONS93] F. Orejas, M. Navarro, and A. Sanchez. Implementation and Behavioural Equivalence: A Survey. In M. Bidoit and C. Choppy, editors, *Recent Trends in Data Type Specification, Proc. Workshop on Specification of Abstract Data Types (ADT'91)*, volume 655 of *Lecture Notes in Computer Science (LNCS)*, pages 93–125. Springer Verlag, 1993.

[ORS92] S. Owre, J. Rushby, and N. Shankar. PVS: A Prototype Verification System. In D. Kapur, editor, *Automated Deduction — CADE-11*, volume 607 of *Lecture Notes in Computer Science (LNCS)*. Springer Verlag, 1992.

[OS01] S. Owre and N. Shankar. *Theory Interpretation in PVS*. Technical Report SRI-CSL-01-01, SRI International, 2001.

[OSC89] F. Orejas, V. Sacristan, and S. Clérici. Development of Algebraic Specifications with Constraints. In *Proc. Workshop on Categorical Methods in Computer Science with Aspects from Topology*, volume 393 of *Lecture Notes in Computer Science (LNCS)*, pages 102–123. Springer Verlag, 1989.

[Pau96] L. Paulson. *ML for the Working Programmer*. Camebridge University Press, 2nd edition, 1996.

[Pie91] B. Pierce. *Basic Category Theory for Computer Scientists*. MIT Press, 1991.

[RN95] S. Russell and P. Norvig. *Artificial Intelligence — A Modern Approach*. Prentice-Hall, 1995.

[Rob01] A. Robinson, editor. *Handbook of Automated Reasoning*. Elsevier, 2001.

[RS93] P. Robinson and J. Staples. Formalizing a Hierarchical Structure of Practical Mathematical Reasoning. *Journal of Logic Computation*, 3(1):47–61, 1993.

[RT99] P. Rudnicki and A. Trybulec. On Equivalents of Well-Foundedness — An Experiment in Mizar. *Journal of Automated Reasoning*, 23:197–234, 1999.

[RT01a] P. Rudnicki and A. Trybulec. Mathematical Knowledge Management in Mizar. In *Proceedings of the First International Workshop on Mathematical Knowledge Management (MKM2001), Linz, Austria*, 2001.

[RT01b] P. Rudnicki and A. Trybulec. Multivariate Polynomials with Arbitrary Number of Variables. *Formalized Mathematics*, 9(1):95–110, 2001. available by anonymous ftp from http://mizar.uwb.edu.pl/JFM/Vol11/polynom1.html.

[San93] D. Sannella. A Survey of Formal Software Development Methods. In R. Thayer and A. McGettrick, editors, *Software Engineering: A European Perspective*, pages 281–297. IEEE Computer Society Press, 1993.

[Sch96] S. Schupp. *Generic Programming Such That one can build an Algebraic Library*. Ph.D. Thesis, University of Tübingen, Germany, 1996.

[Sch00a] C. Schwarzweller. The Binomial Theorems for Algebraic Structures. *Formalized Mathematics*, 9(3):559–564, 2000. available by anonymous ftp from http://mizar.uwb.edu.pl/JFM/Vol12/binom.html.

[Sch00b] C. Schwarzweller. Mizar Correctness Proofs for Generic Fraction Field Arithmetic. In M. Jazayeri, R. Loos, and D. Musser, editors, *Generic Programming — International Seminar on*

Generic Programming, volume 1766 of *Lecture Notes in Computer Science (LNCS)*, pages 178–191. Springer Verlag, 2000.

[Sch01a] J. Schumann. *Automated Theorem Proving in Software Engineering*. Springer Verlag, 2001.

[Sch01b] C. Schwarzweller. Designing Mathematical Libraries based on Minimal Requirements for Theorems. In *Proceedings of the First International Workshop on Mathematical Knowledge Management (MKM2001), Linz, Austria*, 2001.

[Sch02] C. Schwarzweller. Symbolic Deduction in Mathematical Databases based on Properties. In V. Sorge and S. Colton, editors, *Proceedings of the Second International Workshop on the Role of Automated Deduction in Mathematics, Kopenhagen, Denmark*, 2002.

[Sch03] C. Schwarzweller. Designing Mathematical Libraries Based on Requirements for Theorems. *Annals of Mathematics and Artificial Intelligence*, 2003. to appear.

[Sho67] J. Shoenfield. *Mathematical Logic*. Addison-Wesley, 1967.

[SJ95] Y. Srinivas and R. Jüllig. Specware^{TM}: Formal Support for Composing Software. In *Proceedings of the Confeence on Mathematics of Progam Construction, Kloster Irsee, Germany*. Springer Verlag, 1995.

[SL94] A. Stepanov and M. Lee. *The Standard Template Library*. Technical Report HPL-94-34, 1994. revised 1995.

[SL99] J. Siek and A. Lumsdaine. *The Matrix Template Library — Generic Components for High-Performance Scientific Computing*. IEEE Computer Society, Computing in Science & Engineering, 1999.

[SL00a] S. Schupp and R. Loos. SuchThat — Generic Programming Works. In M. Jazayeri, R. Loos, and D. Musser, editors, *Generic Programming — International Seminar on Generic Programming*, volume 1766 of *Lecture Notes in Computer Science (LNCS)*, pages 133–145. Springer Verlag, 2000.

[SL00b] J. Siek and A. Lumsdaine. Concept Checking — Binding Parametric Polymorphism in C++. In *First Workshop on C++ Template Programming*, 2000.

[SLL99] J. Siek, L. Lee, and A. Lumsdaine. The Generic Graph Component Library. In A. Berman, editor, *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 399–414. ACM Press, 1999.

[SLL02] J. Siek, L. Lee, and A. Lumsdaine. *The Boost Graph Library — User Guide and Reference Manual*. Pearson Education, 2002.

[SM96] Y. Srinivas and J. McDonald. *The Architecture of Specware^{TM}, a Formal Software Development System*. Technical Report KES.U.96.7, Kestrel Institute, 1996.

[SO99] N. Shankar and S. Owre. Principles and Pragmatics of Subtyping in PVS. Invited Paper at the 1999 Workshop on Abstract Datatypes, 1999.

[Som01] I. Sommerville. *Software Engineering*. Addison-Wesley, 6th edition, 2001.

[SST92] D. Sannella, S. Sokołowski, and A. Tarlecki. Toward Formal Development of Programs from Algebraic Specifications: Parameterisation Revisited. *Acta Informatica*, 29:689–736, 1992.

[ST88] D. Sannella and A. Tarlecki. Specifications in an Arbitrary Institution. *Information and Computation*, 76:165–210, 1988.

[ST91] D. Sannella and A. Tarlecki. Extended ML: Past, Present and Future. In H. Ehrig, K. Jantke, F. Orejas, and H. Reichel, editors, *Recent Trends in Data Type Specification: 7th Workshop on Specification of Astract Data Types*, volume 534 of *Lecture Notes in Computer Science (LNCS)*, pages 297–322. Springer Verlag, 1991.

[ST97] D. Sannella and A. Tarlecki. Essential Concepts of Algebraic Specification and Program Development. *Formal Aspects of Computing*, 9:229–269, 1997.

[ST99] D. Sannella and A. Tarlecki. Algebraic Methods for Specification and Formal Development of Programs. *ACM Computing Surveys*, 31, 1999.

[Ste90] G. Steele. *Common LISP — The Language*. Digital Press, 2nd edition, 1990.

[Str67] C. Strachey. *Fundamental Concepts of Programming Languages.* Lecture Notes for the International Summer School in Computer Programming, Copenhagen (reprinted in [Str00]), August 1967.

[Str94] B. Stroustrup. *Design and Evolution of C++.* Addison-Wesley, 1994.

[Str97] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, 3rd edition, 1997.

[Str00] C. Strachey. Fundamental Concepts of Programming Languages. *Higher-Order and Symbolic Computation*, 13(1/2):11–50, 2000. (reprinted from [Str67]).

[SW83] D. Sannella and M. Wirsing. A Kernel Language or Algebraic Specification anf Implementation. In M. Karpinski, editor, *Proceedings of the 11th Colloquium on Foundations of Computation Theory*, volume 158 of *Lecture Notes in Computer Science (LNCS)*, pages 413–427. Springer Verlag, 1983.

[SW99] D. Sannella and M. Wirsing. Specification Languages. In E. Astesiano, H.-J. Kreowski, and B. Krieg-Brückner, editors, *Algebraic Foundations of Systems Specification*, chapter 8, pages 243–272. Springer Verlag, 1999.

[Tar38] A. Tarski. Über Unerreichbare Kardinalzahlen. *Fundamenta Mathematicae*, 30:68–89, 1938. in German.

[Tar39] A. Tarski. On Well-Ordered Subsets of Any Set. *Fundamenta Mathematicae*, 32:176–183, 1939.

[Tho99] S. Thompson. *Haskell — The Craft of Functional Programming.* Addison-Wesley, 2nd edition, 1999.

[Unr94] E. Unruh. *Prime Number Computation.* ANSI X3J16-94-0075/ISO WG21-462, 1994.

[Vel95a] T. Veldhuizen. Expression Templates. *C++ Report*, 7(5):26–31, 1995.

[Vel95b] T. Veldhuizen. Using C++ Metaprograms. *C++ Report*, 7(4):36–43, 1995. reprinted in [Lip98].

[Wad89] P. Wadler. Theorems for free! In *Procceedings of the 4th International Symposium on Functional Programming Languages and Computer Architecture*. Springer Verlag, 1989.

[WB89] P. Wadler and S. Blott. How to Make Ad-hoc Polymorphism Less Ad-hoc. In *Proceedings of the 16th Annual ACM Symposium on Principles of Programming Languages*, 1989.

[Wei03] R. Weiss. *Compiling and Distributing Generic Libraries with Heterogenous Data and Code Representation*. Ph.D. Thesis, University of Tübingen, Germany, 2003. to appear.

[Wil82] R. Wille. Restructuring Lattice Theory: An Approach Based on Hierarchies of Concepts. In I. Rival, editor, *Ordered Sets*, pages 445–470. Reidel, Dordrecht-Boston, 1982.

[Wir86] M. Wirsing. Structured Algebraic Specifications: A Kernel Language. *Theoretical Computer Science*, 42:123–249, 1986.

[Wir95] M. Wirsing. Algebraic Specification Languages: An Overview. In E. Astesiano, G. Reggio, and A. Tarlecki, editors, *Recent Trends in Data Type Specification: 10th Workshop on Specification of Abstract Data Types*, volume 906 of *Lecture Notes in Computer Science (LNCS)*, pages 81–115. Springer Verlag, 1995.

[Wol96] S. Wolfram. *The Mathematica Book*. Wolfram Media and Camebridge University Press, 1996.

[Wol97] P. Wolper. The Meaning of "Formal" — From Weak to Strong Formal Methods. *Journal on Software Tools for Technology Transfer*, 1(1/2), 1997.

[Wos94] L. Wos. OTTER *3.0 — Reference Manual and Guide*. Technical Report ANL-94/6, Argonne National Laboratory, Argonne, 1994.

[WSL01] J. Willcock, J. Siek, and A. Lumsdaine. Caramel: A Concept Representation Sysytem for Generic Programming. In *Second Workshop on C++ Template Programming*, 2001.