Generic Programming

July 21, 1998

A Dagstuhl Seminar on the topic of Generic Programming was held April 27–May 1, 1998, with forty seven participants from ten countries. During the meeting there were thirty seven lectures, a panel session, and several problem sessions. The outcomes of the meeting include

- A collection of abstracts of the lectures, made publicly available via this booklet and a web site at

  http://www-ca.informatik.uni-tuebingen.de/dagstuhl/gpdag.html.

- Plans for a proceedings volume of papers submitted after the seminar that present (possibly extended) discussions of the topics covered in the lectures, problem sessions, and the panel session.

- A list of generic programming projects and open problems, which will be maintained publicly on the World Wide Web at

  http://www-ca.informatik.uni-tuebingen.de/people/musser/gp/pop/index.html
  http://www.cs.rpi.edu/~musser/gp/pop/index.html.

Contents

1 Motivation

This statement, written by the organizers, was included with the invitations to participants.

Generic programming is a sub-discipline of computer science that deals with finding abstract representations of efficient algorithms, data structures, and other software concepts, and with their systematic organization. The goal of generic programming is to express algorithms and data structures in a broadly adaptable, interoperable form that allows their direct use in software construction. Key ideas include:

- Expressing algorithms with minimal assumptions about data abstractions, and vice versa, thus making them as interoperable as possible.
- Lifting of a concrete algorithm to as general a level as possible without losing efficiency; i.e., the most abstract form such that when specialized back to the concrete case the result is just as efficient as the original algorithm.
- When the result of lifting is not general enough to cover all uses of an algorithm, additionally providing a more general form, but ensuring that the most efficient specialized form is automatically chosen when applicable.
- Providing more than one generic algorithm for the same purpose and at the same level of abstraction, when none dominates the others in efficiency for all inputs. This introduces the necessity to provide sufficiently precise characterizations of the domain for which each algorithm is the most efficient.

The intention in this seminar is to focus on generic programming techniques that can be used in practice, rather than to discuss purely theoretical issues. By the end of the seminar we would like to come up with the following results:

1. A list of problems in generic programming. These include new components, new kinds of abstractions, language extensions, tools.
2. A process for extending the existing body of generic components, as well as methods for their specification and verification and for establishing their efficiency in actual programs.

We think that to accomplish these goals we need to share a common vocabulary. Therefore, we will use the vocabulary established by the C++ Standard Template Library (STL) of fundamental data structures and algorithms. This is not intended to preclude discussion of generic programming issues that occur in other areas and that might be more easily illustrated with other libraries and languages. For example, topics might include language extensions to support generic programming in more recent languages such as Haskell or Java, or how generic programming goals intersect with design patterns or frameworks research.

2 Standards Panel

A panel session was held during the meeting on the topic of Library Standardization. The panel members were Matt Austern (SGI), Kurt Melhorn (Saarbrücken), and Wolf Zimmermann (Karlsruhe); Rüdiger Loos moderated. [A report on the session is in preparation.]

3 Lectures

There were thirty seven lectures by participants, on topics that can be classified (post-hoc) into the following categories. Note that in many cases, however, lectures touched on issues in more than one of the categories. Full paper submissions corresponding to many of the talks are being collected for a refereed proceedings publication.

3.1 Foundations and Methodology Comparisons

Fundamentals of Generic Programming

Jim Dehnert and Alex Stepanov

Generic programming depends on the decomposition of programs into components which may be developed separately and combined arbitrarily, subject only to well-defined interfaces. Among the interfaces of interest, indeed the most pervasively and unconsciously used, are the fundamental operators common to all C++ built-in types, as extended to user-defined types, e.g. copy constructors, assignment, and equality.

We investigate the relations which must hold among these operators to preserve consistency with their semantics for the built-in types and with the expectations of programmers. We can produce an axiomatization of these operators which yields the required consistency with built-in types, matches the intuitive expectations of programmers, and also reflects our underlying mathematical expectations.

Automatic Program Specialization by Partial Evaluation

Robert Glück

Partial evaluation is an automatic program optimization technique, similar in concept to, but in several ways different from optimizing compilers. Optimization is achieved by changing the times at which computations are performed. A partial evaluator can be used to overcome losses in performance that are due to highly parameterized, modular software. This has a quite remarkable impact on software development because it allows the design of general and reusable software without the penalty of being too inefficient. This presentation gives an introduction to automatic program specialization by off-line partial evaluation.

Resources Good starting points for the study of partial evaluation are Jones, Gomard, and Sestoft's textbook [JGS93], Consel and Danvy's tutorial notes [CD93], Mogensen and Sestoft's encyclopedia chapter [MS97], and Gallagher's tutorial notes on partial deduction [Gal93].

Further material can be found in the proceedings of the Gammel Avernæs meeting [BEJ88, NGC88], in the proceedings of the ACM conferences and workshops on Partial Evaluation and Semantics-Based Program Manipulation (PEPM) [PEPMa, PEPMb], and in special issues of several journals [JFP93, JLP93, LASC95, TCS98]. A comprehensive volume on partial evaluation appeared in the Lecture Notes of Computer Science series [DGT96].

References

[BEJ88] Bjørner, D., Ershov, A. P., and Jones, N. D. Eds. *Partial Evaluation and Mixed Computation.* North-Holland 1988.

[CD93] Consel, C. and Danvy, O. Tutorial notes on partial evaluation. In *Proc. 20th Annual ACM Symposium on Principles of Programming Languages*, pp. 493–501. ACM Press, 1993.

[DGT96] Danvy, O., Glück, R., and Thiemann, P. Eds. *Partial Evaluation*, Volume 1110 of Lecture Notes in Computer Science. Springer-Verlag, 1996.

[Gal93] Gallagher, J. Tutorial on specialisation of logic programs. In *Proc. ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pp. 88–98. ACM Press, 1993.

[JFP93] *Journal of Functional Programming*, special issue on Partial Evaluation, 3(3) 1993.

[JGS93] Jones, N. D., Gomard, C. K., and Sestoft, P. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall, 1993.

[JLP93] *Journal of Logic Programming*, special issue on Partial Deduction, 16(1&2), 1993.

[LASC95] *Lisp and Symbolic Computation*, special issue on Partial Evaluation, 8(3), 1995.

[MS97] Mogensen, T. Æ. and Sestoft, P. Partial evaluation. In A. Kent and J. G. Williams Eds., *Encyclopedia of Computer Science and Technology*, Volume 37, pp. 247–279, Marcel Dekker 1997.

[NGC88] *New Generation Computing*, special issue on Partial Evaluation and Mixed Computation, 6(2&3), 1988.

[PEPMa] *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation.* ACM Press, 1991, 1993, 1995, 1997.

[PEPMb] *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation.* 1992, 1994.

[TCS98] *Theoretical Computer Science*, special issue on Partial Evaluation. to appear 1998.

Evaluating Generic Programming in Practice

Mehdi Jazayeri

We all think generic programming is wonderful and will lead to improvements in software engineering. But other than an intuitive feeling about it, how do we really know it? How do we convince practicing software engineers that this is a technology they should learn and use? While many researchers are busy developing new techniques and extensions to old ones, we should also pay some attention to concrete measurements of the benefits of the technology. With the appearance of STL, we have an instance of a generic programming library that may actually be useful in practice and therefore can be used in experiments to measure the benefits of the technology. We have conducted one such experiment. The results are not conclusive or even convincing. But it is a start and we see future steps to take. The experiment involved transforming a 5000-line network-management program written in C into a C++ program. The major part of the transformation was to replace hand-crafted C-code to the use of STL data structures and algorithms. We expected to see a reduction in code size because of the replacement of repeated code sequences by calls to library routines. In fact, the code size did not change appreciably. The pure code (.c files) decreased in size while the header files (.h files) increased in size. Intuitively, we can argue that the new code is better because it factors common code into header files. The code size measure, however, does not prove the superiority of the new code. It is possible to conduct experiments to see how maintainable the code is but such experiments are expensive and are usually not convincing. Instead, we prefer to look for code metrics that are more appropriate for evaluating generic code.

Polytypic Programming

Johan Jeuring

Many functions have to be written over and over again for different datatypes, either because datatypes change during the development of programs, or because functions with similar functionality are needed on different datatypes. Examples of such functions are pretty printers, debuggers, equality functions, unifiers, pattern matchers, etc. Such functions are called polytypic functions. A polytypic function is a function that is defined by induction on the structure of user-defined datatypes. This talk introduces polytypic functions, and shows some example applications: pretty printing, data compression, and database table generation.

Recasting Algorithms As Objects: An Alternative to Iterators

Murali Sitaraman

Typical techniques for problem solving separate the steps of choosing data structures and algorithms. This separation and encapsulation of algorithms as procedures exposes representation details, and complicates specification and reasoning. In addition, a procedural encapsulation of algorithms leads to computation of entire solutions in batch style, even for problems where only partial solutions are needed and where partial solutions can be computed efficiently in incremental fashion.

Recasting is an alternative approach in which data structures and algorithms are encapsulated together to address a particular class of problems. Generic data abstractions that result from recasting hide both how and when computations take place, and provide both functional and performance flexibility. When a data abstraction interface is suitably designed and specified, it permits alternative implementations to be developed using different data structures and algorithms so that there is at least one efficient implementation for each intended class of application of the abstraction. Through recasting, for example, the problems of ordering a collection of items and sorting are unified and recast into a Prioritizer data abstraction, which provides an abstract data type and operations to insert items to be prioritized and to remove a next item in order. Instead of a procedure for finding a minimum spanning forest of a graph (and other graph optimization problems), recasting leads to data abstractions that permit edges on a solution to be extracted one at a time. Alternative implementations of these abstractions may compute the solution either incrementally or in batch style, providing clients with a new degree of temporal flexibility.

Recasting is one of the key research results of the RESOLVE work on component-based software engineering framework, discipline, and language. For more information see: http://www.csee.wvu.edu/~resolve

References

B. W. Weide, W. F. Odgen, and M. Sitaraman, "Recasting Algorithms to Encourage Reuse," IEEE Software, Vol. 11, No. 5, September 1994, 80-88.

S. Sreerama, D. Fleming, and M. Sitaraman, "Graceful Object-Based Performance Evolution," Software - Practice and Experience, Vol. 27, No. 1, January 1997, 111-122.

M. Sitaraman, B. W. Weide, and W. F. Odgen, "Using Abstraction Relations to Verify Abstract Data Type Representations," IEEE Transactions on Software Engineering, March 1997, 157-170.

Using Genericity to Improve OO Designs

Karsten Weihe

Translating an abstract object-oriented design into a programming language is a difficult task. We consider one particular problem, the classical circle-ellipse dilemma, and discuss various approaches for resolution. It turns out that common approaches either result in an unsafe realization or in a combinatorial explosion of classes. We show that this combinatorial explosion can be avoided to some extent by introducing layers of generic *holds-a* relations into the inheritance hierarchy. Roughly speaking, these layers turn explicit interrelations between classes into implicit ones, which reduces the complexity of the whole system.

Inheritance, Genericity, and Class Hierarchies

Wolf Zimmermann

The talk reports experiences and conclusions drawn from the development of the class library KARLA (KARlsruhe Library on Algorithms). We observed that combining classes using genericity and inheritance may lead to unexpected errors which might occur deeply in the calling hierarchy of a library. The talk presents results on how to combine classes safely using inheritance and genericity.

We assume that every class C is specified by its invariant Inv_C, and pre- and postconditions $Pre_{m,C}$ and $Post_{m,C}$, respectively, for all its methods m. Consider now method calls $x.m(y_1, \ldots, y_k)$ where the declared type A of x is polymorphic (i.e., x can be an object of any subclass of A) or a parameter of a generic class. Two questions arise:

1. What must be satisfied in order to ensure the precondition of the method call without knowing the concrete class of x?
2. What is satisfied after the method call; i.e., which invariant and postcondition?

The conformance condition is the key to answer these questions. Informally, a class B conforms to a class A iff A can always be substituted by B. Formally, this condition is defined by implications on invariants, pre- and postconditions. If conformance is required for inheritance, then for the method call $x.m(y_1, \ldots, y_k)$ it is sufficient to show that $Pre_{m,A}$ is satisfied, and after the call $Inv_A \wedge Post_{m,A}$ is satisfied. If the declared type of x is a generic parameter T and we bound the genericity by A, i.e., all instances for T must conform to A, we have the same situation.

Inheritance is also used for code reuse. Therefore, there are non-conforming inheritance relations. In particular, the specialization relation turns out to be important in practice. Informally, a class B is more special than a class A, if all objects of class B are the objects of A which satisfy some constraint, e.g. the class of acyclic directed graphs is a specialization of the class of directed graphs. The talk discusses several specialization hierarchies and their systematic construction.

3.2 Programming Methodology

Hierarchical Iterators and Algorithms

Matt Austern

Many data structures are naturally segmented; generic algorithms that ignore this feature, and that treat segmented data structures as a flat one-dimensional range of elements, are unnecessarily inefficient. Segmented iterators make it possible to write generic algorithms that explicitly make use of segmentation.

Generic Programming in C++: Matrix Case Study

Krzysztof Czarnecki

This talk presents a concrete example of a complete generative development method for libraries of reusable algorithmic components and its application to the domain of matrix computations. The overall method is a specialization of the domain engineering method ODM by Simos et al. In the analysis phase, an extended version of so-called feature diagrams is used to model the features of the domain as well as the features of the key concepts and constraints between features are recorded in constraint tables. In the design phase, domain specific languages (DSL) and the implementation component architecture are derived from the domain analysis model. The concrete matrix computation example involves defining a matrix configuration DSL and a matrix expression DSL. The implementation architecture is based on the layered GenVoca model, where the valid combinations of the implementation components are described by an implementation components configuration language (ICCL). Finally, template metaprogramming is used to implement a generator component translating matrix descriptions in the configuration DSL into concrete configurations in the ICCL. The expression DSL is implemented using expression templates to achieve loop fusing and eliminating temporaries. The generative matrix component currently covers 1840 different types of matrices using about 6000 lines of code. The performance of the generated matrices corresponds to the performance of hand-optimized matrix variants.

Generative Programming: Beyond Generic Programming

Ulrich Eisenecker

Most of the current object-oriented analysis and design methodologies focus on developing single systems only. Little support is provided for developing reusable software, e.g. frameworks or component libraries. Furthermore, when we increase the generality of reusable software implemented using conventional technology (e.g. framework and component technology), its complexity grows more than required and it also suffers unnecessary performance losses. Generative programming addresses these three problems. First, it deploys Domain Engineering, which allows us to model classes of systems rather than single systems. Second, aspect-oriented decomposition and composition techniques, domain-specific languages, and configuration knowledge are used to increase the intentionality of system representation and reduce unnecessary complexity. Third, generative programming techniques (e.g. generators, dynamic metaprogramming, and reflection) are used to implement the above concepts and to achieve high performance. This talk presents some generative C++ techniques allowing for parameterization of method binding time and implementation of statically configurable frameworks with statically bound methods and no overhead in time and space incurred by unused polymorphic functionality as in dynamic frameworks.

Generic Programming Using Adaptive and Aspect-Oriented Programming

Karl Lieberherr

Aspect-Oriented Programming (AOP) addresses the problem of tangling of aspects in programs and splits a program into loosely coupled building blocks. Each building block addresses one aspect of the application; some example aspects are: structure, synchronization, remote invocation, quality of service, etc. A weaver combines the building blocks, at least conceptually, into a program where the building blocks are tangled together and sometimes replicated several times.

Adaptive Programming is the special case of Aspect-Oriented Programming where one of the building blocks is expressible in terms of graphs and where the other building blocks refer to the graphs using traversal strategies. A traversal strategy is a partial specification of a class diagram pointing out a few cornerstone classes and relationships. Traversal strategies are graphs where each edge defines a regular expression specifying a traversal through a graph.

The experience with AP has been very positive. It has been used in several industrial projects. Programs become both simpler and more flexible. For more information, see www.ccs.neu.edu/research/demeter

Complete Traversals: Implementation and Generality

Arturo Sánchez-Ruíz

This talk has two parts. The first part presents the concept of *complete traversal* (CT), which for a given a container C (such as a set) is defined by the iteration scheme

$$\textbf{for all } x \in C \\ \mathcal{F}(x, C)$$

where $\mathcal{F}$ is a function that might possibly modify C by inserting new elements into it. We assume that the order in which the elements are treated is not relevant, as long as the iteration continues until $\mathcal{F}$ has been applied to all of the elements currently in C, including those $\mathcal{F}$ has inserted. Standard iteration mechanisms, such as the iterators provided in the C++ Standard Template Library (STL), do not directly support complete traversals. We show generic solutions to the complete traversal problem by means of two generic algorithms and a container adaptor. We also discuss the complexity of these solutions, and their scope in terms of the class of containers they support. This part is based on a joint paper with Dave Musser and Eric Gamess published elsewhere.[1]

The second part of the talk addresses the question of how general this iteration scheme (CT) is. The way of approaching this question is by comparing the CT problem with well-known problems such as the computation of the transitive closure (TC) of a relation, and the computation of the closure of a set under a given relation (CR). We formally define the meaning of *problem* and the relation *is an instance of* among problems, to prove that TC and CR are indeed instances of the CT problem, and present a diagram which depicts the relation *is an instance of* among CT, TC, CR, and other well-known problems.

Template Support for Variation

Georg Trausmuth

Generic programming based on C++ templates can be used to model a generic framework for a family of software systems that share common structure. We developed an approach that uses templates and compile-time instantiation to configure the various members of a system family. We explored the technique in an industrial case study to adapt a generic configuration to meet specific functional requirements imposed by hardware variation. As the software was targeted for an embedded system, the resulting code size is of high interest. We could show that the code is highly optimized through the use of inlined template functions. Optimizations comparable to those achieved are not possible for similar C++ designs which are based on inheritance and virtual functions. A

[1]Eric Gamess, David Musser, and Arturo J. Sánchez-Ruíz. *Complete Traversals and their Implementation Using the Standard Template Library*, in Proceedings of the *XXIII Latinamerican Conference on Informatics*, Valparaíso, Chile. November, 1997.

drawback of our technique is its current limitation to create executables that contain only one variant of the software at a time. As future work we will be testing advanced template techniques to overcome these difficulties for the next versions of the framework.

An Overview of Generic Programming in RESOLVE

Bruce W. Weide

RESOLVE is the name for three related notions: (1) a conceptual framework for thinking about, understanding, and designing component-based software; (2) a research language that supports this conceptual framework; and (3) a discipline for software component engineering within this conceptual framework that can be applied in practice in languages that support generic programming, particularly C++ and Ada. Our emphasis is on "industrial-strength" software systems, so we focus on key technical issues of concern in the software engineering community: correctness, efficiency, and maintainability. Key ideas and mottoes from RESOLVE include contributions to specification design (formal model-based specifications, a.k.a. "domain analysis considered harmful"), behavioral interface design (the swapping paradigm, a.k.a. "assignment considered harmful", and recasting algorithms as objects, a.k.a. "iterators considered harmful"), implementation design (fully-parameterized components, a.k.a. "concrete-to-concrete component coupling considered harmful"), and application design (modular reasoning about component-based software, a.k.a. "visible pointers considered harmful"). The RSRG web page is at http://www.cis.ohio-state.edu/rsrg.

3.3 Applications

Software Components for Computer Algebra

Giuseppe Attardi

Software components encourage code reuse and simplify application development. An increasing number of applications are built assembling components developed by third parties, taking advantage of language-independence, object orientation, ease of use and other features of modern component architectures.

Computer algebra systems could exploit the software component approach, but several issues must be addressed, mostly due to the sophisticated data structures required for representing mathematical objects. We discuss these problems and present a proposal based on the OpenMath specifications.

We built an prototype framework for developing and using mathematical components. The framework uses IDL from CORBA for specifying the interfaces for objects. Code developed in the framework is mapped into either the COM object model for creating ActiveX components or into CORBA objects for creating servers implemented as dynamic modules.

Generic Programming in the POOMA Framework

James Crotinger

POOMA (Parallel Object Oriented Methods and Applications) is a C++ framework for developing portable parallel scientific applications. In order to present a high level of abstraction without sacrificing performance, POOMA depends heavily on generic programming techniques in general, and expression templates in particular (in the form of PETE, the Portable Expression Template Engine). The next version of POOMA, currently under development, introduces the concept of an Engine that is responsible for managing the storage of data and for mapping sub-domains to subsets of the Engine's values. These "Views" are also managed by Engines, possibly of a different type. An Array class, templated on an Engine, has been introduced. This class essentially provides a common user-interface for all Engines (including "View" Engines), and will serve as a basis for implementing the Field, ParticleAttribute, and other data-parallel classes. I present an overview of the current PETE, and discuss the new Engine-based Arrays, the implications that they have on using PETE, and extensions that have been made to PETE to support this new design.

Generic Programming with Black Boxes

Erich Kaltofen and Angel Diaz

The FoxBox system puts in practice the black box representation of symbolic objects and provides algorithms for performing the symbolic calculus with such representations. Black box objects are stored as functions. For instance: a black box polynomial is a procedure that takes values for the variables as input and evaluates the polynomial at that given point. FoxBox can compute the greatest common divisor and factorize polynomials in black box representation, producing as output new black boxes. It also can compute the standard sparse distributed representation of a black box polynomial, for example, one which was computed for an irreducible factor. We establish that the black box representation of objects can push the size of symbolic expressions far beyond what standard data structures could handle before.

Furthermore, FoxBox demonstrates the generic program design methodology. The FoxBox system is written in C++. C++ template arguments provide for abstract domain types. Currently, FoxBox can be compiled with SACLIB 1.1, Gnu-MP 1.0, and NTL 2.0 as its underlying field and polynomial arithmetic. Multiple arithmetic plugins can be used in the same computation. FoxBox provides an MPI-compliant distribution mechanism that allows for parallel and distributed execution of FoxBox programs. Finally, FoxBox plugs into a server/client-style Maple application interface.

STL-Style Generic Programming with Images

Ullrich Köthe

As in other domains, reusability is a major goal of software design in the field of image processing and analysis. Three main requirements must be met by a design method in this domain: First, reusability must not lead to performance degradation. Second, especially good support for algorithm reuse is needed since algorithms are the most important abstractions. Third, the method must lead to independent components that can be combined in many different ways, according to the requirements of the application context.

During the past several years, we have been developing the generic image processing and analysis library VIGRA which achieves these goals by applying the design principles of the Standard Template Library (STL). Based on a detailed analysis of algorithm requirements, we have defined new iterator categories for 2-dimensional and cyclic data access. Iterators conforming to these requirements were implemented for many different image and graph data structures. To further decouple our algorithms from the underlying data representations, we have found promotion traits and data accessors extremely useful, because they allow us to treat scalar and multi-spectral (e.g., RGB) images uniformly.

As a result, we can easily customize our algorithms for the task at hand. This leads to dramatic reductions in source code size (up to 90%) and development time since we do not need different algorithm implementations for minor variations, as was the case in traditional libraries. Our experience with over 50 algorithms currently implemented shows that generic programming on the basis of the C++ template mechanism is well suited to achieving high flexibility whithout loss of performance.

Wrapping Computer Algebra Components with Java and CORBA

Wolfgang Küchlin and Andreas Weber

We address two software reuse problems in the field of Computer Algebra. First, we investigate an architecture for the use of installed remote systems via a Java based Web interface. Second, we show how, and at what levels of granularity, algebraic algorithms can be efficiently reused in other Computer Algebra systems by calling them as CORBA objects. The first problem is illustrated by a parallel Gröbner solver [1] which is best installed on a shared memory parallel SPARC server. While the system is portable to a large degree, porting parallel software is generally much more problematic than sequential software. Therefore it is advantageous to access the remote parallel server via a Java-based Web interface. Our interface includes data format conversion routines for polynomials based on the MathBus protocol. The interface comes in two flavors, one for interactive GUI access, and one for scripting access from source code. For a more detailed exposition see [3].

The second problem is illustrated by the task of extending the Gröbner basis software to include parametric Gröbner bases. The new algorithm needs g.c.d. computations on multivariate polynomials which already exist in various other systems, such as SACLIB. However, the memory management of these systems is incompatible with ours. Therefore we wrapped the necessary SACLIB algorithms as CORBA objects, using the ILU implementation of CORBA, and thus reused the SACLIB code rather than reimplementing the algorithms. We show detailed timings indicating the performance and overhead of this approach. Overall, we found that we could efficiently reuse algorithms at a granularity down to a few tens of milliseconds. For a more detailed exposition see [2].

References

[1] Amrhein, B., Gloor, O., and Küchlin, W. A case study of multi-threaded Gröbner basis completion. In *Proc. Intl. Symposium on Symbolic and Algebraic Computation (ISSAC '96)* (Zürich, July 1996), Y. N. Lakshman, Ed., Association for Computing Machinery.

[2] Hoss, J., Küchlin, W., and Weber, A. Wrapping computer-algebra-systems as software components—the CORBA/ILU approach. In *Proceedings Sixth Rhine Workshop on Computer Algebra* (1998).

[3] Weber, A., Küchlin, W., Eggers, B., and Simonis, V. Parallel computer algebra software as a web component. In *ACM 1998 Workshop on Java for High-Performance Network Computing* (Palo Alto, CA, U. S. A., Mar. 1998), S. Hassanzadeh and K. Schauser, Eds., Association for Computing Machinery, pp. 261–264. `http://www.cs.ucsb.edu/conferences/java98`.

Generic Graph Algorithms

Dietmar Kühl

Implementing (non-trivial) graph algorithms is generally a difficult enterprise due to the complexity of the algorithms. Thus, it is desirable that an implementation of graph algorithms is widely applicable. That means in an ideal world it is possible to apply the implementation of a graph algorithm to an arbitrary graph data structure (as long as it is suited to the requirements of the algorithms: it makes no sense to apply an algorithm for planar graphs to a general graph). Since graph algorithms often extend more basic algorithms, injecting additional computation into an algorithm would also be desirable. Both goals can be achieved, at least to a certain degree. The applicability to an arbitrary graph data structure can be provided by implementing algorithms independent from a graph data structure and using only some kind of minimal abstraction. A workable abstraction consists of the following parts:

- STL-like iterators to access the set of nodes and edges
- adjacency iterators to access the edges incident to a node
- data accessors to access data associated with an object identified by an iterator

These abstractions are sufficient for a large class of algorithms. Additionally, auxiliary modifications of graphs can easily be made by adapters providing a different view of the graph. To also provide additional operations during the execution of an algorithm, the easiest approach is to use algorithm objects which behave similarly to iterators. After initialization the algorithm object is used advanced stepwise until the user of the algorithm wants to execute additional code. The algorithm object iterates through the states of the algorithm with the user having full control over the execution of the algorithm.

Iterators There is nothing special about the iterators accessing the set of nodes and edges; these are just iterators for sequences like, for example, the STL iterators. The only thing to be noted is that the data associated with an object identified by an iterator is not directly accessed via the iterator but rather through a data accessor (see below). Special to graphs are **adjacency iterators** which are used to explore the local structure of a graph. An adjacency iterator iterates over the nodes adjacent to a fixed node. A possible set of methods (in addition to general functions like constructors, assignment, and destructors) for an adjacency iterator could be (`ait1` and `ait2` are adjacency iterators):

`ait1.valid()` Return `true` if there still is an adjacent node

`ait1.next()` Advance the iterator to the next adjacent node

`ait1.cur_adj()` Return an adjacency iterator for the current adjacent node

`ait1.same_node(ait2)` Return `true` if both iterators are positioned on the node (independent from the current node)

`ait1.same_edge(ait2)` Return `true` if both iterators are positioned on the same current edge

Like the iterators for the set of nodes and edges, data for the objects identified by an iterator is accessed through data accessors.

Data Accessors Graph algorithms normally access data associated with the nodes and edges of the graph which is often interpreted as weight, length, cost, or flow, depending on the context of the algorithm. Because graph algorithms often use other algorithms as subalgorithms, the interpretation of the associated data may change, for example, from "length" to "cost." In addition, the algorithms need to store auxiliary data with the nodes and/or edges, for example to indicate a temporary flow value or that a node was already visited. In general, there are many ways to represent the data used by the algorithms. For example, the data may be stored in the node or edge data structure or in some container indexed by node or edge numbers, or the data may be calculated from other fields. Algorithms should not depend on a specific approach for the representation but rather allow the user of the algorithm to select an appropriate representation. Thus, data associated with objects identified by iterators are not directly accessed but rather through some abstraction called *data accessor* with a simple interface such as (`da` is a data accessor, `it` is some iterator, and `val` is an object of the type of the associated data):

`get(da, it)` Retrieve the data identified by `da` which is associated with the object identified by `it`

`set(da, it, val)` Set the data identified by `da` which is associated with the object identified by `it` to the value `val`

In some sense the data associated with a node or an edge can be seen as a table where the rows are identified by an iterator and the columns are identified by data accessors. However, the organization of this table can be very different from a normal table: some columns may be made up from one or more tables while other columns are stored with the corresponding objects directly or are calculated.

Algorithm Objects Graph algorithms are often slightly modified to suit specific needs. For example, during the execution of an algorithm certain intermediate data may be recorded (a typical example is the DFS number during the execution of a DFS) or some data is collected to provide a "certificate" justifying the correctness of a result. Although the additional computations are essential in some situations, they are completely useless in others. To provide the possibility to modify an algorithm, non-trivial algorithms should be implemented as *algorithm objects* which behave much like iterators. The major difference between normal iterators and algorithm objects is that the latter iterate over a set of intermediate states during an algorithm instead of iterating over a collection of objects. Together with appropriate access functions to the current state of the algorithm, this can be used for example to record intermediate data, inject additional computations, modify the behavior of the algorithm, terminate the algorithm prematurely after the desired result is computed, or to provide recovery points to avoid loss of computed results during time consuming computations. This approach is, of course, not specific to graph algorithms. However, it is not yet widely applied (e.g., the STL algorithms are all monolithic blocks).

A Generic Programming Environment for High-Performance Mathematical Libraries

Wolfgang Schreiner

We describe a programming environment for developing generic mathematical libraries with high-performance requirements. The environment is based on the concept of functors as pioneered by SML, but also on a number of original concepts; we especially focus on the combination of the functor-based programming principle with software engineering principles in large development projects. The generated code is highly efficient and can be easily embedded into foreign application environments.

The features of the programming environment include:

- Specifications

  Module interfaces are described by axiomatic specifications that do not only represent syntactic interfaces but also semantic properties (input and output conditions). These properties do not only give precise meanings to modules but are also (with restrictions) used to automatically generate assertions and also default implementations. Specifications may be parameterized to describe functor interfaces; a specification (i.e., also a module) may contain modules as components.

- Functors

  Functors are parameterized over other modules with denoted specifications and return a result module. Contents of other modules may be referenced by qualification or by import into the current environment; multiple functions with the same name but different type interfaces may coexist in the same environment (overloading). Types have constructors and destructors associated to them that are invoked when corresponding values/variables are declared. No direct reference to any builtin type is required, all language constructs can be used in a generic fashion.

- Module Instantiation

  Functors may be instantiated with modules (i.e., the results of other functors) in order to yield executable results. The instantiation system forwards all implementation information transparently to the code generator such that e.g. function code may be inlined and special code may be generated for machine-level operations. The generated C++ code is compiled with the native machine compiler and has the same performance as manually written code.

- Packages

  A sophisticated package handling concept allows to group specifications, functors, module descriptions, and also packages into hierarchies of program units. The environment of program units visible in a package and thus the access to other program units may be explicitly controlled, also may the contents of other packages be imported and thus virtual package environments be constructed.

- Dependence and Permission Control

  The system utilizes an efficient dependence control mechanism which frees the user from any concern about the maintenance of relationships between different program units. This is of particular importance, since a functor is in general the source of different module instantiations on which in turn other instantiations may depend. Likewise, a permission control mechanism allows to share the same package tree by multiple software developers without need for duplication of module instantiations.

The system is currently in transit from alpha to beta state; a corresponding mathematical sample library is under development. We will further develop the system in order to support revision control and to include higher-order functors (functors as components of modules and functors parameterized over other functors).

Generic and Generative Programming in Blitz++

Todd Veldhuizen

Scientific computing requires domain-specific abstractions, such as arrays, matrices, and tensors. Implementing these abstractions in libraries (rather than in compilers) makes them hard to optimize, since compilers lose semantic knowledge of the abstractions. The solution may be to move high-level optimizations out of compilers and into libraries. The Blitz++ library offers an example of how this may be done: using the compile-time metalevel processing abilities of C++, Blitz++ implements many optimizations which were previously the responsibility of compilers. The library offers functionality and efficiency competitive with Fortran, but without any language extensions.

3.4 Specification and Verification

Representing, Verifying, and Applying Generic Software Development Steps Using PVS

Axel Dold

This talk shows how to use the specification and verification system PVS as a framework for formal software development. Software is developed according to the methodology of stepwise refinement: starting from an abstract requirement specification a series of correctness-preserving development steps is applied to obtain an executable and efficient program. We use PVS to provide a fully mechanized treatment of the transformational development process which comprises the formalization (i.e., implementation in the PVS specification language), verification, and correct application of development steps and development methods. A rigorous formal treatment is important since it greatly increases the confidence in the soundness of transformations and their application to specific problems. The software development steps presented in this talk include the well-known algorithmic paradigm of divide-and-conquer, the optimization transformation *finite differencing*, and an example of data structure refinement (implementing finite sets by binary trees). All development steps are represented within a parameterized PVS theory which defines the required data structures and formalizes the application conditions by means of assumptions. Application (i.e., instantiation) of the steps is illustrated by means of simple examples. For example, we show how to correctly derive a *mergesort* program using divide-and-conquer. When applying a software development step the system automatically generates the necessary proof obligations which can be discharged using built-in and user-defined proof strategies.

Filter-based Model Checking of Partial Systems

Matthew B. Dwyer and Corina S. Pasareanu

Recent years have seen dramatic growth in the application of model checking techniques to the validation and verification of correctness properties of hardware, and more recently software, systems. Success in scaling these techniques to be practical for realistic software systems has met with a number of obstacles, such as exponentially-sized state spaces and the problem of constructing finite-state system models for analysis. We describe an automatable approach for building finite-state models of partially defined software systems that are amenable to model checking using existing tools. It enables the application of model checking techniques to individual system components taking into account assumptions about the behavior of the environment in which the components will execute. We illustrate the application of the approach by validating and verifying properties of a reusable parameterized programming framework.

Generic Specification and Verification

Friedrich von Henke, in collaboration with
F. Bartels, A. Dold, H. Pfeifer, H. Rueß

We discuss formal specification and verification in a generic context. Our view is that generic entities, such as templates or generic program components, may benefit from applications of formal methods even more than non-generic ones since often their correctness crucially depends on parameters being properly instantiated; thus the formal specification of semantic requirements on parameters and the verification that instantiations satisfy those constraining requirements is essential.

The talk builds on the experience gained in building up a collection of generic theories for programming language semantics and compilation of simple Pascal-like languages, in the context of project *Verifix*. The basic theories include a development of the required mathematics, including domain theory, fixed-point theory and fixed-point induction; denotational semantics for elementary programming constructs and, derived from this, other forms of semantics. The compilation theories are structured into several conceptually different levels (identifiers and variables, expressions, control structures) in such a manner that each level abstracts as much as possible from details of the lower ones. In all theories, special emphasis is given to modularity and genericity to maximize the degree of reusability. The vehicle for the formal development is the PVS system; we discuss to what extent PVS provides appropriate mechanisms for achieving the desired level of genericity.

Although this work addresses issues of generic programming only in a rather restricted domain (compilation), we believe that both the approach to generic specification and verification and the mathematical theories developed so far provide us with a model that will be applicable to the formal treatment of a much wider variety of generic programming problems.

Applying Larch/C++ to the STL

Gary Leavens

Larch/C++ is a model-based interface specification language tailored to the specification of C++ programs. As background, some basics of model-based interface specifications were described. By giving an abstract model of container objects, one can state conditions that require elements to remain in a container after they are inserted. Aspects of specification related to templates, the so-called “required interface” can be precisely described by Larch/C++, as shown by a case study involving the set template of the Standard Template Library

(STL). This case study revealed some minor documentation problems in the STL, and pointed out some problems in Larch/C++.

Mizar **Verification of Generic Algebraic Algorithms**

Christoph Schwarzweller

We propose the *Mizar* system as a theorem prover capable of verifying generic algebraic algorithms on an appropriate abstract level. The main advantage of the *Mizar* theorem prover is its special input language that enables textbook-style presentation of proofs, hence proofs in the language of algebra. Using *Mizar* we were able to give a rigorous machine-assisted correctness proof of a generic version of Brown/Henrici arithmetic in the field of quotients over arbitrary gcd domains. In this talk we give a short introduction into the *Mizar* language and show how to use the system to verify generic algebraic algorithms. We also deal with proof documentation based on literate programming, claiming that even complex proofs can be presented so that they stay readable and understandable.

Language Independent Container Specification

Alexandre Zamulin

The purpose of this talk is to propose an iterator and container specification mechanism which makes the specification independent of a particular Programming Language and makes the specification formal. The following options have been considered:

- classical algebraic specifications.

  Advantages: sound mathematical foundation, languages and tools exist.
  Disadvantages: state must be modeled, very complex specifications as a result, differences between containers disappear.

- Abstract State Machines (evolving algebras).

  Advantages: imperative style of specification, built-in notion of state.
  Disadvantages: too low level of specification, container operations cannot be formally specified.

The chosen technique, called Object-Oriented Gurevich Machine, combines advantages of both approaches mentioned above. It permits conventionally specifying a needed set of data types and specifying by transition rules a needed set of object types. An object is an entity possessing a unique identifier and a state. It is characterized by a number of attributes defining its state and a number of methods for observing (observers) and changing (mutators) the state. An object type defines a set of states of a particular object.

An iterator is represented as an object of the corresponding iterator type with an address as the object's identifier. Iterator types are combined in iterator categories each having a definite set of iterator operation. A container is represented as an object of the corresponding container type possessing a number of attributes, observers, and mutators. Observers are specified in terms of attribute values. Mutators are specified in terms of transition rules updating attribute values.

3.5 Language Design or Extensions

Piccola—a Small Composition Language

Oscar Nierstrasz

Piccola is a "small composition language" currently being developed within the Software Composition Group. The goal of Piccola is to support the flexible composition of applications from software components.

Piccola can be seen as a "scripting language" in the sense that compositions should compactly describe how components are plugged together. Because Piccola should also document the architectural styles that components conform to, it should also function as an architectural description language.

Since components may come from diverse platforms and adhere to very different architectural styles, a third important aspect is that Piccola can be seen as a "glue language" for adapting components so they can easily work together. Finally, since components and applications are inherently concurrent and distributed, Piccola can also be viewed as a coordination language.

To address these various issues, we propose to develop Piccola based on a formal model of composable "glue agents" that communicate by means of a shared composition medium. Abstractions over messages and agents are first class values, and can be used to adapt compositions at run-time.

Controlling Genericity

Rüdiger Loos

Starting with Knuth's definition of an algorithm we call an algorithm generic if we drop the requirement of definiteness. In 1971, David Musser and George Collins called generic algorithms 'abstract' algorithms and we draw a historical parallel to the concept of abstract or modern algebra introduced at the beginning of this century. Van der Waerden renamed his *Modern Algebra* after 17 successful editions to *Algebra* and we expect that 'generic programming' will one day simply be called 'programming.'

Details of a generic algorithm left unspecified may be the particular domain, the representation of the elements of the domain, and the subalgorithms used. Particular specifications result in definite algorithms. Hence, generic algorithms have to deal with specifications of abstract concepts in order to control genericity.

We propose to use Musser's concept specification language Tecton together with algorithm descriptions for generic programming. The Tecton language was introduced for the Tecton proof system, but in addition it can be combined with generic algorithms in order to verify them, to control type parameter instantiation, and to specify algorithmic efficiency requirements for a library of generic algorithms.

Currently, Tecton has two major applications. It is suitable for expressing a formal, C++ independent specification of the Standard Template Library. Also, it provides a basis of the semantics of Sibylle Schupp's and the author's generic programming language SuchThat and allows for the verification of the axioms of SuchThat's type system. We report on work in progress to implement the Tecton language.

Generic Java—Making the Future Safe for the Past

Martin Odersky, joint work with Philip Wadler and Enno Runne

We present a design to add generic types and methods to the Java programming language. These are both explained and implemented by translation into the unextended language. In fact, there are two such translations: the homogeneous translation maps type variables to a uniform representation, while the heterogeneous translation expands the program by specializing parameterized classes according to their arguments. This talk describes both translations in detail, compares their time and space requirements and discusses how each affects the Java security model.

Generic Programming in SuchThat

Sibylle Schupp

SuchThat is a purely generic programming language under active development. Focusing on type checking issues of generic programming we explain the requirements of a type system for a generic language and give a rationale for the SuchThat type system. We illustrate the challenge of type checking parameterized and attributed structures with examples from computer algebra, and present the implication calculus as a formalism that models well the instantiation of generic algorithms.

Xroma: Extensible Translation

Daveed Vandevoorde

Xroma is a translation environment that supports the concept of smart libraries: libraries that can actively inspect the environment in which they are used and adapt to it. The Xroma system enables this both at compile-time and at run-time through the general principle of reflection. The system exposes syntax trees (called Xromazene), the translation cycle (parsing, semantic checking, ?) and the Xroma object model. Thus the Xroma programmer can substitute library-specific actions at specified points of the translation cycle to take control of the process. These actions are themselves Xroma elements and can be treated as ordinary software library components. Applications include component-specific optimizations, generalized genericity constraints, evolving interfaces, program analysis tools, and the coexistence and derivation of multiple object models (e.g., Xroma-to-COM translation).

3.6 Libraries and Standardization

Exception Safety in Generic Components

David Abrahams

Contrary to popular lore, it is both possible and practical to provide exception-safety in generic components. In fact, exceptions usually lead to more efficient code than "traditional" error-handling methods. Generic exception-safety relies for its foundation on a contractual arrangement between component and client, whereas clients of traditional libraries are usually bound by comparatively few requirements. Also covered are: levels of exception-safety, techniques for distinguishing how much exception-safety should be specified for a component, and an automated testing method for verifying exception-safety.

Issues of the Standard STL

Nicolai Josuttis

The upcoming C++ standard contains the generic template library STL as a library component. However, the library and the standard STL are neither complete nor perfect. So, for the ordinary user there are a lot of practical issues to get the most benefit from using the STL.

Among other things we have the remaining problems of unnecessary inconsistencies, missing minor components (such as some function adapters), not handling const correctness in the best way, and inconveniences in the interface. In addition, we need more and better environments to help the programmer to avoid making mistakes. I simply would like to emphasize the aspect that it is always very important to provide techniques and components that are as intuitive and convenient as possible. And I try to put the focus on the point that it is very important not to forget the details that help great ideas, concepts, and innovations to become an easy-to-use reality.

What Kind of Standards Should There Be for Generic Algorithm Performance?

David Musser

The first premise of this talk is that for maximum usability (including portability), a library of generic software components should be standardized—there can be many different implementations of the library, but every implementation must adhere to a given set of requirements. The recently finalized ANSI/ISO C++ standard is an example. A second premise is that there must be some form of requirements on performance. How should they be expressed? Compared to traditional analysis of concrete algorithms, performance analysis of generic algorithms must be done more abstractly to encompass the range of possible algorithm specializations. But for greatest range of use and most accurate algorithm selection, performance must be characterized more precisely than is possible with O notation. This is an open problem, but looking at it from the standpoint of generic program library standardization may provide new insights and tools to help solve it. Several small examples of performance requirement specification and algorithm design are presented as background to the problem.

Generic Programming in CGAL

Stefan Schirra

CGAL (Computational Geometry Algorithms Library) is a C++ software library of geometric algorithms and data structures. It is developed by several European research institutes and universities, see `http://www.cs.ruu.nl/CGAL` for further information. We give three examples of genericity in CGAL. In the first two examples genericity is achieved by parameterization. Thus they are examples for generic programming via parameterized programming.

The first example is parameterization of the classes in the kernel of CGAL. All constant-size geometric types in the kernel are parameterized by a "representation class." Essentially, this parameter must provide an implementation for the kernel types. CGAL currently offers two concrete models for the concept representation class, a model which uses Cartesian coordinates for the implementation and a model using homogeneous coordinates. Both models are parameterized by the number type used for the coordinates. We use computation of minimum diameter of a set of moving points to illustrate the use of the number type `real` from LEDA (see `http://www.mpi-sb.mpg.de/LEDA`) in the CGAL kernel classes to abolish precision problems.

The second example is data types and algorithms in the basic library part of CGAL. They are parameterized by the types on which they operate and the components that are used to operate on these types. Instead of having a long parameter list, parameters are collected into a single class called "traits class" in CGAL. The name was chosen because of the conceptual similarity to the traits classes used in the C++ standard library. By the parameterization CGAL gains a lot of flexibility and adaptability.

The third example is the *circulator* concept. It has been contributed to CGAL by Lutz Kettner (ETH Zürich). A circulator is a variation of the iterator concept for circular sequences. Circular sequences arise frequently in geometry. CGAL data types provide access to such circular sequences through circulators. Circulators turned out to be very useful in CGAL.