Research proposal: Generic programming — algorithms and tools

Institution:	Department of Computing, University of Copenhagen (DIKU)
Project duration:	1.1.2006–31.12.2008
Principal investigator:	Jyrki Katajainen, Assoc. Prof.
Other investigators:	Hervé Brönnimann, Ass. Prof., Polytechnic University Christopher Derek Curry, M. Sc., Netcompany A/S Amr Elmasry, Ph. D., Alexandria University Claus Jensen, M. Sc. (2005 expected) Andrei Bjarke Moskvitin Josephsen, B. Sc. Stephan Lynge, B. Sc. Torben Ægidius Mogensen, Assoc. Prof. Fabio Vitale, M. Sc. (2005 expected), University of Insubria
Academic partners:	Robert Glück, Programming-language group, University of Copenhagen Rasmus Pagh, Algorithmics group, IT University of Copenhagen Sibylle Schupp, Software-systems group, Chalmers University of Technology

Abstract

The objective of generic programming is to develop software components that allow the highest level of reuse, modularity, and usability. The theoretical challenge taken in this project is to provide genericity without loss of efficiency. The primary goal of this project is to design and analyse efficient generic software components, and provide implementations of such components in the form of program-library modules. The secondary goal is to develop software tools which make the use and development of generic program libraries easier.

1. Background

An ***algorithm*** is a set of instructions that specifies the sequence of operations needed for solving a computational problem, starting from a given set of inputs, if any, and ending at a set of desired outputs, which may include side-effects on associated data structures. Traditionally[1], it is required that each operation is precisely defined (***definiteness***), that an algorithm

[1] Donald E. Knuth, *Fundamental Algorithms*, *The Art of Computer Programming* **1**, 3rd Edition, Addison Wesley Longman (1997), § 1.1

CPH STL Report 2005-5, August 2005. Revised September and October 2005.

is effective in the sense that its operations are sufficiently primitive to be carried out in a finite length of time (***effectiveness***), and that an algorithm always terminates (***finiteness***). In a ***generic algorithm*** the requirement of definiteness is relaxed by letting an algorithm operate on unspecified data types. A generic algorithm is supposed to work for all data types having a given common structure; of course, the amount of such data types can be infinite. A generic algorithm is made definite first after the parametrized data types are specified.

As a concrete example, consider a sorting algorithm which sorts a given sequence of elements in-place in nondecreasing order with respect to a given ordering relation. The signature of such an algorithm could be as follows:

sort⟨*element* E, *sequence*⟨E⟩ S, *ordering*⟨E⟩ F⟩
inputs: s of type S and f of type F
outputs: none but s is modified.

That is, the algorithm takes three type parameters E, S, and F, and two data parameters s and f. The type categories *element*, *sequence*, and *ordering* are to be defined separately; and naturally, the categories *sequence* and *ordering* should operate on the same type of elements to be compatible.

According to Alexander Stepanov[2], who was the principal designer and the original implementor of the C++ Standard Template Library (STL), the objective of generic programming "is to develop a taxonomy of algorithms, data structures, memory allocation mechanisms, and other software artifacts in a way that allows the highest level of reuse, modularity, and usability". As pointed out by Musser et al.[3], generic algorithms were already studied in the 1970's under the name algorithm schemas, but generic programming became popular first after the development of the STL, which later became part of the C++ standard library. In algorithmics community, however, specific issues related to the design, analysis, and implementation of generic algorithms have not been considered until recently.

In ***performance engineering*** the goal is to translate algorithms into efficient computer programs, efficient both with respect to execution time and the amount of space used. The subject is also called ***algorithm engineering***.

At the Department of Computing[4] at the University of Copenhagen the performance engineering laboratory[5] (PE-lab) was founded at the beginning of 1999. The mission of the PE-lab is to educate elite programmers and to do high-quality research related to all aspects of programming, performance programming in particular. Since the foundation of the laboratory one doctor and 16 masters have completed their studies under the supervision of the

[2] David R. Musser, Gillmer J. Derge, and Atul Saini, *STL Tutorial and Reference Guide: C++ programming with the Standard Template Library*, 2nd Edition, Addison-Wesley (2001), xxi-xxii
[3] David Musser, Sibylle Schupp, and Rüdifer Loos, Requirement oriented programming, *Generic Programming*, *Lecture Notes in Computer Science* **1766**, Springer-Verlag, 12–24
[4] http://www.diku.dk/
[5] http://www.diku.dk/research-groups/performance-engineering/

principal investigator (including Claus and Fabio who are expected to finish their M. Sc. studies this year). Of the students supervised, Jeppe Nejsum Madsen was awarded the best-M. Sc.-thesis-in-2003 prize by Dansk Selskab for Datalogi[6].

The goal in the research, for which we apply support, is to design and analyse generic algorithms, and to provide industry-strength implementations of the developed algorithms in the form of program-library modules. The total funding applied is 414 000 DKK from FNU (*rammebevilling*) and 2 880 000 DKK from FTP (a research assistant and a Ph. D. student), to be distributed over the years 2006, 2007, and 2008 (for further details, see appendix 5 of application form no. 1).

The planned research is detailed in §§ 2 and 3; these sections are written for experts in computing and can be skipped by other readers. Some new algorithmic problems raised by genericity are discussed in § 2. Based on the experience with the CPH STL[7] project, which was initiated in autumn 2000, we know that generic programming is tedious, so much of our implementation efforts will be spent on the development of tools that make the use and implementation of generic program libraries easier. The tools that we have at the top of our wish list are given in § 3. The ultimate goal is to generate efficient programs automatically.

The actual research will be carried out in collaboration between the students associated with the PE-lab and other investigators including the principal investigator. The project group is backed up by our academic partners which are all well-known figures in the field: Robert Glück[8] (program committee co-chair for a forth-coming conference on generative programming GPCE'05[9]), Rasmus Pagh[10] (program committee member for two forth-coming theory conferences ICALP'06 and SWAT'06), and Sibylle Schupp[11] (program committee co-chair for a forth-coming workshop on library-centric software design LCSD'05[12]). To encourage new students to take part in the project, Christopher Derek Curry, Robert Glück, and the principal investigator will organize a one-week workshop on generative software development at the beginning of 2006 (week 5).

2. New types of algorithmic problems

The main body of the research done in the PE-lab is basic research — both theoretical and empirical. Often we have no specific application domain in mind, but try to develop general program-library components that can be used in wide range of applications in all major scientific, engineering,

[6] http://www.datalogi.dk/
[7] http://www.cphstl.dk/
[8] http://www.diku.dk/~glueck/
[9] http://www.gpce.org/05/
[10] http://www.itu.dk/people/pagh/
[11] http://www.cs.chalmers.se/~schupp/
[12] http://lcsd05.cs.tamu.edu/

and business areas. In this section we mention a few algorithmic problems which we have encountered in our recent research, and which we have listed as potential topics for further study. The problems listed should give a flavour of the theoretical research planned for the next three years.

Unknown methods as a resource

A generic algorithm consists of two parts: the description of the operations to be executed and the description of the requirements posed for the parametrized types, which may involve type definitions, attribute signatures, and method signatures. Of these the methods are most significant, because the cost of the calls of these methods is unknown. Therefore, it is natural to minimize the number of calls of the unspecified methods.

Let us return to the generic sort algorithm. Normally, the elements are assumed to be assignable so the methods to be supported are copy construction and assignment operation. The cost of these operations may be high if the elements are large objects like strings. Similarly, the cost of the the comparison function is unknown. It is not until recently[13] an in-place sorting algorithm was devised which performs at most $O(n)$ element moves and $O(n \log_2 n)$ element comparisons, where n is the number of elements to be sorted. If space for $4n$ extra bits is available, almost optimal bounds are achievable[14]: $n \log_2 n + 0.59n$ element comparisons and $2.5n$ element moves.

Alternatively, the comparison function can be expensive, e.g. when sorting n strings that are all k characters long, so the objective is to minimize the number of character comparisons. It is well-known that classical sorting algorithms require $O(nk \log_2 n)$ character comparisons, whereas an algorithm performing only $O(nk + n \log_2 n)$ character comparisons is known even if the algorithm is required to operate in-place[15]. Therefore, it must be possible to specialize a generic sorting algorithm such that the improved sorting algorithm is called, instead of a default algorithm, when strings of characters are to be sorted. However, such specialization mechanisms seem to be underdeveloped.

Strength of iterators as a resource

Iterators are objects that point to other objects, call them elements. An ***iterator*** is a generalization of an array pointer; it can be used to iterate over a collection of elements or simply to access the element pointed to.

[13] Gianni Franceschini and Viliam Geffert, An in-place sorting with $O(n \log n)$ comparisons and $O(n)$ moves, *Journal of the ACM* **52** (2005), 515–537

[14] Jyrki Katajainen and Fabio Vitale, Navigation piles with applications to sorting, priority queues, and priority deques, *Nordic Journal of Computing* **10** (2003), 238-262

[15] Gianni Franceschini and Roberto Grossi, Optimal in-place sorting of vectors and records, *Proceedings of the 32nd International Colloquium on Automata, Languages and Programming*, *Lecture Notes in Computer Science* **3580**, Springer-Verlag, 2005, 90–102

As Dehnert and Stepanov[16] write "the best algorithms for some functions (e.g., rotate and random shuffle) differ dramatically for bidirectional and random access iterators". This is true, but often the asymptotic complexity of such functions is the same for both types of iterators. Therefore, in the C++ standard the requirements specified for some generic algorithms are too strong and for some they are too weak. For example, it is required that the sequence given for the sort function supports random access iterators, but it is known that in-place sorting can be done efficiently using forward iterators[17]. In contrast to this, binary search is defined for forward iterators even if for them it has linear cost, not logarithmic as it should. Based on these observations the iterator requirements stated in the C++ standard should be revisited and revised.

Guaranteeing iterator validity

An iterator and the element pointed to live a close symbiosis; when the element is moved, the iterator may become invalid if it is not updated accordingly. A data structure is said to provide ***iterator validity*** if the iterators to its elements are kept valid at all times independent of the element moves done. In the CPH STL it is required that all fundamental data structures (singly/doubly linked lists, singly/doubly resizable arrays, unordered/ordered dictionaries, and singly/doubly-ended priority queues) should support bidirectional iterators, keep the iterators valid under modifications, execute all iterator operations in $O(1)$ time in the worst case, and use linear space on the number of elements stored. These requirements should be compared to those given in the C++ standard: iterator operations are only required to take amortized constant time, the rules for iterator validity are quite arbitrary, and no space bounds are specified.

Theoretically, the issue of iterator validity is settled, but the STL containers allow a very restricted form of iteration. It is not allowed to perform insertions or deletions in a container during a traversal of the container; if these are made, no guarantee is given that all elements will be met during the traversal. Therefore, it might be necessary to consider more general forms of iterator validity like complete traversal mechanisms[18] or partial persistent data structures[19].

---

[16] James C. Dehnert and Alexander Stepanov, Fundamentals of generic programming, *Generic Programming*, *Lecture Notes in Computer Science* **1766**, Springer-Verlag, 1–11

[17] Jyrki Katajainen, Tomi Pasanen, and Jukka Teuhola, Practical in-place mergesort, *Nordic Journal of Computing* **3** (1996), 27–40

[18] David R. Musser and Arturo J. Sánchez-Ruíz, Theory of generality of complete traversals, *Generic Programming*, *Lecture Notes in Computer Science* **1766**, Springer-Verlag, 91–101

[19] Gerth Stølting Brodal, Partially persistent data structures of bounded in degree with constant update time, *Nordic Journal of Computing* **3** (1996), 238–255

Element constructions/destructions as a resource

It is natural to require that for a sequence — like a singly/doubly linked list or a singly/doubly resizable array — every insertion/deletion of an element at the ends of the sequence performs at most one element construction/destruction. In theoretically oriented papers the issue about element constructions/destructions is totally ignored making the algorithms proposed hopelessly slow in a generic environment. In most STL implementations, e.g. in the Silicon Graphics Inc. (SGI) implementation, each update at the ends requires $O(1)$ element constructions/destructions in the amortized sense.

In the CPH STL each such update is required to have the worst-case cost of $O(1)$. For lists and singly resizable arrays it is possible to give the stronger guarantee of at most one construction/destruction per update, but for deques this stronger guarantee is difficult to achieve[20]. In a recent study[21], we observed that for doubly-ended priority queues a similar observation applies, even though for singly-ended priority queues the stronger guarantee is attainable.

3. Development challenges

In the CPH STL project we have been able to create an interesting repository of programs. The development challenges mentioned in this section are in one way or other related to the development and management of this code repository. Even if the wishes are specific for the CPH STL development, in which C++ is used, we expect that in a modified form the tools developed would have wider use.

Extensions to the CPH STL

Currently, we are developing the following extensions to the original STL: arrays, unordered dictionaries, and priority queues. Other natural avenues for extending the library is to consider generic algorithms for string manipulation. Already in the SGI implementation of the STL provided the rope class as an alternative to a string class. In addition to a string container, one should systematically consider traditional string algorithms and make them generic so that they could be applied to solve large-alphabet problems (as in Unicode strings) and small-alphabet, long-pattern problems (as in DNA strings). For an initial work in this direction, see the paper by Musser and Nishanov[22].

[20] Jyrki Katajainen and Bjarke Buur Mortensen, Experiences with the design and implementation of space-efficient deques, *Proceedings of the 5th Workshop on Algorithm Engineering*, *Lecture Notes in Computer Science* **2141**, Springer-Verlag (2001), 39–50

[21] Amr Elmasry, Claus Jensen, and Jyrki Katajainen, Improved worst-case bounds for double-ended priority queues, submitted (2005)

[22] David R. Musser and Gor V. Nishanov, A fast generic sequence matching algorithm, Web document (1998)

C++ without C

When teaching C++ new students have often difficulties in accepting the differences between built-in types and user-defined types, or the C part in C++ in general. Our goal is to provide alternative constructs for the C part of C++. In teaching `struct` can be avoided since it is only a synonym for a public `class`. Using type lists[23] built-in types can be encapsulated in a portable way. Also, the C array could be made a container which may be allocated both from the stack and from the heap; so genericity with respect to allocators is necessary.

In teaching, the efficiency is not the primary concern. It is more important that after these changes clean object-based programming would be possible. However, after minor modifications most compilers should be able to support the encapsulated types with small abstraction penalty. As a consequence of these additions, the use of generic algorithms could be simplified so that they take sequences as inputs, not iterators.

C++ with categories

Template programming can be tedious; a small misunderstanding of the C++ type system may generate a long error message. Even more frustrating is the fact that not all compilers can handle constructs specified in the C++ standard, which was ratified in 1998. In our opinion, the main problem in C++ templates is that they are typeless. This makes compiler construction a real art. One possible improvement is to disallow general genericity, and create a language that only supports constrained genericity. Our work name for such a language is C++ with categories. This idea is by no means new; already in 1995 Alexander Stepanov dreamed about such a language[24].

For example in our sorting algorithm, the type category *element* could be defined as follows:

```
category element {
  @(const @&); // copy constructor
  @& operator=(const @&); // assignment operator
};
```

Here the syntax is an amalgam of C++ and the language-independent specification language proposed by Zamulin[25]. In particular, `@` means the data type being specified.

Categories can be used to express the interactions between different generic components. For example, in the current specification of the STL these interactions are implicit, and therefore these may be difficult to understand by programmers as well as compilers. We would not be as ambitious as

[23] Andrei Alexandrescu, *Modern C++ Design: Generic Programming and Design Patterns Applied*, Addison-Wesley (2001), § 3

[24] Al Stevens interviews Alex Stepanov, Web document (1995). Available at `http://www.sgi.com/tech/stl/drdobbs-interview.html`

[25] Alexandre V. Zamulin, Language independent container specification, *Generic Programming*, *Lecture Notes in Computer Science* **1766**, Springer-Verlag, 192–205

the authors of some earlier proposals; it seems to be difficult to let a compiler check the axioms to be fulfilled by the operations and the complexity requirements of the operations. Our main purpose of proposing C++ with categories is to automate the concept checking which has been successfully implemented above the STL components[26]. Using the developed tools it is possible to build a preprocessor that translates programs written in C++ with categories into C++.

Further development of our benchmark tool

The benchmark tool `benz` was built at the end of 2002 and at the beginning of 2003 by the principal investigator. Later the students associated with the PE-lab have added some new functionality to the tool. Using this tool it has been possible for us to speed up the process of benchmarking programs. Earlier it could take a week to build the necessary scaffolding code for a single module. Now this task can be accomplished in an afternoon or so. The functionality of the tool should be improved before it can be taken into wider use. Moreover, the documentation is only available in Danish.

Automating unit testing

In user-driven software development a software product is released as quickly as possible, and further development is based on customer feedback. Often serious errors are found when the product is in production use. In the CPH STL project we have not been able to provide an official release of the library for two reasons. First, we have not had human resources to do the testing rigorously. Second, we have not wanted to release a defect product. A possible remedy from this dilemma is to automate unit testing.

Technically, such an automation is possible (see, e.g. the paper by Pelliccione et al.[27] and the references therein). As a first step, one could try this approach for the C++ header `<algorithm>`. Based on a specification of the generic algorithms and hints, for which data these should be tested, one could generate test programs automatically. Here the challenge is to design a suitable specification language — a little language[28] as it is called in the literature.

Automating documentation

In the CPH STL project the development work is documented in our design documents which are made public via our website. The main problem with

[26] See the Boost concept check library, which is available at `http://www.boost.org/`

[27] Patrizio Pelliccione, Henry Muccini, Antonio Bucchiarone, and Fabrizio Facchini, TeStor: deriving test sequences from model-based specifications, *Proceedings of the 8th International SIGSOFT Symposium on Component-Based Software Engineering*, *Lecture Notes in Computer Science* **3489**, Springer-Verlag (2005), 267–282

[28] Jon Bentley, *More Programming Pearls: Confessions of a Coder*, Addison-Wesley Publishing Company (1988), § 9

the design documents is that after some changes the documentation is no more up-to-date.

Most parts of the website are generated automatically from the files we have in our CVS repository. However, the system used for the generation of web pages could be considerably improved; in particular, it should include the documentation of the source code itself. The main challenge in the design of such a tool is to understand the format in which the CVS repository stores its information. From this information interesting data could be displayed on the website.

On behalf of the research group

Copenhagen, 30 August 2005

Jyrki Katajainen
Assoc. Prof., Ph. D.