Modular Generics

Jeremy Siek
Open Systems Laboratory
150 S. Woodlawn Ave.
Bloomington, IN USA 47405
jsiek@osl.iu.edu

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features—*abstract data types, constraints, polymorphism*; D.2.13 [**Software Engineering**]: Reusable Software—*reusable libraries*

General Terms

Languages, Design, Reliability

Keywords

Generics, generic programming, polymorphism, modularity, C++, Standard ML, Haskell

1. INTRODUCTION

Generic programming is an increasingly important paradigm for the development of software libraries. David Musser and Alexander Stepanov developed the methodology of generic programming in the late 1980's [6] and applied it to the construction of sequence and graph algorithms in Scheme, Ada, and C. In the early 1990's they shifted focus to C++ and took advantage of templates to construct the Standard Template Library [12] (STL). The STL soon became part of the C++ Standard, which brought generic programming into the mainstream. Since then, generic programming has been successfully applied in the construction of libraries for numerous problem domains [1, 2, 4, 5, 8, 9, 11, 13, 14].

Many programming languages have some support generic programming and some languages are in the process of adding support. In [3] we compared the support for generic programming in the following languages: C++, Java (with the generics extension), Generic C#, Haskell, Standard ML, and Eiffel. Since then we have extended the study to include O'Caml and Cecil. During the study we identified the following properties of languages as important in the construction of generic libraries such as the STL and the Boost Graph Library (BGL) [9].

Multiple constraints Can several constraints be applied to a type parameter of generic function?

Associated type access How easy is it to access types associated with a type parameter, such as the element type of a container, from within a generic function?

Constraints on associated types Is it possible to place constraints on an associated type as part of an interface definition?

Retroactive conformance Can a class be made to conform to an interface without changing its definition?

Separate type checking Is the body of a generic function type checked independently of any call to the function, and are calls to the function type checked with respect to the function's signature and not with respect to the body of the function?

Separate compilation Can a generic function be compiled (to an object file or byte code) independently of any call to the function?

Implicit instantiation Are the type arguments of a generic function deduced from the types of the arguments, or must the caller explicitly instantiate the function?

Type aliases Does the language provide the equivalent of `typedef`?

Scalable syntax Does the amount of text required to compose layers of generic components grow linearly (or worse) with the number of components?

We implemented a non-trivial subset of the BGL in each of the languages and evaluated the languages according to that experience. While several languages performed quite well, no single language had all of the properties we were looking for. C++ is very good in expressiveness and convenience but provides neither separate type checking nor separate compilation for function templates. Even worse, there are loop-holes in the name lookup rules that defeat the modularity provided by namespaces. Standard ML, on the other hand, provides separate type checking and separate compilation for generic components implemented with Functors. However, functors must be explicitly instantiated, which is inconvenient for users of generic libraries. Haskell's type classes provide separate compilation and type checking, and also provide the convenience of implicit instantiation. However, Haskell suffers from some modularity problems: situations can arise where a user may not be able to use two independently developed libraries due to clashing `instance` declarations. We detail these problems in [10].

Generic programming is concerned with the construction of libraries of reusable software components and is inherently about programming “in the large.” Thus, the construction and use of generic libraries is difficult when the host language fails to provide sufficient modularity.

2. GOAL STATEMENT

The purpose of this research is to design a programming language, named $\mathcal{G}$, that improves on the state of the art in language support for generic programming. In particular, the focus will be on improving modularity.

A language for generic programming must provide modularity while at the same time allowing for flexible and rich interfaces. There is significant tension between these two goals, and finding a language design that satisfies both is non-trivial. This research will produce answers to the following questions:

1. Constraints on type parameters are best organized according to abstractions in the problem domain. How should the constraint language be formulated to faithfully and naturally describe abstractions?

2. What role do the constraints play in the type system of the language? How can separate type checking be accomplished while at the same time allowing for caller-provided functionality to be accessed by a generic function. Can the generic function ask for, and the caller provide, functionality in a way that maintains modularity but that does not make generic functions much less convenient to call than non-generic functions?

3. How can separate compilation be achieved for generic functions, and even more challenging, for generic function that dispatch to alternate algorithms based on properties of the input types? What compilation techniques are necessary to provide efficient run-time execution of generic functions?

3. APPROACH AND EVALUATION

This research will produce a **formal definition** of $\mathcal{G}$, including a definition of the **type system** and a single-step **operational semantics**. The formalization will be performed in the Isabelle/Isar proof system [7]. We selected this proof checker because it is industrial strength, the input proof language is human readable, and proof maintenance and refactoring is easier than in many other proof systems. We will prove that the operational semantics respects the static semantics. In addition, we will prove context-independence for modules (the meaning of a function does not depend on the context from where it was called, except for its arguments) and that modules may be freely composed (name clashes, etc. will not occur).

This research will produce a **compiler** that translates from $\mathcal{G}$ to C++. The compiler will allow us to answer questions about separate compilation and efficiency of generic functions. We will prototype several **generic libraries** in $\mathcal{G}$, including a subset of the STL and the BGL, which will provide the experience needed to evaluate the usability of the language with respect to constructing generic libraries. Because the BGL uses the STL, we will also be able to evaluate the use of generic libraries in $\mathcal{G}$.

4. REFERENCES

[1] J.-D. Boissonnat, F. Cazals, F. Da, O. Devillers, S. Pion, F. Rebufat, M. Teillaud, and M. Yvinec. Programming with CGAL: the example of triangulations. In *Proceedings of the fifteenth annual symposium on Computational geometry*, pages 421–422. ACM Press, 1999.

[2] G. Brown, H. Lee, and T. Schulthess. C++ and generic programming for rapid development of monte carlo simulations. In *17th Annual Workshop on Recent Developments in Computers Simulations Studies in Condensed Matter Physics*. Springer-Verlag, 2004.

[3] R. Garcia, J. Järvi, A. Lumsdaine, J. Siek, and J. Willcock. A comparative study of language support for generic programming. In *Proceedings of the 18th ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 115–134. ACM Press, Oct. 2003.

[4] U. Köthe. *Handbook on Computer Vision and Applications*, volume 3, chapter Reusable Software in Computer Vision. Acadamic Press, 1999.

[5] J. Maddock. A proposal to add regular expressions to the standard library. Technical Report J16/03-0011= WG21/N1429, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, March 2003. `http://anubis.dkuug.dk/jtc1/sc22/wg21`.

[6] D. R. Musser and A. A. Stepanov. Generic programming. In P. P. Gianni, editor, *Symbolic and algebraic computation: International Symposium ISSAC ’88, Rome, Italy, July 4–8, 1988: Proceedings*, volume 358 of *Lecture Notes in Computer Science*, pages 13–25, Berlin, 1989. Springer Verlag.

[7] T. Nipkow. Structured Proofs in Isar/HOL. In H. Geuvers and F. Wiedijk, editors, *Types for Proofs and Programs (TYPES 2002)*, volume 2646, pages 259–278, 2003.

[8] W. R. Pitt, M. A. Williams, M. Steven, B. Sweeney, A. J. Bleasby, and D. S. Moss. The bioinformatics template librarygeneric components for biocomputing. *Bioinformatics*, 17(8):729–737, 2001.

[9] J. Siek, L.-Q. Lee, and A. Lumsdaine. *The Boost Graph Library: User Guide and Reference Manual.* Addison-Wesley, 2002.

[10] J. Siek and A. Lumsdaine. Modular generics. In *Concepts: a Linguistic Foundation of Generic Programming*. Adobe Systems, April 2004.

[11] J. G. Siek and A. Lumsdaine. *Advances in Software Tools for Scientific Computing*, chapter A Modern Framework for Portable High Performance Numerical Linear Algebra. Springer, 2000.

[12] A. A. Stepanov and M. Lee. The Standard Template Library. Technical Report X3J16/94-0095, WG21/N0482, ISO Programming Language C++ Project, May 1994.

[13] M. Troyer, S. Todo, S. Trebst, and A. F. and. *ALPS: Algorithms and Libraries for Physics Simulations.* `http://alps.comp-phys.org/`.

[14] J. Walter and M. Koch. *uBLAS*. Boost. `http://www.boost.org/libs/numeric/ublas/doc/index.htm`.