Parametric Polymorphism for Computer Algebra Software Components

Yannis Chicha[1], Michael Lloyd, Cosmin Oancea, and Stephen M. Watt

Ontario Research Centre for Computer Algebra
Department of Computer Science
University of Western Ontario
London Ontario, Canada N6A 5B7
{mlloyd,coancea,watt}@csd.uwo.ca

Abstract. This paper presents our experiments in providing mechanisms for parametric polymorphism for computer algebra software components. Specific interfaces between Aldor and C++ and between Aldor and Maple are described. We then present a general solution, *Generic IDL (GIDL)*, an extension to CORBA IDL supporting generic types. We describe our language bindings for C++, Java 1.5 and Aldor as well as aspects of our implementation, consisting of a GIDL to IDL compiler and tools for generating interface code for the various language bindings.

1 Introduction

As the field of computer algebra software matures, we expect that computer algebra software components developed independently should be composed with the same ease that software modules may be composed in other areas. When we examine what issues specific to computer algebra must be resolved before this can happen, we see that the mathematical genericity of well-structured computer algebra software is not well supported by current technologies for software component architectures. The goal of this paper is to present mechanisms by which computer algebra software components, written independently and in different languages, can take advantage of the code structuring benefits of parametric polymorphism. This leads us to a software architecture that is well suited to the combination of generic mathematical modules, and naturally extends a general-purpose software component architecture.

Parametric polymorphism is a programming language mechanism that allows generic programs to be written, and later on specialized (by supplying specific values for the type-parameters). For example, templates in C++ can be used to provide a module that sorts arrays of elements of any type T for which there is an ordering operation $<: T \times T \to T$. The generic sorting module can then be instantiated with T being the type `int`, `float`, or `double` for example.

[1] Currently at the University of Nice-Sophia Antipolis

Parametric polymorphism has been used in the computer algebra setting for more than two decades. Computer algebra provides a compelling application of parametric polymorphism, where various algebraic constructions, such as polynomials, series, matrices, vector spaces, etc, are used over various different coefficient structures, which are typically rings or fields. The work initiated by Jenks and Trager [8] led to the formulation of domain and category constructors in Axiom [18] [7], and higher-order domain and category producing functions in Aldor [17] [16]. In Aldor, the assertions made in declaring polymorphic modules are used by the optimizer in compiling efficient code that avoids many run-time checks. The computer algebra system Maple now provides a "module" facility for grouping related functions. Parametric polymorphism is achieved naturally by writing Maple functions that return modules.

From the numeric point of view, Ada's mechanism for generics [9] has been used for some time to provide numeric modules that can be specialized over floating point types of different precision. Modern versions of Fortran have provided their own mechanisms to parameterize modules.

Relatively recently, certain main-stream programming languages have provided parametric polymorphism by a "template" mechanism. Both the NTL library for number theory [13] and the Linbox library for symbolic linear algebra [6] use C++ templates to achieve genericity. Version 1.5 of the Java programming language, just released, supports templates and will certainly be the implementation platform for future mathematical software.

There are now quite a few popular programming languages with support for parametric polymorphism, albeit with differing semantics. It is therefore now a significant problem that the standard software component architectures provide no support for genericity through parametric polymorphism. This is primarily due to the historical development that parametric polymorphism started to be widely used only after these component architectures were defined.

Our experiments have all involved interaction between Aldor and other contexts. Aldor's model of parametric polymorphism is based on type-constructing run-time functions, operating at the level of types (Aldor domains) and type-categories (Aldor categories). This is a general mechanism that includes most others as special cases. A related research line (based on functors and categories) goes into proving the correctness of stipulated properties in parameterized domains/classes [3][2].

This paper investigates how parameterized modules, written in different programming languages, can be made to interoperate. We describe our approach for interoperability between C++ and Aldor in Section 2 and between Maple and Aldor in Section 3 (published in [10]). This work motivated a more general approach, where we developed an extension to CORBA-IDL (Interface Definition Language) to support parameterized interfaces. This extended specification language, which we call *Generic IDL* (GIDL) captures a very general notion of parametric polymorphism such that it can meaningfully be supported by various languages, and has the power to model the structure and semantics of systems' components. In Section 4 we present the semantics of GIDL. Section 5 presents

the architecture of our implementation and Section 6 describes the bindings of GIDL for Aldor, C++ and Java. The ideas behind our component architecture "extension" allow it to be easily adapted to work on top of any software component architecture in use today – CORBA is just our working study case.

In this paper we see another instance of a general problem in programming languages first surfacing in the context of mathematical computation: We are unaware of other effort, besides ours, aiming at endowing software component architectures with the parametric polymorphism feature, in a heterogeneous environment (i.e. not assuming the existence of a common intermediate language).

2 Interoperability of Generics between C++ and Aldor

Our initial work in exposing modules with parametric polymorphism across a language boundary arose in the FRISCO project, the ESPRIT Fourth Framework project LTR 21.024. This work has been previously described in project technical reports [4][5], but it is published here for the first time. The two main background items brought into the project were (1) a complex C++ library, PoSSo, for the exact solution of multivariate polynomial equations over various coefficient fields, and (2) an optimizing compiler for a higher-order programming language, Aldor, used in computer algebra. One of the specific objectives of the project was to allow Aldor programs to make use of the PoSSo library.

From this very practical problem arose an interesting challenge in programming languages. On one hand we had a complex library making heavy use of C++ templates. On the other, we had a programming language in which types could be created at run time by user-defined functions. The general problem was how to make use of C++ template libraries from Aldor (and vice versa), not only at a detailed software engineering level, but more importantly to bridge the significant differences in object semantics.

The first step was to define a correspondence between the two languages. It is assumed readers are familiar with C++. Aldor is a strongly typed functional programming language with a higher order type system and strict evaluation. For details on Aldor see [1]. The Aldor type system has two levels: each value belongs to some unique type, known as its *domain*. Domains are in principle run-time values, but they belong to *type categories* which can be determined statically. Categories can specify properties of domains such as which operations they export, and are used to specify interfaces and inheritance hierarchies. The biggest difference between the two-level domain/category model and the single-level subclass/class model is that a domain *is an element of* a category, whereas a subclass *is a subset of* a class. This difference eliminates a number of deep problems in the definition of functions with multiple related arguments. Dependent products and mapping types are fully supported in Aldor. Generic programming is achieved through explicit parametric polymorphism, using functions that take types as parameters and which operate on values of those types, e.g. `f(R: Ring, a: R, b: R): R == a * b - b * a`. The programming style used

to create parametrized types in Aldor is to have functions that return domains, e.g. `Matrix(R: Ring): Module(R) == {`*implementation*`}`.

Both C++ and Aldor provide a set of low-level types, e.g. fixed size integers and floating point numbers, strings, etc, and the correspondence between these low-level types was straightforward.

To use a C++ class from Aldor, a proxy Aldor domain/category pair was created. The category specified the public interface, and the domain provided the implementation. The domain provided exports corresponding to the non-private methods and fields of the C++ class. Because Aldor is not based on classes, the exported operations would all have one extra parameter corresponding to the implicit "self" parameter of the C++ methods. When many C++ classes were used, the inheritance among the Aldor proxy categories would match the inheritance among the C++ classes. The `Join` operation on categories would be used when multiple inheritance was required.

To use Aldor categories and domains from C++, proxy objects would similarly be generated: For each category, a C++ abstract base class would be generated, and for each domain, a C++ class. In both cases (Aldor calling C++ and vice versa), the wrapper proxy would perform their operations through a C foreign function interface.

To use a C++ template class from Aldor, a pair of proxy functions would be created: one function returning a domain value, and the other a category. For the domain-producing function to behave completely natively, it was necessary that it could be called at run-time with any suitable parameter. To achieve this effect, it was necessary to generate an additional small C++ file. A suitable base class was generated for each template parameter, and the C++ template was instantiated over these. Then all the instantiations which an Aldor program would create at run-time could be created through inheritance on this one prototypical instantiation. Because the uses of the C++ template were arising from Aldor code, it was always possible to wrap the parameter types suitably.

To use an Aldor domain-producing function from C++, a proxy template class was generated, and each distinct instantiation of the template would correspond to a different result of the Aldor domain-producing function.

The salient conclusions of the project are the following: Firstly, we can produce the proper binding time semantics by prototypic instantiation of templates. Secondly, we can match generic programming between programming languages with very different base concepts: objects and type-categories. Thirdly, we can produce lightweight proxies to make hierarchies available on either side of the language interface. And finally, the special purpose interface between languages with parametric polymorphism is much more complicated than that between languages not supporting generic types.

While the PoSSo library is perhaps no longer as widely used, the interoperability of C++ and Aldor continues to be of interest, as various mathematical C++ template libraries continue to increase in importance, including NTL and Linbox.

3 Interoperability of Generics between Maple and Aldor

This recent project proposes a non-traditional approach to structuring computer algebra software: using an efficient, compiled language, designed for writing large complex mathematical libraries (Aldor) together with a top-level system based on user-interface priorities and ease of scripting (Maple). The resulting package allows standard Aldor libraries to be used in a standard Maple environment in an effective and natural way. Our investigation has had two goals: First, we are interested in the practical problem of using Aldor as an extension mechanism for the popular Maple computer algebra system. This both allows users to make extensions that operate at the same efficiency as the Maple kernel, and allows Maple users to take advantage of Aldor's mechanisms for structuring correct large-scale libraries. Secondly, we are interested in understanding the issues that arise in matching the compile-time parametric polymorphism of Aldor's dependent types with the dynamic parametric polymorphism of Maple's module-producing functions. We are briefly presenting the main ideas employed in interfacing the two languages (a detailed version is reported in [10]). Elsewhere we have reported on the low-level systems issues that arise in Maple-Aldor interoperability [15].

The key of the high-level interface design was to create dynamic types corresponding to the hierarchy of (available) Aldor types, and to design a dynamic type-checking mechanism for the foreign Maple objects. We have made the following correspondence between Aldor and Maple concepts: An Aldor domain was translated into a Maple function producing at run-time a Maple "module," that in addition encapsulates whatever information is necessary for type-checking its parameters and exports. Aldor's name overloading was simulated in Maple by generating a single polymorphic functions for each export name, with these polymorphic functions dynamically dispatching to the appropriate implementation based on the number and type of arguments. The type-checking mechanism was greatly simplified by the fact that it occurs at the application run-time, when most of the types are instantiated (the exception being function values whose signatures involve dependent types).

To type-check that a Maple domain object belongs to a certain category type, the Aldor's run-time system is invoked ("has" operation). To test that a foreign Maple functional object of type $S_1 \to T_1$ satisfies a functional type $S_2 \to T_2$, one has to verify that S_2 is a subtype of S_1, and T_1 is a subtype of T_2. When testing this, a run-time unification algorithm is used, which computes and works with the fix-point representation of a type (otherwise for mutually recursive types the algorithm will never end). The only non-trivial sub-typing lattices for our type system are the lattices of categories and functions.

While from the scientific perspective we are happy to have discovered a solution to bridge the semantic gap between compiled categorical systems (Aldor) and interpreted functional systems without overloading (Maple), from a pragmatic perspective it is perhaps more important that it is now possible to easily use compiled system with parameterized formal interfaces (Aldor) from a popular computer algebra system (Maple).

4 A General Solution: Generic IDL

The experiments described in Section 2 and 3 proved that we can overcome the semantic gap between two very different environments, and motivated us to explore the possibility of a systematic solution for parametric polymorphism, that should encompass more languages in a simpler way. It was a natural idea to enhance an existing specification language (CORBA's IDL) with a bounded parametric polymorphism system, and to develop tools to generate mappings to the mainstream programming languages. This way, in order to achieve interoperability between `n` programming languages, only `n` translators have to be written instead of `n*n`.

A mechanism to combine modules in different programming languages must be able to support both *compile-time* and *run-time* instantiation of modules, and both *qualified* and *un-qualified* type variables. Our approach is to design a qualification-based generic type model (in which qualifications are defined in terms of sub-typing and/or exported functionality) in order to accommodate the programming languages with dynamic binding times (Java, Aldor), and to enforce these qualifications (in our mapping) for the programming language with static binding time (C++, Modula, etc.). This qualification of generics allows specifications to be clearer, more precise, and easily extensible, featuring the nice property that the type of an expression is context independent.

We constructed a generic model for GIDL in some ways similar to the one existent in Generic Java. We are using a homogeneous implementation approach, based on a type erasure technique ([11][14]) to ensure that our generic extension framework will work, with minimal modifications, on top of any software component architecture that uses an "interface specification language" (CORBA, DCOM, .NET), also preserving the backward compatibility with "old" (non-generic) applications developed in such environments.

CORBA–IDL, as presented in [12], is a declarative language used to describe the interfaces that the client objects call, and the object implementations provide, separating the specification and the implementation aspects of a module. It defines basic types (short, float, double, ...), structured types (struct, sequence, array), and provides signatures for interface types, fully specifying each operation's parameters. Thus, a specification written in CORBA–IDL encapsulates the information needed in order to develop clients using the specified services.

Briefly, the GIDL compiler (consisting of about 33,500 lines of code in 133 Java classes) generates an IDL specification file (by erasing the generic type information), together with stub/skeleton code in the desired programming language (Aldor, C++, Java) to retrieve the erased information. This section briefly presents the semantics of Generic Interface Definition Language (shortly GIDL) – our extension to CORBA–IDL that supports parametric polymorphism, and the code translation mechanism between GIDL and IDL.

We start with an example that will introduce the varieties of parametric polymorphism supported by GIDL. Suppose we want to write a very simple GIDL interface describing a priority queue, as in Figure 1. A type instantiation of an *extension-based qualified* generic type will be validated by the compiler only if it is

```
module GenericStructures {
 interface PriorElem {short getPriority(); short compareTo(in Object r);};
 interface Foo_extend : PriorElem { /* ..... */ };
 interface Foo_export{  //not in a isA logical relation with PriorElem
   short getPriority();    short compareTo(in Object r);
 };

 interface PriorQueue1<A: PriorElem>  // extend-based qualification for A
 { void enqueue(in A a); A dequeue(); boolean empty(); short size(); };
 interface PriorQueue2<A:-PriorElem>  // export-based qualification for A
 { void enqueue(in A a); A dequeue(); boolean empty(); short size(); };

 interface Test<A: PriorElem, B:- PriorElem>{
   Test<Foo_extend, Foo_export> op1(); //line 27 - OK
   Test<Foo_export, Foo_export> op2(); //line 28 - ERROR
   Test<Foo_extend, Foo_extend> op2(); //line 29 - OK
 }; /* end interface Test*/ }; /*end module GenericStructures*/
```

Fig. 1. Generic interfaces with specified qualifications for generic types

actually inheriting from the qualifier. The definition of the `PriorQueue1` interface introduces such a relation, as it specifies a priority queue of objects whose types have to be the `PriorElem` interface or to explicitly extend it (be a sub-type of it). A type instantiation of an *export-base qualified* generic type will be validated by the compiler only if it is found to implement the whole qualifier's functionality The `PriorQueue2` interface accepts as valid candidates for its generic type all the interfaces that export the `short getPriority()` and the `short compareTo(in Object r)` operations. For example, at line 27 in Figure 1, the type checker will successfully verify the `Test<Foo_extend, Foo_export>` scoped-name, because the interface `Foo_extend` is inheriting from `PriorElem`, and the `Foo_export` interface is implementing the whole functionality of the `PriorElem` interface. Line 28 will generate a type check error since `Foo_export` is not inheriting from `PriorElem`, therefore violating the extension based qualification `A: PriorElem`. GIDL also supports an *un-qualified* generic type, similar to that of C++ (e.g. `PriorityQueue3<A>`), that allows the template-candidate to be any of CORBA-IDL's accepted types.

The code translation employed by our type-erasure mechanism is based on the sub-typing polymorphism supported by OMG IDL. It firstly deletes the generic type declarations from the GIDL file. Then, the *non-qualified* / *export-based qualified* type variables will be substituted by the `any` / `Object` IDL type, while the *extend-based-qualified* ones will be substituted with the (type variable erased) interface type it is supposed to extend. The result should be a valid OMG IDL file, which can be compiled with a regular IDL compiler. The generic information lost by this translation will be recovered by generating skeleton/stub wrapper classes under the supported language bindings (Aldor, C++, Java).

5 The architecture of the GIDL base application

This chapter presents a high level view of our GIDL architecture – how the architecture components are created and how they interact to successfully accomplish an invocation. Figure 2 illustrates the design of our proposed architecture. The circles stand for user's code, the rectangle boxes represent components in the standard OMG-CORBA architecture (IDL specification, IDL stub/skeleton, the ORB), the hexagons represent the components needed by our generic extension (GIDL specification, GIDL Wrapper Stub/Skeleton), while the solid arrows describe the method invocation flow.. The dashed arrows represent the *compile to* relation among components. Mainly, a GIDL specification compiled with our GIDL compiler will generate an IDL specification file (where the generic types have been erased, as we have seen in the previous section), together with GIDL wrapper Stub/Skeleton bindings, which will recover the lost generic information.

The bottom part of the figure represents the CORBA's internals. However, with minimal modifications in the wrapper code generation, our generic extension architecture can sit on top of other software component architectures such as .NET or DCOM (i.e. the bottom part of the picture can be replace with the DCOM or .NET internals).

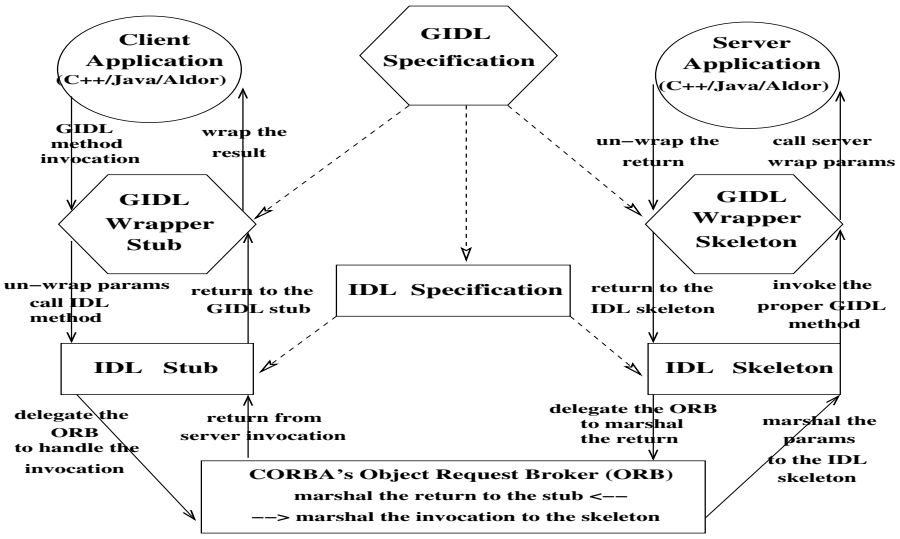

Fig. 2. The GIDL Architecture
Circles: user code, Hexagons: GIDL components, Rectangles: CORBA components
Dashed arrows: "compiled to" relation, Solid arrows: method invocation flow

Our generic extension introduces an extra level of indirection in the original mechanism. For every type in our GIDL specification, we construct a wrapper stub (C++/Java/Aldor), which will keep a reference to a CORBA-stub object

(the one generated by the IDL compiler). When the client is invoking an operation, it is actually calling a method on an GIDL stub wrapper object. The GIDL method implementation retrieves the CORBA-objects hidden by the wrapper-objects taken as parameters, invokes the method on the CORBA-object's stub hidden inside our wrapper class, gets the result, encloses it in a newly formed wrapper if necessary, and returns it to the client application. The wrapper skeleton functionality is the inverse of the client.

To conclude this section, our generic extension for CORBA (our case study) can be applied on top of any CORBA-vendor implementation, maintaining in the same time the backward compatibility with standard CORBA applications. More than this, with minimal changes, our architecture can be applied on top of any heterogeneous system which uses an *interface definition language* (for example DCOM, .NET architectures). Our approach is to design a general and clean extension architecture, and then apply aggressive static/dynamic optimization techniques, to speed up the applications working in such heterogeneous environments (these are left for now as future work). The performance penalty incurred in our design by the extra indirection, and by the boxing/un-boxing mechanism requested by the type-erasure technique, can in most cases be eliminated with a combination of conservative optimizations: code inlining, pointer aliasing, scalar replacement of aggregates, copy propagation, dead code elimination. From the users perspective, our architecture places little burden, as the steps in the application (client/server) design are the same as the ones required for a "standard" CORBA application, but now the client/server implementation can use the generic programming paradigm, as desired.

6 GIDL to Aldor, C++ and Java Mappings

This section presents the high-level translation correspondence involved in the generation of stub/skeleton wrappers for the mapped languages (Aldor, C++, Java). The mapping for Java and C++ are natural extensions of those for the unparameterized case. CORBA does not provide standardized mappings for IDL to Aldor, so we had to define them.

Our wrapper objects, no matter what GIDL type they represent, can be seen as an aggregation of a reference to the "erased" CORBA object/value they represent, the generic type information associated with them, and the "casting" functionality they define. They also inherit the functionality (possibly augmented with generic types) of the IDL type they represent.

The *extend based qualification* is directly supported by both Aldor (through dependent types), and Java (through its *F-bounded* polymorphism). The C++ mapping relies on the C++'s static binding time. A simple cast to the qualifier's type will suffice (enforcing the condition that the substituted type has to inherit from the qualifier, otherwise a compile-time error will be generated). The discussed mapping adds virtually no run-time overhead as it is provable that the verification code (the casting) is not reachable, therefore exposed to copy propagation and branch elimination optimizations.

The *export-based qualification* is reduced to an *extend-based qualification* relation at the GIDL level. The algorithm (not reported here due to space constraints), is based on the construction of the corresponding *most general generic unifier (MGGU)* interface, and works under the assumption that the GIDL compiler has access to the entire specification at compile-time. The Aldor mapping does not use the *MGGU* construct, but provide a simpler translation scheme, where the type parameter is qualified by an un-named Aldor category (defined by means of the *with* construct: GIDL's `interface Dom<A:-Qualifier> {...}` is translated to Aldor's `Dom(A: with Qualifier): DomCat == add {...}`).

The remaining of this section briefly presents the mapping correspondences between GIDL and the three considered languages (C++, Java, Aldor). GIDL basic types (short, long, etc) are mapped into corresponding C++, Java classes and Aldor domains. The user expected functionality is provided by means of operator overloading for the languages that support this feature (Aldor, C++), or by Java "boxing" classes (see java.lang.Integer).

GIDL modules are translated into C++ namespaces, Java packages, and Aldor packages (domains not belonging to any category). GIDL interfaces are translated into possible parameterized classes in C++, interface-class pairs in Java, and domain-category pairs in Aldor. Because Aldor is not based on classes, wrappings of the exported operations receive one extra parameter corresponding to the implicit "self" parameter of the GIDL methods. The multiple inheritance hierarchy at the GIDL level is directly translated into multiple inheritance of classes in C++, interfaces in Java, while for Aldor the "join" operation on the proxy categories will be used.

GIDL scopes directly create C++ and Aldor scopes as both languages provide support for nested classes/domains that match the semantics of the GIDL nested structures (i.e. they are exported to the global scope). GIDL scopes correspond to Java packages; GIDL structures that are nested in the scope of a generic interface and are using some generic types defined by that interface are mapped into a parameterized Java final class (the generic types are duplicated and migrate to the structure definition as well, since the structure will not be in the scope of the corresponding generic interface any more).

One drawback for the Java mapping is that it requires the user's help. Java does not support object instantiation of a generic type (`new A()`), nor reflection feature on its generic types. The constructor of a parameterized class (which is the mapping of a GIDL type) will force the user to pass an extra parameter for each generic type introduced by that class. We need this because otherwise we cannot enforce an exact boxing/un-boxing mechanism between our wrapper objects and CORBA-stub objects.

7 Conclusions

Parametric polymorphism is an essential software structuring technique for computer algebra, where mathematical algorithms achieve their most natural form when expressed generically. As generic computer algebra components are in-

creasingly implemented in different languages (such as Aldor and C++, and Java in the future), it becomes increasingly important that we are able to use these together in a natural way.

The overall goal of this work has been to examine the feasibility of exposing parametric polymorphism for use in the context of multi-language software component systems for computer algebra. We have shown that there are no major impediments to doing this.

We have examined two settings where the semantics and binding times of the parametric polymorphism were substantially different and have found unifying semantics to allow meaningful inter-operation: Our experiments using parameterized C++ modules with Aldor show that we can emulate dynamic parameterization with static templates. They also show that we can effectively model differing forms of parameterized modules with each other: template classes with dependent higher-order function types, and vice versa. Our experiments in using Aldor with parameterized Maple modules have shown that we can generate code to handle dynamically in Maple matters that are handled statically in Aldor. This allows Aldor to be used as an elegant kernel extension language for Maple.

To allow computer algebra software components to participate naturally in a general software environment, we have addressed the problem of supporting parametric polymorphism in standard software component architectures. Our investigation was at two levels: first to determine what sort of parameterization mechanisms could be well defined in a multi-language setting, and secondly how could these be supported. We have seen from our implementation of Generic IDL that qualification (restriction) of type parameters can be enforced in various target languages, even when the target language does not support qualification of its generics. We have shown that both extension-based, and export-based qualification can be supported effectively. The former is easy to implement, with all the verification purely at the GIDL level. The later, we have introduced as a less restrictive form of qualification to allow existing hierarchies to be used without modification. It requires, however, generating (non-executed) code which is checked by the target language compiler. We have shown that this parameterization in GIDL can be supported by translation to IDL, with the generation of appropriate stub/skeleton wrappers. This allows such code to be used with existing CORBA implementations, and to take advantage of the usual support for distributed applications. Note, however, that this has meant we have restricted ourselves to a minimum extension of IDL. Applications that are not distributed may make use of GIDL simply to support multi-language use of generic modules. This use involves minimal overhead.

While many special-purpose programming languages have supported parametric polymorphism for some time, it has really only been C++ which has been in mainstream use. Now, with the availability of generics in Java, it is rather important that we understand how to support generics in a multi-language setting. This paper has aimed to make a contribution in this area. The fact that parametric polymorphism has been in use in computer algebra for over two decades

has forced us to consider parametric polymorphism for software component architectures before the need was evident in wider contexts.

References

1. Aldor User Guide, http://www.aldor.org/aldoruserguide/, 2003.
2. B. Buchberger. SFB Report No. 2003-49 - Groebner Rings in Theorema: A Case Study in Functors and Categories. Technical report, Johannes Kepler University Linz, 2003.
3. B. Buchberger, T. Jebelean, F. Kriftner, M. Marin, E. Tomuta, and D. Vasaru. A Survey of the Theorema Project. In *Proceedings of ISSAC'97*, pages 384–391, 1997.
4. Y. Chicha, F. Defaix, and S. Watt. TR537 - The Aldor/C++ Interface: User's Guide. Technical report, Computer Science Department - The University of Western Ontario, 1999.
5. Y. Chicha, F. Defaix, and S. Watt. TR538 - The Aldor/C++ Interface: Technical Reference. Technical report, Computer Science Department - The University of Western Ontario, 1999.
6. J. G. Dumas, T. Gautier, M. Giesbrecht, P. Giorgi, B. Hovinen, E. Kaltofen, B. D. Saunders, W. J. Turner, and G. Villard. LinBox: A Generic Library for Exact Linear Algebra. In *Proc. of ICMS*, pages 40–50. A. Miola ed. Academic Press, 2002.
7. R. D. Jenks and R. S. Sutor. *AXIOM, The Scientific Computation System*. Springer-Verlag, 1992.
8. R. D. Jenks and B. M. Trager. A Language for Computational Algebra. In *Proc. SYMSAC*, pages 6–13. ACM, 1981.
9. H. Ledgard. *ADA Reference Manual*. Springer: Berlin ; New York, 1980.
10. C. Oancea and S. M. Watt. A Framework for Using Aldor Libraries with Maple. In *Proc. EACA 2004*, pages 219–224, 2004.
11. M. Odersky, P. Wadler, G. Bracha, and D. Stoutamire. Making the future safe for the past: Adding Genericity to the Java programming language. In *ICMS'2002 Proceedings*, pages 183–200, 1998.
12. OMG. Common Object Request Broker Architecture — OMG IDL Syntax and Semantics. Revision2.4 (October 2000), OMG Specification, 2000.
13. V. Shoup. NTL: A Library for doing Number Theory, still valid on july 30st, 2004, http://www.shoup.net/ntl/doc/tour.html.
14. M. Viroli and A. Natali. Parametric Polymorphism in Java: an Approach to Translation Based on Reflective Features. In *OOPSLA'00 Proceedings*, pages 146–165. ACM, 2000.
15. S. M. Watt. A Study in the Integration of Computer Algebra Systems: Memory Management in a Maple-Aldor Environment. In *Proc. International Congress of Mathematical Software*, pages 405–411, 2002.
16. S. M. Watt. Aldor. In J. Grabmeier, E. Kaltofen, and V. Weispfenning, editors, *Handbook of Computer Algebra*, pages 154–160, 2003.
17. S. M. Watt, P. A. Broadbery, S. S. Dooley, P. Iglio, S. C. Morrison, J. M. Steinbach, and R. S. Sutor. *AXIOM Library Compiler User Guide*. Numerical Algorithms Group (ISBN 1-85206-106-5), 1994.
18. S. M. Watt, R. D. Jenks, R. S. Sutor, and B. M. Trager. The Scratchpad II Type System: Domains and Subdomains. In *Proc. of Computing Tools For Scientific Problem Solving*, pages 63–82. A. Miola ed. Academic Press, 1990.