Parametric Polymorphism for Software Component Architectures and Related Optimizations

by

Cosmin Eugen Oancea

Graduate Program in Computer Science

Submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy

Faculty of Graduate Studies
The University of Western Ontario
London, Ontario
July, 2005

THE UNIVERSITY OF WESTERN ONTARIO
FACULTY OF GRADUATE STUDIES

CERTIFICATE OF EXAMINATION

Chief Advisor	Examining Board
_____	_____
Advisory Committee	
_____	_____
_____	_____
_____	_____

The thesis by
Cosmin Eugen Oancea
entitled

Parametric Polymorphism for Software Component Architectures and Related Optimizations

is accepted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

_____	_____
Date	Chair of Examining Board

Abstract

Parametric polymorphism has become a common feature of mainstream programming languages, but software component architectures have lagged behind and do not support this feature. The immediate consequence is that applications cannot naturally combine the functionality exposed by various parameterized modules, if it happens that the implementation language differs. This significant problem surfaced first and most acutely in the computer algebra community, where parametric polymorphism is heavily used for the specification and enforcement of the algebraic interfaces and in the implementation of algorithms that work over various coefficient rings or fields. Complex, specialized mathematical libraries, servicing disjoint areas are implemented in various languages and therefore they cannot yet work together to attack increasingly difficult problems. This thesis examines the problem of accommodating parametric polymorphism, and related optimizations in a multi-language, distributed setting.

We report on a first experiment, where we developed the Alma framework that allows Aldor libraries to extend Maple in a effective and natural way, and constitutes a new approach to structuring computer algebra systems. The motivation for this experiment are twofold: First, 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, and in matching the Aldor's strongly type system with Maple's dynamically typed system. Second, we are interested in the practical problem of using Aldor as an extension mechanism for the popular Maple computer algebra system.

The details of generics, templates or functors, as they are variously called, differ significantly in different programming languages. We investigated how to resolve different binding times and parametric polymorphism semantics in a range of representative programming languages, and identified a common ground that can be suitably mapped to different language bindings.

We explore the possibility of a systematic solution for parametric polymorphism, that should encompass many languages in a simple way. We present a generic component architecture extension that provides support for parameterized components, and can be easily adapted to work on top of various software component architectures in use today: CORBA, JNI, DCOM. We have implemented and tested our extension on top of CORBA.We present *Generic Interface Definition Language (*GIDL*)*, an extension to CORBA-IDL, supporting generic types, and our language bindings for C++, Java, and Aldor. We describe our implementation of GIDL, consisting of a GIDL to IDL compiler and tools for generating linkage code under the language bindings.

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 system's components. To test the effectiveness of our model for generics, we have investigated how to expose C++'s STL and Aldor's BasicMath libraries to a multi-language environment, and discuss our mappings in the context of automatic library interface generation.

Our work in the context of exposing generic libraries to a multi-language, potentially distributed environment has revealed several performance issues. First, as different components are separately compiled, the traditional compiler optimizations, such as inlining and parallelization, will fail to perform aggressively. Second, the overhead introduced by the inter-process communication stalls can be quite significant. Finally, this thesis explores *speculative optimizations* in the attempt to speed up the application performance in distributed environments.

Acknowledgments

First of all, I would like to express my sincere gratitude to my supervisor, Professor Stephen Watt, for his valuable guidance, continuous interest, and support for this research work.

I wish further to express my sincere appreciation to Jason Selby and Professor Mark Giesbrecht for the many hints, valuable discussions, comments and collaboration related to the "Compiler Middleware" NSERC strategic grant projects.

I am very grateful to Professor Hanan Lutfiyya for carefully reading and sharing her expertise in connection to three of my papers and for her helpful suggestions, support and feedback.

I would like to thank my colleagues and faculty at the ORCCA lab, Western Ontario University of whom many contributed to this thesis in one way or another. I also express my heartfelt thanks to all Faculty, staff, and fellow graduate students in the Computer Science Department.

Above all, I thank my family for their encouragements, support, and love during all these years. A further special thanks goes to my mother, Lucia Baltag, for being to me a model of determination, perseverance, and hard work, and to my wife Monica Ulici for her patience and understanding during the course of my graduate studies.

Chapters of this thesis are based on co-authored papers accepted to the 2005 International Symposium on Symbolic and Algebraic Computation, the 2005 International Conference on Object-Oriented Programming, Systems, Languages and Applications, and the 2005 International Conference on Parrallel and Distributed Processing Techniques and Applications. I thank again the co-authors Stephen Watt, Jason Selby, and Mark Giesbrecht for the fruitful collaboration.

Table of Contents

List of Tables

List of Figures

List of Definitions

Chapter 1

Introduction

This thesis studies mechanisms by which software components can take advantage of the code structuring benefits of parametric polymorphism, in a multi-language, potentially distributed environment. We introduce two component-based architectures that are well suited to combine generic modules and naturally extend general-purpose, lower level interoperability solutions. To test our frameworks, we have conducted several experiments to translate comprehensive parts of existent generic libraries across language boundaries. This work has revealed several performance issues, especially in a distributed setting, and consequently, we have investigated speculative-based optimizations to reduce the observed overhead. Before describing the objectives of this thesis in detail, we briefly describe what parametric polymorphism is, where its application is most beneficial, and why we think it is important to investigate it in the context of software component architectures.

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*.

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. This has led to the appearance of many generic mathematical libraries, implemented in various languages, such as: the NTL library for number theory [61], the Linbox library for symbolic linear algebra [19], the Sumit library for differential operators [3], the Triade library for triangular sets [39], to restrict ourselves to just a few.

A significant problem however is that there is virtually no re-use of these libraries outside of their original language. On one hand, the programmer finds that to compose these heterogeneous pieces of software is difficult and rather unsafe. On the other hand, the library developer finds that much of the programming effort is spent in implementing functionality corresponding to common mathematical structures and operations, already existent in other libraries. These libraries attempt to service different problems from the same or disjoint mathematical areas, but the point is that their functionalities significantly overlap.

When we examine what issues prevent the software components to be easily combined, we find that parametric polymorphism, which has become a common feature of mainstream programming languages, is not yet well supported by current technologies for software component architecture. One reason for this is that the semantics of generics and the binding time models they use differ substantially in various program-

ming languages. It is thus important, from both a research and practical perspective, to investigate what should be the attributes of a generic model that can successfully expose parametric polymorphism across language boundaries.

Our work in this context was inspired by an experiment from the FRISCO project, where two languages with very different binding times and parametric polymorphism models were made to interoperate [10, 11]. The experiment used C++ with compile-time template instantiation, and Aldor [70, 69] with run time higher-order functions producing dependent types.

We started our work with an experiment in which we have studied a method to use compiled, strongly typed Aldor domains in the interpreted, expression-oriented Maple environment. Our framework, *Alma*, proposes a non-traditional approach to structuring computer algebra software: using an efficient, compiled language, designed for writing large complex mathematical libraries together with a top-level system based on user-interface priorities and ease of scripting. Alma allows Aldor functions to run tightly coupled to the Maple environment, able to directly and efficiently manipulate Maple data objects. Our solution builds on top of several low-level foreign function interfaces: Aldor to C, and C to Maple. Since the computational models of Maple and Aldor differ significantly, we employ new run-time code to implement a non-trivial, high-level semantic correspondence between the two languages. In particular, we must match up different flavors of parametric polymorphism: Aldor's statically-checked dependent types with Maple's modules-producing functions. In general, we provides a mechanism to *safely* interoperate a statically *vs.* a dynamically typed language.

The Aldor/C++ iteroperability experiment proved that one can overcome the inter-language semantic gap, 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 [49], with a bounded parametric polymorphism system, and to develop tools to generate mappings to a set of representative languages: C++, Java, Aldor. This covers a comprehensive range of parametric polymorphism semantics and binding time models. We have dubbed this *Generic IDL* or GIDL for short. To test our architectural design, we have exposed part of C++'s *Standard Template Library (*STL*)* and Moreno Maza's library for triangular set decomposition to use across the GIDL and Alma frameworks respectively.

The GIDL framework, presented in Chapter 4, is based on a generic-type erasure technique, and consists of a GIDL to IDL compiler that performs the erasure, and tools to automatically generate skeleton/stub wrappers for C++, Java, and Aldor languages to recover the lost generic information. The Aldor skeleton/stub is currently being implemented by Michael Lloyd, and the main mapping ideas, in which I was involved, are also mentioned here. Our component architecture "extension" does not assume homogeneous environment: a common intermediate representation to which the languages are compiled, like Sun's bytecode and Microsoft's common intermediate language. GIDL's design allows it to be easily adapted to work on top of other software component architectures (JNI, DCOM, etc) – CORBA is just our working study case.

In Generic IDL, we have tried to capture a very general notion of parametric polymorphism such that it can be meaningfully supported by various languages, and has the power to model the structure and semantics of system's components. To test our architectural design we have constructed a clean mechanism to expose part of

one generic library (STL) to a multi-language environment (Section 4.7). We found GIDL to be better suited to express the STL's semantics, as our specification is self-explanatory and self-constrained: it does not need free language annotations as in the C++ case.

The Alma framework is presented in Chapter 3. Since the language models of Maple and Aldor differ, the interface code implements various semantic correspondences: Aldor's features like run-time domain types, overloading, dependent types, mapping types are mapped at the Maple level. We are thus implicitly performing a language extension. The key for the translation of these features is to create, via the Maple stub, dynamic types corresponding to the hierarchy of available Aldor types, and to design a dynamic type-checking mechanism for the foreign Maple objects. We found that *Alma* can foster a richer connectivity between the two languages considered, compared to adding Maple to the GIDL framework, as several high level concepts (closures, transparent casting) and related optimizations could not be directly supported by GIDL.

Our work in the context of exposing generic libraries to a multi-language, potentially distributed environment has revealed several performance issues. First of all, as different components are separately compiled, static compiler optimizations, such as inlining, tower type, parallelization, cannot be performed aggressively. Secondly, in the distributed case, the overhead introduced by the inter-process communication can be quite significant. Chapter 5 introduces a novel application of thread-level speculation to a distributed heterogeneous environment, in which we propose and evaluate two speculative models that attempt to address the previously mentioned performance-related issues. The first model effectively reduces the method call overhead associated with distributed objects by overlapping the client-server commu-

nication with useful computation performed on the server side under the form of speculation. The second model simulates "procedural inlining", it is more aggressive and yields better speed-ups than the first, however security concerns associated with code migration may in some cases prevent its use.

This thesis is organized as follows: Chapter 2 summarizes the necessary background and related work. It reviews the interoperability experiment between C++ and Aldor that served as motivation for our work, summarizes the parametric polymorphism semantics in several languages, reviews the mainstream component technologies in use today, and looks into the current approaches and applications of thread level speculation (TLS). Chapters 3 and 4 introduce the *Alma* and GIDL frameworks respectively, while Chapter 5 proposes two distributed TLS optimizations that may be employed to effectively speed-up distributed computations.

We are currently unaware of any other effort, besides ours, aimed at endowing software component architectures with parametric polymorphism in a heterogeneous, multi-language environment, or at exploring the specific optimizations that should be applied in such a context. 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 for any significant time. Now, with the availability of generics in Java, it is rather important that we understand how to support generics in a multi-language setting.

Chapter 2

Background and Related Work

This chapter is organized as follows: Section 2.1 reviews an early experiment in which two languages with very different semantics, namely Aldor and C++, were made to interoperate. The experiment motivated part of the work encompassed in this thesis, in which we investigate a more general and systematic approach to unifying the different flavors of parametric polymorphism and binding time semantics across language boundaries.

Section 2.2 lays out the context of the work, summarizing the parametric polymorphism semantics in various languages and the design point we desire to satisfy (Section 2.2.2), together with a brief introduction to the (formal) theory of type systems with emphasis on parametric polymorphism (Section 2.2.1).

Section 2.3 reviews the component technologies in use today, and observes that out of all of these, only Microsoft's .NET provides support for parametric polymorphism, and only in the context in which all mapped languages implement the same semantics and binding time models. The study of these architectures has revealed some common points, that have allowed us to develop a *generic* extension solution for parametric polymorphism in heterogeneous systems (described in Chapter 4).

Finally, section 2.4 looks into the current approaches and applications of thread level speculation (TLS), and helps to understand what should be the characteristics of such a system that targets a distributed environment.

2.1 Early Experiment

FRISCO (A Framework for Integrated Symbolic/Numeric Computation) [41] was a three-year project that aimed at providing algorithmic and software tools for solving complex polynomial systems.

Many research groups were involved and languages including C++, Fortran and Aldor [70] were used. Several powerful mathematical libraries had been developed in C++ using template programming. At the same time, developers of new algebraic libraries were quite interested in using Aldor, as the language proposed many interesting features. We do not assume the reader to be familiar with the Aldor language, therefore we briefly introduce it in Section 2.1.1. Interoperability between these languages (C++, Aldor, Fortran) was a goal of the project. Aldor language already provides an interoperability layer with Fortran, and the difficult step was to design a bi-directional semantic correspondence between Aldor and C++. This step is succinctly presented in Section 2.1.2.

2.1.1 The Aldor Programming Language

Aldor [68, 70] is a strongly typed functional programming language with a higher order type system and strict evaluation. Aldor has been used primarily in the area of symbolic mathematics software. The type system has two levels: each value belongs to

Figure 2.1 Aldor category/domain example

```
1
2  define Module(R: Ring): Category == Ring with {
3     *: (R, %) -> %;
4  }
5
6  Monomial(R: Ring): Module(R) == add {
7     Rep == SingleInteger;
8     import from Rep;
9     (r: R) * (x: %) : % == per(r * (rep x));
10 }
```

some unique type, known as its *domain*. Domains are (in principle) run-time values, but they belong to *type categories* that are determined statically. Categories can specify properties of domains such as which operations they export, and are used to specify interfaces and inheritance hierarchies. We want to emphasize that throughout this thesis, the term "category" refers to these type categories and not to categories as in the mathematical field called category theory. 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 which 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 .
```

An example of Aldor program is presented in Figure 2.1. It defines a parametrized category `Module(R)`, representing the mathematical category of *R-Modules*. Cate-

gories specify the requirements on parameters, and state the properties of the result; domains belonging to `Module(R)` export the `*` (multiplication with scalar) operation, which returns an element of the domain. In Aldor, within a domain-valued expression, the name `%` refers to the domain name being computed. This is fixed-pointed, and can be used as a type name. A type `Rep` is defined for every domain to give a representation for `%`, while `rep` and `per` are type conversion functions (`rep:%->Rep; per:Rep->%`). In Figure2.1 the `Monomial` domain is an element of the `Module(R)` category having the dependent mapping type: `(R: Ring)-> Module(R)` that takes one parameter `R` (which is a domain satisfying the `Ring` category), and produces another *R-Module*. Static analysis can use the fact that `R` provides all the operations required by `Ring`, thus allowing static resolution of names and separate compilation of parameterized modules.

2.1.2 C++/Aldor Interoperability

Initial work in using modules with parametric polymorphism across a language boundary arose in the FRISCO project, the ESPRIT Fourth Framework project LTR 21.024. 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 the 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 first step was to define a correspondence between the two languages. Both languages 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 would have 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 generate 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.

A detailed account of this work is given elsewhere [10, 11], but the salient conclusions are that:

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

With this experience we were motivated to develop a more general solution, that could encompass more languages in a simpler way.

2.2 Parametric Polymorphism Semantics

Theoretical work on the type theory and semantics of parametric polymorphism goes back at least to the work of Girard in 1972 on the context of proof theory in logic [22].

He was the first to formulate the now well known *System F* type system that supports universal types. A little later, in 1974, Reynolds developed independently a type system with essentially the same power, and named it *polymorphic lambda calculus* [57, 58].

The ML-style of *let polymorphism* was first introduced by Milner in 1978 [37]. It constitutes a less general form of parametric polymorphism then that of *System F*, but has other advantages such as it effectively employs type-reconstruction to compute the *principal type*: the most general type possible for every expression and declaration [16]. This problem is known to be undecidable for *System F*.

Bounded quantification is a means to type-check functions that operate uniformly over all subtypes of a given type. The first language to support a type-safe bounded quantification appears to have been CLU [32](1981). Cardelli and Wegner formulated in 1985 $SystemF_{<:}$ [7] that combines polymorphism and subtyping, increasing both the expressive power of the system and its complexity. Section 2.2.1 briefly presents a subset of the $SystemF_{<:}$ type system.

Extensions of the $SystemF_{<:}$ relax the context scope rules: *F-bounded quantification* [5] allows the type variable to appear in its bound (as in $\lambda X <: \{a : Bool, b : X\}\, .t$). The Generic Java type system [47] permits mutual recursion between type variables via their upper bounds. Parametric polymorphism is also a common feature of higher-order systems with dependent types; these issues were explored early on in the context of the Russell language [18].

Parametric polymorphism is one of the mechanisms that have proved most useful in the programming language field over the last decade. It increases the flexibility, re-usability, and expressive power of the programming environment, avoids the need

for down-casting, and allows a compiler to find more programming errors. There have been many special-purpose programming languages that have supported parametric polymorphism for some time (Ada, Modula3, ML, Aldor), but it has really only been C++ which was in mainstream use. Recently, with the availability of generics in Java, one may claim that it has become a common feature of mainstream programming languages.

There are quite a few popular programming languages with support for parametric polymorphism, albeit with differing semantics. Section 2.2.2 summarizes a few, to give an idea of the range a general facility must be able to map on to. The salient conclusion (Section 2.2.2.4) is that 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 *non-qualified* type variables.

2.2.1 A Formal Introduction to Types Systems and Parametric Polymorphism

This section is intended as an introduction to type-system theory, and presents part of the Cardelli and Wegner $SystemF_{<:}$ [7]. The material presented here closely follows some of the ideas described in two well-known works: the "Type Systems" article of Luca Cardelli [6] and the Benjamin Pierce book "Types and Programming Languages" [52], and is provided to make this background section self-contained.

A type systems provides conceptual tools used in the definition of a typed-language to prove that certain types of *execution errors* will not occur during the running of a well formed program (i.e. a program that complies with the rules of the type system).

The type-rules of a programming language are specified in a way independent of the particular typechecking algorithms used for implementation. The reasons are twofold: First, this enforces a separation of concerns between the process of defining a language, which is achieved in a formal way through specifying a type system, and the implementation of that language (through a compiler). Second, it is better and easier to explain the typing aspects of a language via a type system, rather than by the algorithm a compiler uses as different compilers may use different typechecking algorithms.

2.2.1.1 Informal Nomenclature for Typing, Execution Errors, and Related Concepts

Definition 1 The *type* of a variable is the upper-bound of the range of values the variable can assume during the execution of a program. □

In practice, these types also provide *representation* and *interpretation* for data. For example, the pair of machine words `(w1, w2)` could be interpreted either as a complex number with integer coefficients or as a double word integer, and in both cases all possible values for the words `w1` and `w2` would be admitted. However, from a theoretical point of view, we wish to view types as providing bounds on a set of values without reference to their representation.

Definition 2 Languages where variables can be given non-trivial types are called *statically typed languages*. Languages that do not restrict the range of variables are called *dynamically typed languages*: they do not have types or, equivalently, have a single universal type that contains all values. □

Definition 3 A *type system* is that component of a typed language that keeps track of the types of all expressions in a program (variables included). □

Definition 4 A *trapped error* is an execution error that causes the computation to stop immediately (eg: division by zero, accessing an illegal address). An *untrapped error* is an execution error that may go unnoticed for a while and later cause arbitrary behavior (eg: improper access of a legal address, jumping to the wrong address, accessing data past the end of an array in absence of run-time bounds checks). □

Definition 5 A program fragment is *safe* if it does not cause untrapped errors to occur. Languages where all program fragments are safe are called *safe languages*. □

Dynamically typed languages may enforce *safety* by performing run-time checks. Statically typed languages may enforce *safety* by statically rejecting all programs that are potentially unsafe; they may also use a mixture of run-time and static checks. Although safety is an important property of programs, it is usually the case that statically typed languages aim to rule out large classes of trapped errors in addition to the untrapped ones.

Definition 6 *The forbidden errors* of a language are a set of the possible execution errors that (usually) include all of the untrapped errors, plus a subset of the trapped errors. □

Definition 7 A program fragment is *well behaved* if it does not cause any forbidden errors to occur. The contrary is to be *ill behaved*. □

Definition 8 A language where all of the legal programs are well behaved is called *strongly checked*. □

From the above definitions it follows that a strongly checked language has the following properties:

- no untrapped errors occur.
- none of the trapped errors designated as forbidden occur.
- other trapped errors may occur; it is the programmer's responsability to avoid those.

Statically typed languages enforce good behavior by performing compile time checks. These languages are *statically checked*. The checking process is called *typechecking*; the algorithm that performs this checking is called *typechecker*. A program passing the typechecker is said to be *well typed*, otherwise it is *ill typed*. Statically checked language (Pascal, ML) usually need to perform run-time tests to achieve safety (array bounds must in general be tested dynamically).

Several languages provide mechanism to perform sophisticated dynamic tests. An example in this sense is the Modula-3's TYPECASE construct that discriminates based on the run-time type of an object. These languages are still considered statically checked because the dynamic tests for type-equality are compatible with the algorithm used by the typechecker to determined type equality at compile time.

Dynamically typed languages can enforce good behavior (including safety) by means of detailed run-time checks that rule out all forbidden errors. The checking process in this languages is called *dynamic checking*. For example, Lisp is a dynamically typed language that is strongly checked even though it has neither static checking, nor a type system. It is also the case that most dynamically typed languages are strongly checked by necessity, otherwise programming in such a language would be quite frustrating.

Definition 9 A language for which the set of forbidden errors does not include all the untrapped errors is called *weakly checked*. □

For example the C language has many unsafe and widely used features such as pointer arithmetic and casting. Modula-3 supports unsafe features, but only in modules that are explicitly marked *unsafe*, while Pascal is unsafe when untagged variant types and function parameters are used. The choice between a safe and unsafe language is ultimately related to the tradeoff between development time and execution time. It seems that the trend is away from weak typing. This is confirmed by the appearance of mainstream programming languages like C++ and Java that alleviate some of the problems caused by weak typing in C.

We have seen that type systems are used to define the notion of well typing, which is itself a static approximation of good behavior. *Formal type systems* are the mathematical characterizations of the informal type systems, which are described in the languages manuals, and their role is to prove that the type rules of a language do not accidentally allow ill behaved programs to slip through. To prove this, one have to prove a *type soundness theorem* stating that *well-typed programs are well behaved*.

Definition 10 A type system is said to be *sound* if all the well typed programs of a statically typed language are well behaved. □

2.2.1.2 Concepts Related to Formalizing a Language

This section describes the steps involved in formalizing a type system, and consequently a programming language. The first step is to describe the syntax, and this usually reduces to describing the syntax of its *types*, which describe static knowledge

about programs, and its *terms*, which express the algorithmic behavior (statements, expressions).

The next step is to define the *scoping rules* of a language that bind a use of an identifier with its definitions. The scoping needed for the typed languages is usually *statical*, in the sense that the bindings are determined before run-time. When the binding locations are determined purely based on the syntax of the language it is said that the language uses *lexical scoping*. The lack of statical scoping is called *dynamic scoping*. Scoping is formally specified by defining the set of *free variables* of a program fragment, and this involves specifying how variables are bound to declarations.

Further on, the type rules of a language are defined. These describe *has-type* relations, of the form $M : A$, which means that the type of the term M is A. Some languages also require a *subtype-of* relation, of the form $A <: B$ between types, which specifies that the range of values defining the type A is a sub-set of the one for the type B. In many cases, it is also required to define a *type equivalence* relation, of the form $A = B$, which indicates type equivalence. Another fundamental concept that appears in type rules, and is not reflected in the language syntax is the *static typing environment*. This is used to record the types of free variables corresponding to a program fragment, and closely resembles the compiler's symbol table associated with the typechecking phase. For example, the type rule describing a has-type relation is written in full as: $\Gamma \vdash M : A$, meaning that M has type A in environment Γ.

The collection of all the type rules of a language forms its type system. Consequently, a language that has a type system is called *typed*. Finally, the last step in formalizing a language is to define its semantics as a *has-value* relation between terms and a collection of results. This is the essence of the *soundness theorem: the type*

of a term and of its result are the same. This final step bounds the type system to the semantics of the corresponding programming language. However, the fundamental notions of the type system are applicable to virtually all computing paradigms (functional, imperative), and individual type rules can often be adopted unchanged. For example, the basic type rules for functions are independent on the calling mechanism: call-by-name, call-by-value, or call-by-value-result. In what it follows we shall describe the type systems independent of semantics.

The description of a type system starts with a collection of *judgments*.

Definition 11 A typical *judgment* has the form: $\Gamma \vdash \Im$ where $\Im$ is an assertion, and the free variables of $\Im$ are declared in Γ. The relation reads Γ *entails* $\Im$. □

In the definition above Γ is a *static typing environment*, and it usually consists of an ordered list of variable-types pairs: $\phi, x_1 : A_1, ..., x_n : A_n$, where ϕ is the empty environment. The collection of variables declared in Γ is indicated by $dom(\Gamma)$, and all the free variables of $\Im$ must be declared in Γ. The most important judgment is *the typing judgment* of the form: $\Gamma \vdash M : A$ that asserts that the type of term M is A with respect to a static typing environment Γ for the free variables of M. Another common judgment simply asserts that an environment is *well-formed*, and has the form: $\Gamma \vdash \diamond$. Any given judgment is valid (e.g. $\Gamma \vdash true : Bool$) or invalid (e.g. $\Gamma \vdash true : Nat$).

Definition 12 *Type rules* assert the validity of certain judgments on the basis of other judgments that are already known to be valid. □

Each type rule is written as a number of *premise* judgments $\Gamma_i \vdash \Im_i$ above a horizontal line, with a single *conclusion* $\Gamma \vdash \Im$ below the line. If all the premises are

Figure 2.2 Three examples of type rules

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash n : Nat}\ (\text{Val } n)\ (n = 0, 1, \ldots) \qquad \frac{\Gamma \vdash M : Nat \quad \Gamma \vdash N : Nat}{\Gamma \vdash M + N : Nat}\ (\text{Val } +) \qquad \frac{}{\phi \vdash \diamond}\ (\text{Env } \phi)$$

valid, the conclusion must hold. Each value has a name, where the first word of the name is the kind of the conclusion judgment. Conditions that restrict the applicability of a rule, together with the abbreviations used within the rule are annotated next to the rule name or the premises. A collection of type rules is called a *formal type system*. They are used to carry out step by step deductions concerning the typing of a program.

Figure 2.2 shows three type rules whose conclusions are value typing judgments. The first one states that any natural number is an expression of type Nat, in any well-formed environment Γ. The second states that two expressions M and N, both of type Nat can be combined in a larger expression $M + N$, which also is of type Nat. Moreover, the environment Γ for M and N carries over to the expression $M + N$. The last rule states that the fundamental environment is well formed, with no assumptions.

Definition 13 *A type derivation* in a given type system is a tree of judgments (with leaves at the top) where each judgment is obtained from the one immediately above it via type rules of the system. *A valid judgment* is one that can be obtained as the root of a derivation where the type rules of the system are correctly applied. □

Definition 14 The process of finding a derivation, and hence a type, for a term is called *type inference*. □

Figure 2.3 Type derivation example

$$\dfrac{\dfrac{\dfrac{}{\phi \vdash \diamond}\ \text{by (Env } \phi)}{\phi \vdash 2 : Nat}\ \text{by (Val } n) \qquad \dfrac{\dfrac{}{\phi \vdash \diamond}\ \text{by (Env } \phi)}{\phi \vdash 4 : Nat}\ \text{by (Val } n)}{\phi \vdash 2 + 4 : Nat}\ \text{by (Val } +)$$

For example the derivation presented in Figure 2.3 establishes that $\phi \vdash 2+4 : Nat$ is a valid judgment. If a valid derivation with respect to a term does not exist, we say that the term is *ill typed*, or that it has a *typing error*.

2.2.1.3 System $\mathbf{F}_{<:}$

This section presents a simplified $F_{<:}$ type system with support for higher order functions, recursive types, sub-typing, and type parametrization. This is part of the system introduced by Luca Cardelli in his well-known "Type Systems" article [6]. Type parametrization is a *second order* feature, leading to a second order type system; the other mentioned features are considered to be *first order*.

The specification of the type system uses the *second-order typed lambda calculus*. The second order lambda calculus, or *polymorphic lambda calculus* as it is variously called, corresponds, via the *Curry-Howard correspondence* [23][15] to the second-order intuitionistic logic, which allows quantification not only over terms (individuals), but also over types (predicates). The main difference with respect to the *untyped* λ *calculus* is the addition of type-annotations for λ-expressions. Thus, in $\lambda x : A.M$, x is the function parameter, A is its type, and M is the body of the function. It also may express polymorphism: $\lambda X <: A.M$ indicates a program M that is parameterized by

Figure 2.4 Judgments for type systems with subtyping

$\Gamma \vdash \diamond$	Γ is a well-formed environment
$\Gamma \vdash A$	A is a well-formed type in Γ
$\Gamma \vdash A <: B$	A is a subtype of B
$\Gamma \vdash M : A$	M is a well-formed term of type A

the type X bounded to be a subtype of A. Also, terms that differ only in their bounded variables such as $\lambda x : K.x$ and $\lambda y : K.y$ are called *syntactically identical*.

Figure 2.4 presents the judgments of the type system. Figure 2.5 shows the syntax used for describing the type system of a simple language that supports bounded parametric polymorphism. In the case of untyped λ calculus, the context free syntax describes exactly the legal programs. For *typed calculi* it is the type system responsability to decide whether a term is well-typed or not. The context free syntax is still needed to describe the scoping rules of the language (i.e. the free/bounded variables). In Figure 2.5, $\lambda X <: A.M$ stands for a program M that is parameterized with the type variable X bounded to be an arbitrary subtype of A. Similarly, $\forall X <: A.B$ constructs a quantified type from a type variable X, bounded to be a subtype of A, and a type B where X may occur. This qualification can also be seen as a generalization of the *unbounded parametric polymorphism*: the term $\lambda X.M$ can be written in our system as $\lambda X <: Top.M$.

The validity of the judgments in Figure 2.4 is defined through the type rules in Figure 2.6. The rule (Env ϕ) is the only axiom of the system; it states that the empty environment is valid. The rule (Env x) is used to extend an environment Γ to a larger one that also contains the variable x of type A, provided that A is a valid type in Γ, and that there is no other variable named x in Γ. The rule (Type Arrow) constructs the functional type: given that both types A and B are well-formed in Γ, it follows

Figure 2.5 Syntax for a language with support for bounded parametric polymorphism

A, B	$::=$	types	
	X		type variable
	Top		the biggest type
	$A \to B$		function type
	$\forall X <: A.B$		bounded universally quantified type
M, N	$::=$	terms	
	x		variable
	$\lambda x : A.M$		function
	MN		application
	$\lambda X <: A.M$		bounded polymorphic abstraction
	MA		type instantiation

that the type of functions with domain in A and codomain in B is well-formed in Γ. The rule (Val x) expresses that a variable named x can be used inside a well-formed environment that contains a definition of the variable: $x : A$. The rule (Val Fun) gives the type $A \to B$ to a function, provided that the function body M has type B, and that the formal parameter has type A. The rule (Val Appl) refers to function application: if a function (M) of type $A \to B$ is applied on a parameter of type A, the result will have type B.

The rules (Env $X <:$), (Type $X <:$), and (Sub $X <:$) are similar with (Env x), and (Val x), only that they refer to a type variable bounded to be a sub-type of a give type. *Subtyping* is a common feature of object oriented languages and captures the notion of inclusion between types. It behaves much like set-inclusion, while type membership is seen as set membership. The expression $A <: B$ denotes that type A is a subtype of B, and the subsumption rule (Val Subsumption) states that in this case, a term/program/expression of type A is also of type B. Subtyping is defined in Figure 2.6 as a reflexive (Sub Refl), and transitive (Sub Trans) relation, that

Figure 2.6 Type rules

(Env ϕ)
$$\frac{}{\phi \vdash \diamond}$$

(Env x)
$$\frac{\Gamma \vdash A \quad x \notin dom(\Gamma)}{\Gamma, x : A \vdash \diamond}$$

(Env $X <:$)
$$\frac{\Gamma \vdash A \quad X \notin dom(\Gamma)}{\Gamma, X <: A \vdash \diamond}$$

(Type $X <:$)
$$\frac{\Gamma', X <: A, \Gamma'' \vdash \diamond}{\Gamma', X <: A, \Gamma'' \vdash X}$$

(Type Arrow)
$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \to B}$$

(Type Forall$<:$)
$$\frac{\Gamma, X <: A \vdash B}{\Gamma \vdash \forall X <: A.B}$$

(Val x)
$$\frac{\Gamma', x : A, \Gamma'' \vdash \diamond}{\Gamma', x : A, \Gamma'' \vdash x : A}$$

(Val Fun)
$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x : A.M : A \to B}$$

(Val Appl)
$$\frac{\Gamma \vdash M : A \to B \quad \Gamma \vdash N : A}{\Gamma \vdash MN : B}$$

(Sub Trans)
$$\frac{\Gamma \vdash A <: B \quad \Gamma \vdash B <: C}{\Gamma \vdash A <: C}$$

(Sub Refl)
$$\frac{\Gamma \vdash A}{\Gamma \vdash A <: A}$$

(Type Top)
$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash Top}$$

(Sub Top)
$$\frac{\Gamma \vdash A}{\Gamma \vdash A <: Top}$$

(Sub $X <:$)
$$\frac{\Gamma', X <: A, \Gamma'' \vdash \diamond}{\Gamma', X <: A, \Gamma'' \vdash X <: A}$$

(Sub Arrow)
$$\frac{\Gamma \vdash A' <: A \quad \Gamma \vdash B <: B'}{\Gamma \vdash A \to B <: A' \to B'}$$

(Val Subsumption)
$$\frac{\Gamma \vdash a : A \quad \Gamma \vdash A <: B}{\Gamma \vdash a : B}$$

(Sub Forall$<:$)
$$\frac{\Gamma \vdash A' <: A \quad \Gamma, X <: A' \vdash B <: B'}{\Gamma \vdash (\forall X <: A.B) <: (\forall X <: A'.B')}$$

(Val Appl2$<:$)
$$\frac{\Gamma \vdash M : \forall X <: A.B \quad \Gamma \vdash A' <: A}{\Gamma \vdash MA' : [A'/X]B}$$

(Val Fun2$<:$)
$$\frac{\Gamma, X <: A \vdash M : B}{\Gamma \vdash \lambda X <: A.M : \forall X <: A.B}$$

contains a maximal element Top (Type Top), which represents the supertype of all the well-formed types (Sub Top), or equivalently the type of all the well-formed terms. The rule (Sub Arrow) defines the functional subtyping: *co-variant* return types, and *contra-variant* argument types. Note that one may extend the type system with type rules for records (products) and variant types (union), and to define additional subtyping rules for these type constructor (which will work componentwise).

The remaining type rules in Figure 2.6 refer to the bounded universal quantifier: (Type Forall<:), (Sub Forall<:), (Val Appl2<:), (Val Fun2<:). Rule (Type Forall<:) constructs a quantified type $\forall X <: A.B$ from a type variable X bounded to be a subtype of A, and a type B where X may occur. Rule (Sub Forall<:) defines the subtyping relation for the quantified types (types in which bounded type variables might occur). Rule (Val Fun2<:) builds a type-parameterized function (a polymorphic abstraction), and finally, (Val Appl2<:) instantiates a polymorphic abstraction to a given type: $[A'/X]B$ is the substitution of A' for all the free occurrences of X in B.

The final aspect of the type system we described is the set of rules corresponding to *recursive types*. Recursive types are formally introduced with the notation $\mu X.A$ that denotes the solution of the equation $X = A$, where X may occur in A. Recursive types are important for both practical and theoretical reasons. From a practical perspective, they allow the other type constructors to create complex and useful types, among which the most common example is *List*. From a theoretical perspective, the use of recursive types allows one to write a well-typed implementation of the fixed-point combinator

$$fix_T = \lambda f : T \to T.(\lambda x : (\mu A.A \to T).f(xx))(\lambda x : (\mu A.A \to T).f(xx)),$$
$$fix_T : (T \to T) \to T,$$

and further on, to embed the whole untyped lambda-calculus (in a well-typed way) into a statically typed language with recursive types [52].

Definition 15 The *unfolding* of a recursive type $\mu X.T$ is the type obtained by replacing all the occurrences of X in the body T by the whole recursive type $[X \mapsto (\mu X.T)]T$. The *folding* of a recursive type is the reverse operation.

$unfold[\mu X.T] : \mu X.T \to [X \mapsto \mu X.T]T$

$fold[\mu X.T] \quad : [X \mapsto \mu X.T]T \to \mu X.T$

Figure 2.7 Type rules for recursive types

(Val Fold)
$$\frac{\Gamma \vdash M : [\mu X.A/X]A}{\Gamma \vdash fold_{\mu X.A} M : \mu X.A}$$

(Val Unfold)
$$\frac{\Gamma \vdash M : \mu X.A}{\Gamma \vdash unfold_{\mu X.A} M : [\mu X.A/X]A}$$

(Type Rec)
$$\frac{\Gamma, X <: Top \vdash A}{\Gamma \vdash \mu X.A}$$

(Sub Rec)
$$\frac{\Gamma \vdash \mu X.A \quad \Gamma \vdash \mu Y.B \quad \Gamma, Y <: Top, X <: Y \vdash A <: B}{\Gamma \vdash \mu X.A <: \mu Y.B}$$

These primitive operations are used in conjunction with the *iso-recursive approach* that considers a recursive type and its unfolding as different, but isomorphic types.

□

The coercions defined above do not have any run time effect, since $unfold(fold(M)) = M$ and $fold(unfold(M')) = M'$. They are usually omitted from the syntax of practical programming languages, but their existence makes the formal treatment easier. Figure 2.7 presents additional type rules that allow recursive types to be a supported feature of the system. The (Sub Rec) rule, also known as the *S-Amber* rule states that in order to show that $\mu X.A <: \mu Y.B$ it suffices to show that A is a subtype of B under the additional assumptions that X is a subtype of Y. The assumption helps when finding matching occurrences of X and Y in A and B, as long as they are in co-variant contexts.

The presence of recursive types has lead to two interesting flavors of parametric polymorphism, associated with different scope rules for type variables. The first is the *F-bounded* parametric polymorphism which corresponds to a scoping rule that makes the following legal:
$\Gamma \vdash X <: \{a : Nat, b : X\}$ or $\Gamma \vdash X <: Iterator < Integer, X >$ (Java style generics). In these cases, the scope of the binding for X includes its own upper bound. The

second flavor permits *mutual recursion* between type variables via their upper bounds, as in: $\Gamma \vdash X <: Comparable < Y >, Y <: Comparable < X >$. Both flavors are supported in Java version 1.2 (see [47] for details).

2.2.2 Parametric Polymorphism Semantics and Implementation in Several Languages

There are quite a few popular programming languages with support for parametric polymorphism, albeit with differing semantics.

2.2.2.1 Modula-3, C++, Ada

In **Modula-3** [40], generics are confined to the module level: generic procedures and types do not exist in isolation. A generic module is a template in which some of the imported interfaces are regarded as formal parameters, to be bound to actual interfaces via type instantiation. In Modula-3 there is no separate type-checking associated with generics, but instead, the implementations will expand the generics and type-check the result. This is a non-homogeneous approach: the source code is reused, but the compiled code is different for different instances.

The templates in **C++** [13, 62], much like in Modula-3, are expanded at the compile time of the client; the same template call may or may not generate a compile time error, depending on the instantiation. However, the generic parameters can be substituted by any C++ type —they are not confined to be a class type as in Modula-3.

Figure 2.8 presents a class that implements a generic stack. The `Stack` class is *templated* with a type parameter `T`, which represents the type of the stack elements,

Figure 2.8 Example of a C++ templated class (`Stack`)

```
1
2  template<T, int MaxDepth> class Stack {
3      private:
4          T*   store;
5          int  top;
6
7      public:
8          Stack() { store = new T[MaxDepth]; top = -1; }
9          ~Stack() { delete store; }
10         void print()   {
11                 for(int i=top; i>-1; i--) {
12                         /* ... */
13                         printObject(store[i]); //store[i].print();
14                         /* ... */
15                 }
16         };
17
18         void push (T t);
19         T    pop  ();
20         /* ... */
21 }
```

and with an integer value `MaxDepth`, which specifies the maximal permitted stack depth. The semantics of the C++ templates allows a uniform manipulation of objects, with no discrimination for those belonging to universally quantified types. This is because the type checking is performed only after a complete expansion of the generic types. For example, one can create an array of elements belonging to a generic type (see line 8: `new T[MaxDepth]`), and also new objects belonging to a generic type (for eg: `new T()`). The validity of the `printObject` call (line 13) depends on the instantiation of the generic type `T`, and also on the environment in which this instantiation takes place. Figure 2.9 shows a valid and an invalid instantiation for `T`: `Stack<int, 50>` is a valid type because the environment contains the `void`

Figure 2.9 The validity of templated types is context dependent

```
1
2 void printObject(int i){ /* ... */ };
3
4 int main() {
5     Stack<int, 50>  valid_stack  = new Stack<int, 50>(); // OK
6     /* ... */
7     Stack<char*, 50> invalid_stack = ...; // type-checking error
8 }
```

`printObject(int)` function, as requested by line 13 in Figure 2.8. A similar line of reasoning concludes that the type `Stack<char*, 50>` will not be found valid by the type-checker.

In **ADA** [31], a generic subprogram or package is defined by a generic declaration, containing a generic part (which may include the definition of generic formal parameters), and by a generic subprogram/package. Generic type definitions may be array, access or private type definitions. Within the specification and body of the generic program unit, the operations available on values of a generic formal type are those associated with the corresponding generic type definition, together with any given by generic formal subprograms. That is, when a template is established (instantiated), all names occurring within it must be identified in the context of the generic declaration. This restriction allows Ada generics to be independent with respect to the instantiation environment (as opposed to C++). The implementation is similar to that of C++ and Modula-3.

Figure 2.10 presents the generic program `Swap` that squares and then swaps variables of any type. The program specification is preceded by the generic formal part that consists of the `generic` keyword together with a list of generic formal (type) parameters, and functions (lines 1, 2, and 3). In order to use `Swap` it is necessary to

Figure 2.10 Ada generic `Swap` program

```
1  generic
2    type T is private;       -- Generic formal type parameter
3    with function "*" (X, Y: T) return T; -- formal operator *
4  procedure Swap (X, Y : in out T);
5  procedure Swap (X, Y : in out T) is
6    Temporal : T;
7  begin
8    Temporal := X;
9    X := Y * Y;               -- the formal operator *
10   Y := Temporal * Temporal; -- the formal operator *
11 end Swap;
12
13 -- ...
14 with Matrices;
15
16 procedure Instance_Swap is new Swap
17         (T => Float, "*" => "*");
18 procedure Instance_Swap is new Swap
19         (T => Matrices.Matrix_T, "*" => Matrices.Product);
```

create instances for the wanted type. If the same identifier is used in the instantiation, each declaration overloads the procedure (see lines 16-19 in Figure 2.10).

2.2.2.2 Java, C#

Java version 1.5 introduces a generic type mechanism inspired by the **Generic Java** (**GJ**) extension. As the Java Virtual Machine has no support for parametric polymorphism, the GJ extension [46] needs to compile away polymorphism through translation strategies. The **GJ** [47] type system is based on a combination of Hindley-Milner type inference [16], F-bounded quantification[5] and type-classes [26]. It uses a homogeneous implementation approach based on a type erasure mechanism that preserves both backward (through row types) and forward (through retrofitting) compatibility.

Figure 2.11 Example of a Java generic class (`Stack`)

```
1  public interface Printable { public void print(); }
2
3  public class Stack<T extends Printable> implements Printable {
4      private T[] store;
5      private int top;  private int maxDepth;
6
7      public Stack(T[] arr)
8      { maxDepth = (arr==null)? -1 : arr.lenght; store = arr; }
9
10     public void print()  {
11         for(int i=top; i>-1; i--) {
12             store[i].print();
13             /* ... */
14         }
15     }
16
17     void push (T t) { /* ... */ }
18     T    pop  ()    { /* ... */ }
19     /* ... */
20 }
```

The main drawback of the approach is that some operations involving type variables are forbidden: one cannot apply the `new` operator on a generic type or on an array type whose signature involves a generic type. Also, the Java objects carry at run-time only the *erased* type information, the reflective mechanisms loosing precision. Other proposed extensions for parametric polymorphism in Java (e.g. NextGen [8]) preserve the run-time information of the type variables and impose fewer restrictions than GJ, but feature a weaker compatibility with legacy code.

Figure 2.11 shows part of the Java implementation for the `Stack` generic class. There are couple of differences with respect to the C++ version, which was presented in Figure 2.8. First, the type-parameter `T` is bounded to implement the `Printable` interface: `T extends Printable` (line 3), and thus it is known to contain the `void`

`print()` method. Otherwise the typechecker would have reported an error upon verifying the correctness of the `store[i].print()` instruction (line 12). The *F-bounded* parametric polymorphism of Java allows type-checking the generic definitions independent of the instantiation environment. Second, the `Stack` constructor receives an array as parameter, while in the C++ case the array was elegantly constructed locally. This is because, in Java, certain operations such as calling constructors or static methods on generic types are forbidden. It is thus illegal to create an array of objects whose type is the generic type `T` (`new T[50];` is illegal). If a reference to a non-null object of type `T` is available, there is a workaround that uses the Java's reflective features:

Class Tclass = obj_T.getClass();

store = (T[]) java.lang.reflect.Array.newInstance(Tclass, maxDepth);

C# [71] semantics for generics are similar to those of Java. The implementation, however, is not through an *erasure* technique. Instead, the .NET 2.0 Beta Common Language Runtime (CLR) provides support for F-bounded parametric polymorphism. To implement its F-bounded quantification, the CLR uses a combination of a homogeneous approach (for reference type instantiations) and macro-expansion (for basic type instantiations).

2.2.2.3 Standard ML and Related Languages

Other programming languages provide parametric polymorphism through higher order functions. **Standard ML** (SML) and **Haskell** provide functors [33] operating on module structures as its form of parametric polymorphism. In addition, the supported *let polymorphism* [37] allows a single part of a ML program to be used with

different types in nested *let* scopes. In contrast, **Aldor** [69, 68, 70] is a functional language, with a higher order type system with dependent types and type categories. Aldor has been used in the area of Computer Algebra, where an expressive type system is required to capture the relationships among abstract mathematical objects. Parametric polymorphism is provided by type-producing functions that accept and produce types belonging to declared type categories at run time. In Aldor, type variables may be qualified by means of named category-subtyping, or by means of a list of exports. This section concentrates on **SML**, as the Aldor language was already described in Section 2.1.1.

Functional languages such as *Standard ML* (SML), *Haskell*, and *Miranda* support the Hindley-Milner type system [37] that naturally encapsulates polymorphic types and generic code. These functional languages expose some polymorphic types that are strictly more general than others, in the sense that the latter can be derived from the former via a suitable substitution (*unifying* the two type expressions will give the less general type). For example `'a list` is more general than `'bool list` since `'a` stands for any type, and unifying the two types yields `'bool list`.

There are two main properties of the Hindley-Milner type system: First, every well-typed expression is guaranteed to have a *unique principal type*, and second, the principal type can be *inferred automatically*. The expression's principal type is, intuitively, "the least general type that contains all instances of the expression". For example, in Figure 2.12, the principal type for the `map` function is `('a -> 'b) -> ('a list) -> ('b list)`, and not `('a -> 'b) -> ('c list) -> ('d list)`. The latter is more general than the former, but does not recognize that the second argument is a list of elements whose type needs to be the same as the domain of the function `f` received as first parameter, and that the return type is a list of elements whose type is the same as the co-domain of `f`.

Figure 2.12 SML example: map, find, and reverse functions

```
1  fun map f nil = nil
2        | map f (hd::tl) = f(hd) :: map f tl
3  map : ('a ->'b) -> ('a list) -> ('b list)
4
5  fun find (nil, _) = false
6        | find (hd::tl, tofind) = tofind = hd orelse find(tl, tofind
             )
7  find : 'a list * 'a -> bool
8
9  fun reverse l =
10     let fun rev(nil, y) = y
11           | rev(hd::tl, y) = rev(tl, hd::y)
12     in
13     rev(l, nil)
14 end
15 reverse: 'a list -> 'a list
```

Finally, the Hindley-Milner type system requires that every expression is properly typed, but the languages themselves (SML, Haskell) do not provide methods for providing additional type constraints. This sometimes leads to unexpected behavior of a polymorphic function that is used in relation with an unsuitable type.

2.2.2.4 Concluding Remarks

We note that programming languages with separate compilation for generic modules and dynamic binding time (Java, Aldor) usually provide support for parametric polymorphism with qualifications. (Here we use qualification as a synonym for quantification.) This allows them to statically type-check the generic code. We also note that they also usually employ a homogeneous approach to implementation. Other programming languages (C++ and Modula-3) rely on their static binding time to implement parametric polymorphism. In these cases, the type-checking has to wait

until the generic type is instantiated, thus the implementation approach is usually a non-homogeneous one.

We conclude that a mechanism to combine modules in different programming languages must be able to accommodate both *compile-time* and *run-time* instantiation of modules and both *qualified* and *unqualified* type variables. Our approach has been to design a qualification-based generic type model to accommodate programming languages that support it, and to enforce these qualifications in our mappings for the programming language which do not. Our model also allows generic types to be unqualified, in which case any GIDL type is a valid candidate for instantiation.

2.3 Mainstream Software Component Architectures

This thesis investigates a way in which software component architectures (SCAs) can be extended with the parametric polymorphism feature. Among the mainstream software component architectures today, besides un-qualified, one can list CORBA, JNI, Microsoft's DCOM, and the more recent .NET architecture.

Among these architectures, only .NET effectively supports parameterized components, but in a homogeneous environment: the semantics of the generic model is defined by the intermediate language (MSIL) to which all the supported languages compile. A generic mechanism adapted from the one described in Chapter 4 can be employed in a straightforward way to add genericity to DCOM, and JNI component architectures, as discussed in Section 4.5, while preserving the backward compatibility with non-generic applications written for the underlying SCA. This is a direct consequence of the type-erasure mechanism that implements our generic model.

This section presents the structure of these architectures as follows: Section 2.3.1 introduces the CORBA architecture and its interface specification language, which will be the study case for our generic extension mechanism. Section 2.3.2 reviews the Microsoft component technology (COM, DCOM, .NET), and finds that the DCOM architecture is in large similar to CORBA: for example it also uses a specification language to describe the component interface. Finally, Section 2.3.3 briefly describes the Java Native Interface (JNI), which allows Java code to call *native* methods and the reverse.

2.3.1 Common Object Requests Broker Architecture

The *Object Management Group* (*OMG*) is a non-profit organization that promotes the use of component technology in heterogeneous, distributed computing systems. OMG pursues this goal through developing standards which allow the distributed object oriented applications to be portable and to interoperate.

The Common Object Request Broker (*CORBA*) [50] is an OMG open standard, which defines an implementation independent architecture for building and seamlessly interconnecting multiple systems involving distributed objects, in a way transparent for the user. In practice, CORBA applications may have some vendor dependency.

CORBA applications are composed of objects, individual units of running software that combine functionality and data. Their design is based on *the OMG Object Model*. The OMG Object Model defines common object semantics for specifying the externally visible characteristics of objects in a standard and implementation-independent way. In this model clients request services from objects (which will also be called servers) through a well-defined interface. This interface is specified in *OMG IDL* (*Interface Definition Language*) [49].

This allows the framework to be platform and language independent, in the sense that the client interfaces (to the objects), and the server implementations (of these object interfaces) can be specified in any programming language. A client accesses an object by issuing a request to the object. The request is an event, and it carries information including an operation, the object reference of the service provider, and actual parameters (if any). The object reference is an object name that defines an object reliably.

The remaining of this section briefly presents the IDL language (Section 2.3.1.1), and succinctly describes the components of the CORBA architecture (Section 2.3.1.2).

2.3.1.1 Interface Definition Language (IDL)

In order to achieve interoperability and portability, the CORBA standard requires the use of the IDL to describe the interfaces of remote objects. The interface is the syntax form of the promise the server object makes to the clients invoking it. This fixes the operations that will be performed and the parameters (input and output) for each. The IDL interface definition is independent of the programming language used for implementation, OMG having standardized mappings to popular programming languages like: C, C++, Java, COBOL, Smalltalk, Ada, Lisp, Python[49].

IDL is a declarative language whose syntax was constructed from a subset of C++ and Pascal instructions. It defines basic types (`short`, `byte`, `float`, `double`, `string`, etc.), structured types (`struct`, `sequence`, `array`, `module`), and provide signatures for `interface` types, fully specifying each operation's parameters. Multiple inheritance among interfaces is supported, but recently adopted features like function/operator overloading, and parametric polymorphism are not. However, since it targets dis-

tributed objects, IDL forces the user to specify additional information with respect to the object interface, such as which method arguments are *input-only*, *output-only*, or *two-way* data transfers. This is achieved using additional keywords on method arguments, before their type specifications: `in`, `out`, and `inout`.

The remainder of this subsection presents an example of and IDL specification together with a C++ server implementation, and a Java client that accesses the server functionality. The reasons are twofold: First, we want to convey to the reader the "look and feel" of creating a multi-language application in an existing SCA. We are using CORBA in this case but the process is quite similar for DCOM, while for .NET is even simpler (as it does not support "remote" objects). Second, and most importantly, we want the reader to "feel" the difference between a SCA and a foreign function interface. The latter usually leads to difficult and complex applications, as the programming style specific to a given language is disrupted by numerous calls to "special" kernel functions. It also leads to a rather un-safe program, as very little from what is "foreign" can be statically type-checked. We hope that our example shows that the SCAs are relatively *easy to use*, and *safe*, as they usually enforce statical type-checking of the foreign calls. Moreover, they are a better solution to the multi-language interoperability problem: in order to accommodate n languages, a usual SCA (one that uses an IL) would require $O(n)$ translators, while a solution based on foreign function interfaces would require $O(n^2)$.

Figure 2.13 shows an IDL interface for a simplified bank account server. According to the specification, a `BankServer` object has three methods: one to verify a PIN number against an account, one to get specifics about an account, and one to process a transaction against an account.

Figure 2.13 A simple IDL specification for a bank server application

```
1
2  module Examples {
3
4    interface Transaction {
5        // ...
6    }
7
8    interface BankServer {
9
10       boolean verifyPIN(in long acctNo, in long pin);
11
12       void   getAcctSpecifics(in long acctNo, in string
             customerName, out double balance, out boolean isChecking);
13
14       boolean processTransaction(in Transaction t, in long acctNo);
15   }
16
17 }
```

In the `BankServer` interface, the two arguments to the `verifyPIN` method are declared as `in` parameters, since they are only used as input to the method and don't need to be read back when the method returns. The `getAcctSpecifics` method has two `in` parameters and two `out` parameters. The two `out` arguments are read back from the server when the method returns as output values. An `inout` argument is both fed to the method as an input parameter, and read back when the method returns as an output value [20]. When the IDL interface is compiled into a client stub and a server skeleton, the input/output specifiers on method arguments are used to generate the code to marshal and unmarshal the method arguments correctly.

Figure 2.14 presents part of the C++ server that implements the `BankServer` interface. On line 1, our implementation (`BankServer_Impl`) inherits from the IDL skeleton class `POA_Example::BankServer` that has been automatically generated when the

Figure 2.14 Part of the C++ implementation of the `BankServer`

```
1  class BankServer_Impl : public virtual POA_Example::BankServer,
2            public virtual ::PortableServer::RefCountServantBase {
3  private:
4    int* pin_arr;  int* balance_arr; int* account_nr_arr;
5
6    int findAccountIndex(int acctNo) { /* ... */ }
7
8  public:
9    BankServer_Impl(int* pin, int* bal, int* acctNo) { /* ... */ }
10
11   virtual CORBA::Boolean verifyPIN(int acctNo, int pin)
12       throw(CORBA::SystemException) {
13       int index = findAccountIndex(acctNo);
14       return (pin == pin_arr[index])? 1 : 0;
15    }
16    // ... further implementation
17 };
```

IDL specification in Figure 2.13 was compiled. Thus, it is "linked" to the CORBA framework. Note that the programmer's job is fairly simple, the code being very close to the one written for a single-space, C++ implementation of the bank server.

Figure 2.15 shows a Java client that uses the functionality of the C++ bank server implemented in Figure 2.14. The client assumes that the `BankServerObj.ior` file contains a a string representation of the bank server object (line 6), together with type information and whatever is required to access the remote reference (machine address and port number). This string is parsed and a remote object interfacing the server is created on line 12. On line 16 the object is "coerced" to its proper type (`serv` is of type `BankServer`). Finally, the server `serv` can be used as if it is local and it is implemented in Java. Lines 22 and 23 present the execution of two remote operation. Our bank server is required to verify a pin number against an account, and if proved valid, to perform a transaction.

Figure 2.15 A simple Java client using the bank server

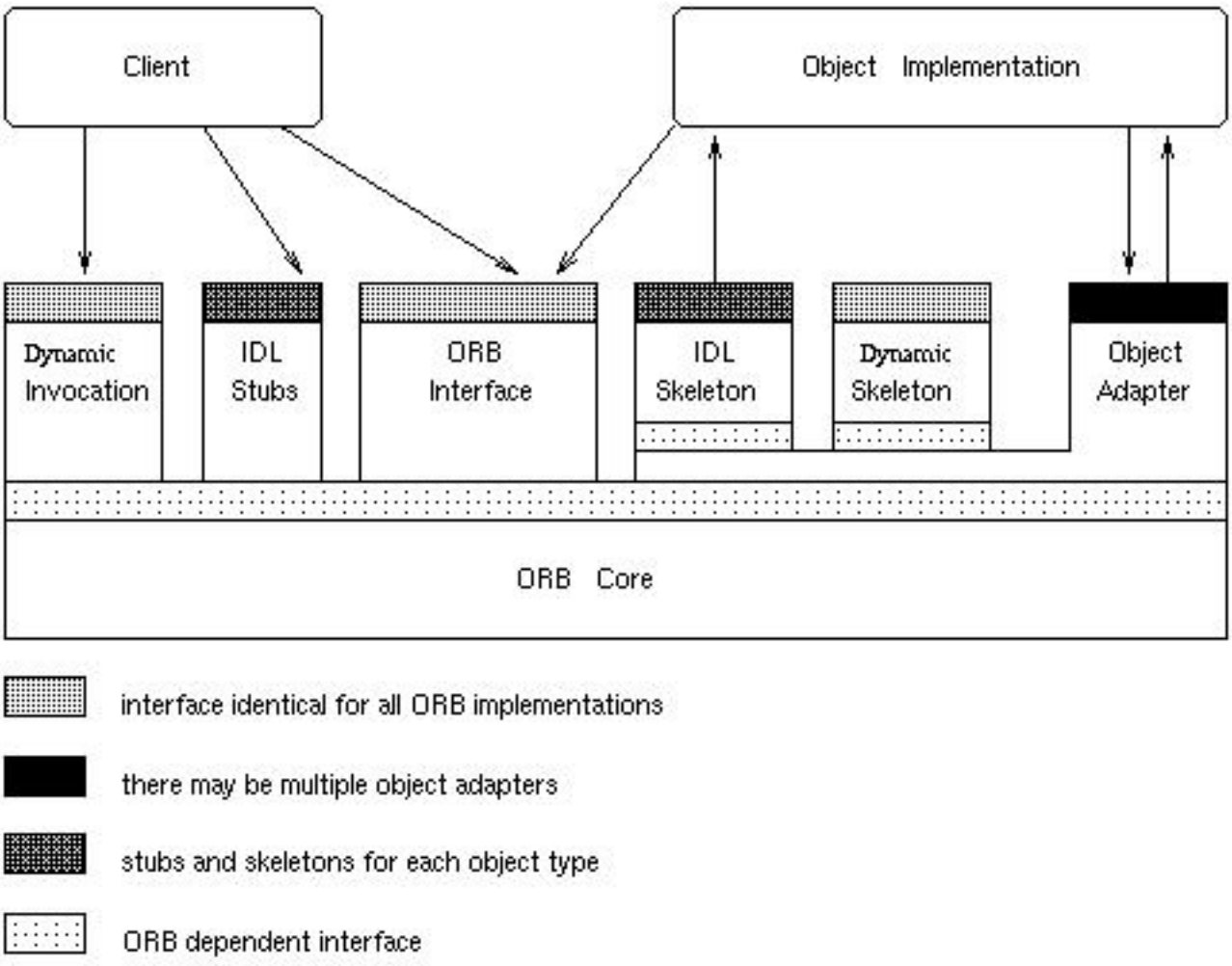
```
1  static int run(org.omg.CORBA.ORB orb)
2        throws org.omg.CORBA.UserException {
3
4    org.omg.CORBA.Object obj = null;
5    try {
6      String refFile = "BankServerObj.ior";
7
8      java.io.BufferedReader in =
9        new java.io.BufferedReader(new java.io.FileReader(refFile));
10     String ref = in.readLine();
11
12     obj = orb.string_to_object(ref);
13
14   }catch(java.io.IOException ex) { return -1; }
15
16   Examples.BankServer serv = Examples.BankServerHelper.narrow(obj);
17   Example.Transaction trans = ...; //create a transaction object
18
19   int acctNo = 1211356256;
20   int pin   = 2145;
21
22   if(serv.verifyPIN(acctNo, pin))
23     serv.processTransaction(trans, acctNo);
24 }
```

2.3.1.2 Overview of CORBA Architectural Components

This section follows the "A Brief Tutorial in CORBA" work of Kate Keahey [28]. Figure 2.16 shows the main components of the ORB architecture and their interconnections.

The central component of CORBA is the *Object Request Broker* (*ORB*). The ORB is the middleware that establishes the client-server relationships between objects. It encompasses all of the communication infrastructure needed to identify and locate objects, handle connection management and deliver data. Using an ORB, a client

Figure 2.16 Main components of the CORBA architecture



can transparently invoke a method on a server object, which can be on the same machine or across a network. The ORB intercepts the call and is responsible for finding an object that can implement the request, pass it the parameters, invoke its method, and return the results.

It is important to emphasize that both the client and the server of CORBA objects use an ORB to talk to each other (they both have an object manager associated with

them), and this leads to the fact that any agent in a CORBA system may act as both a client and a server of remote objects. In general, the ORB is not required to be a single component; it is simply defined by its interfaces (see *ORB interface* in Figure 2.16). The *ORB Core* is the most important part of the ORB as it handles the communication of requests.

Given an IDL specification, the IDL compiler will generate IDL stub/skeleton code (not presented in Figure 2.16). The stub will act as an interface for the *remote object* while the skeleton will provide the implementation. To perform a remote operation, the client transfers a request to the ORB Core via the *IDL stub* or through the *Dynamic Invocation Interface* (*DII*). The *IDL stub* represents the mapping between the implementation language of the client and the ORB core. It follows that the client can be written in any language as long as the implementation of the ORB supports this mapping. The ORB Core then transfers the request to the object implementation which receives the request as an up-call through either an *IDL skeleton*, or a *dynamic skeleton* [28].

The *Object Adapter* (*OA*) is the architectural-component responsible for the communication between the object implementation and the ORB core. It handles services such as generation and interpretation of object references, method invocation, security of interactions, object and implementation activation and deactivation, mapping references corresponding to object implementations and registration of implementations. `POA` is one of the CORBA standard object adaptors (see Figure 2.14, line 1).

There are two ways to specify the object interfaces: through an IDL specification, or by directly adding them to the *Interface Repository (IR)* – a database which

provides persistent storage of object interface definitions. *The Dynamic Invocation Interface* (*DII*) enriches the CORBA object with reflective features: it allows the client to specify requests to objects whose definition and interface are unknown at the client's compile time. To use *DII*, the client composes a request (in a standard way to all ORBs) that contains the object reference, the name of the operation to be invoked, and a list of parameters. The object services are retrieved from the IR and the proper operation is invoked.

2.3.2 Microsoft Component Technology

This section briefly presents Microsoft's approach to structuring the software into components that can be combined together to create single process or distributed applications.

Section 2.3.2.1 and Section 2.3.2.2 introduce the *Component Object Model* (*COM*) and the *Distributed COM* (*DCOM*) respectivelly. The two encompass a set of Microsoft concepts and program interfaces in which client program objects can request services from server program objects (written in various programming languages) on other computers in a network. From a high level perspective, their design is similar to CORBA's, in the sense that every COM object is associated with an interface (written in an intermediate language like IDL), and the only way to access a COM object is through its interface (which, once published, is immutable). It is important to underscore that *parametric plymorphism* is not a supported feature of the COM interface definition language, and thus, *components are not allowed to be type-parameterized*.

The newly defined *.NET* framework is described succinctly in Section 2.3.2.3. The approach taken was to define a *Common Language Runtime* (*CLR* for short) that

provides a common type system, and a common intermediate language (*MSIL*) that facilitates interoperability between programs written in the supported programming languages [29]. Starting with version 2.0 Beta, .NET's MSIL provides support for bounded parametric polymorphism (and so does *C #*). The extension has been proposed by Kennedy and Syme in [29]. However, the semantics is fixed by the definition of the *MSIL*, and all the "managed" languages must comply to it.

2.3.2.1 Component Object Model (COM)

Component Object Model (*COM*) is a Microsoft component technology that builds on the notion of interface and enables software components to be developed and used from any COM aware languages and environments. It encompasses most of the languages supported by Microsoft Visual Studio: C/C++, Visual Basic, .NET languages, etc.

COM [14, 65] defines and implements a common communication protocol for objects originating from various programming environments to interact with each other. The communication is achieved via abstract interfaces. This allows the implementation details to be hidden from the client, as there is no formal way to gain access to the internal implementation of the client code.

Each object satisfies a number of public interfaces. A client can query an object for the interfaces it supports, and can invoke the functionality described in these interfaces. COM infrastructure also supports some other features such as location transparency, and life time management.

Much like in the CORBA case, each interface is defined in an abstract interface definition language (IDL). All programming languages that are able to interpret the meaning of those abstract definitions (mostly the languages fostered by Microsoft) and

Figure 2.17 Example of a COM class (`MyObject`)

```
1
2  /** IDL Specification of the IMath interface **/
3  Interface IMath : IUnknown
4      Factorial
5
6  /** C++ implementation of the MyObject class **/
7  /** MyObject satisfies the IMath interface **/
8  class MyObject : public IMath
9  {
10     public:
11         // IUnknow functions
12         // ....
13         HRESULT Factorial(long val, long* retval);
14 };
```

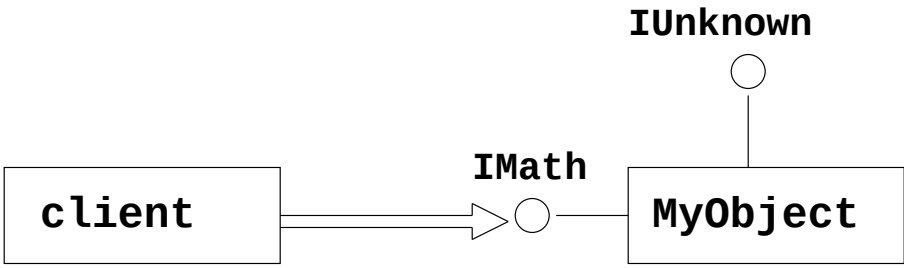
for which there is a language mapping are able to fully utilize COM. This mechanism makes COM language independent. Interfaces are used as contracts between objects and their clients, therefore they are considered immutable after publishing.

All COM interfaces are derived from the `IUnknown` base interface. `IUnknown` contains the functionality needed by clients to manage and use COM objects: it provides methods for dynamic binding, life time management, etc. A COM class (*coclass*) is a named implementation of a an interface, and it can be represented by any concrete data type that satisfies one or more interfaces (i.e. it is not necessarily a class in the object oriented sense). Figure 2.17 presents the definition of the `IMath` interface and part of the implementation of the `MyObject` C++ class that satisfies the `IMath` interface. One can notice that Microsoft definition language is weaker than OMG's IDL (see the specification of the Factorial function).

Client code is able to obtain a reference to a COM object by invoking proper methods on the COM runtime and by providing proper arguments. Figure 2.18

shows a client accessing the `MyObject` object through its `IMath` interface.

Figure 2.18 Client holding a reference to an object



To use a COM object, one has to obtain a reference to an instance of a coclass. This is similar to CORBA's "object reference". Now, the client owns the object and is free to request services from it by using some of the custom interfaces supported by the object (for eg. the `IMath` interface in Figure 2.18). Every COM object supports resource management and once all the client references to it have been released the object will clean up and destroy itself freeing the memory.

2.3.2.2 Distributed Component Object Model (DCOM)

DCOM [14] is an extension to the COM based component technology. It allows COM components and objects to be accessed remotely over the LAN, WAN or even over the Internet. When client and component reside on different machines, DCOM simply replaces the local interprocess communication with a network protocol (see Figure 2.19). DCOM infrastructure takes care of marshalling all the data and communication requests over the network without requiring any client interaction. In essence the DCOM provides an object-oriented RPC mechanism and object lifetime management infrastructure on top of COM.

Figure 2.20 shows the overall DCOM architecture: The COM run-time provides object-oriented services to clients and components and uses RPC and the security

Figure 2.19 Client using a remote object via DCOM

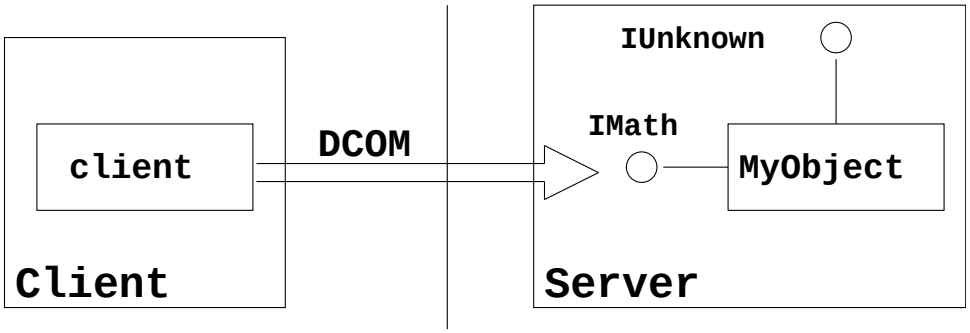
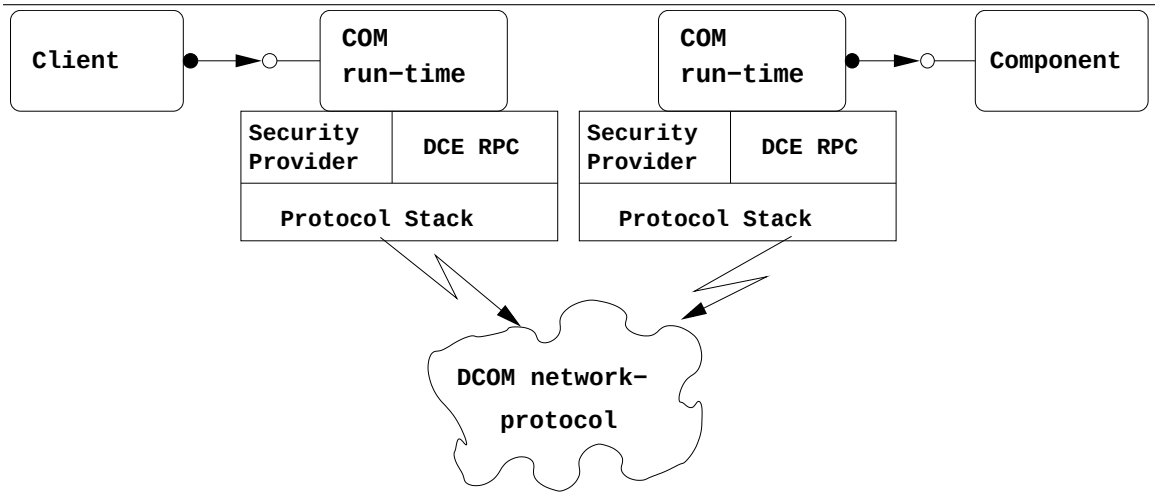


Figure 2.20 Main components of the DCOM architecture



provider to generate standard network packets that conform to the DCOM wire-protocol standard.

One may notice that, from a high-level perspective, the design of DCOM and CORBA is quite similar: "remote" objects are accessed based on some interfaces they implement, and these interfaces are described through a specification language. What differs are the network and the remote invocation protocol, the memory dealocation discipline: reference counting or manual for CORBA vs. distributed garbage collection for DCOM, etc. The programmer has to provide the component specification, the "server" implementation, and the "client" program, which combines different

components to form an application. The rest (linking code) is generated/provided automatically by the CORBA/DCOM architecture.

2.3.2.3 .NET Framework

The approach taken by .NET was that rather than having a framework that standardizes the ways in which components can be defined and use in/from different programming languages, it is perhaps easier and more natural to encapsulate these mechanisms in the implementation of the supported languages by means of a common virtual machine. More precisely they designed a *Common Intermediate Language* (*MSIL* for short) to which all the supported programming languages compile to, and a *Common Language Runtime* (*CLR*), that will dictate the run-time behavior of the MSIL code. This makes the .NET component model largely implicit: developers use standard language syntax to create, export, and use components.

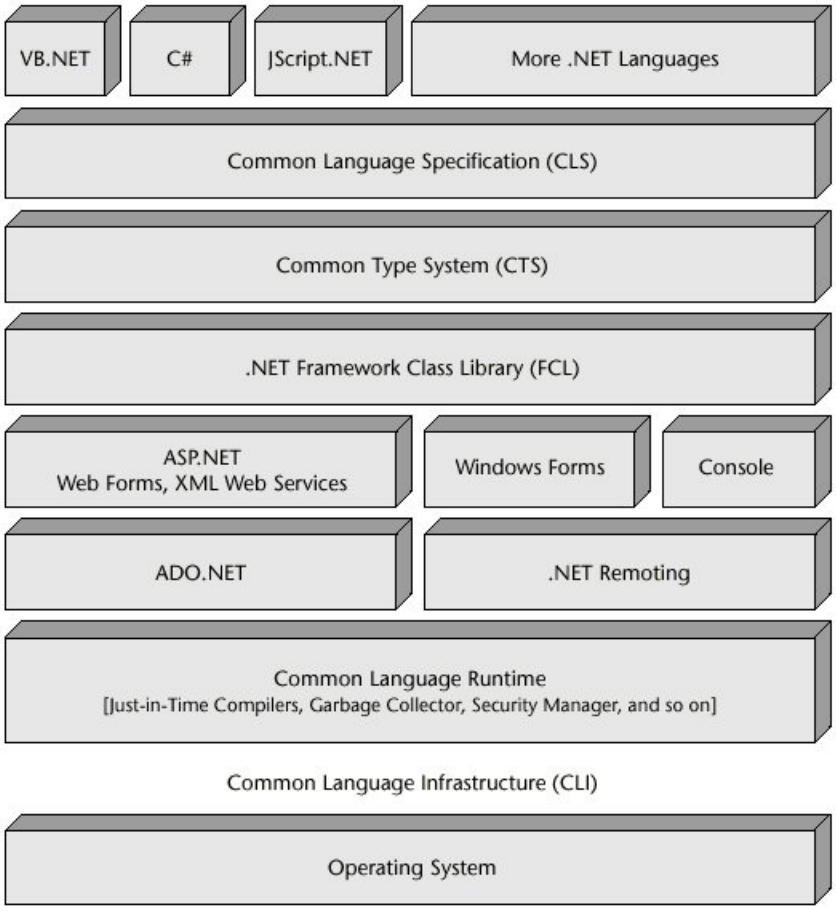
As presented in [34], .NET is tiered, modular, and hierarchal. The architectural layout of the .NET Framework is illustrated in Figure 2.21. Each tier of the .NET Framework is a layer of abstraction:

.NET languages They are the top tier and the most abstracted level.

The Common Language Runtime CLR is the bottom tier, the least abstracted, and closest to the native environment. This is because it is the one who works closely with the operating system to manage .NET applications.

Common Language Specification CLS is a set of specifications or guidelines defining a .NET language. Shared specifications promote language interoperability. For example, CLS defines the common types of managed languages, which is a

Figure 2.21 Main components of the .NET framewok



subset of the Common Type System (CTS). This removes the issue of marshaling, a major impediment when working between two languages.

Common Type System CTS is a catalog of .NET types such as System.Int32, System.Decimal, System.Boolean, etc. Developers are not required to use these types directly, as they are the underlying objects of the specific data types provided in each managed language.

.NET Framework Class Library FCL is a set of managed classes that provide access to system services (file input/output, sockets, database access, remoting, XML). All the .NET languages rely on the same managed classes for the same

services.

ASP.NET, Windows Forms, Console *ASP.NET* is used to create dynamic Web applications and is the successor to ASP. *Windows Forms* is primarily a code generator, generating managed classes for graphical user interface elements (forms, buttons, text boxes, menus). *Console applications* are useful for logging, instrumentation, and other text-based activities.

ADO.NET, .NET Remoting *ADO.NET* offers managed providers for Microsoft SQL and OLE DB databases. It accentuates disconnected data manipulation, integrates open standards, and is perfected for Web application development. *.NET Remoting* provides a way of defining and using remote objects in .NET. It allows the developer to set the specification of the remoting interface (transmission protocol, data protocol, data port, etc).

Starting with version 2.0 Beta, .NET's MSIL provides support for parametric polymorphism (and so does *C#*). The extension has been proposed by Kennedy and Syme in [29]. However, .NET framework implements a homogeneous environment in which all the "managed" languages implements the same semantics for parametric polymorphism: the one defined in the definition of the MSIL, and moreover shares the same implementation (the one of the CLR). More than this, it is not clear to us if the generics semantics are unified across the managed (eg. C#) and the unmanaged (eg. C++) .NET's supported languages: what would happen if one is trying to instantiate a *C sharp* parameterized class making use of bounded type parameters on some invalid types from within *C++*?

Our approach for endowing software component architectures with support for parametric polymorphism, described in Chapter 4, is higher level than the one in

.NET, in the sense that it does not require a homogeneous multi-language environment (a common intermediate language on which all the supported languages must compile to), and "unifies" different semantics and implementations of parametric polymorphism across language boundaries.

2.3.3 Java Native Interface (JNI)

Java Native Interface [63] (JNI) is a native programming interface that allows Java code running inside a JVM to interoperate with applications and libraries written in C, C++ and assembly code.

JNI imposes no restriction on the implementation of the underlying JVM, a native application should work with all JVMs supporting JNI. A Java class that contains some *native* methods, when compiled under JNI (`javah` utility), will produce a C/C++ stub header file, containing the signature of its native methods. The C/C++ developer implements the functions, and links them to a shared library. A static initializer of a Java class can load the created shared library (via the `System.loadLibrary` method), and then the native methods may be called. In the other direction, a C/C++ application can call Java code through a reflective-like invocation to JVM.

The JNI defines mapping types for basic types (`jint, jfloat, ...`), reference types (`jobect`), plus its specializations for array types, `Class`, `String`, and `Throwable` (`jarray`, `jintArray`, *etc*, `jclass`, `jstring`, `jthrowable`). One may observe that all the Java reference types (eg: `Integer`, `List<T>`, etc.) are erased to the `jobject` JNI type. It follows that the type-correctness of the invocation (in either direction) is the programmer's responsibility, and an incorrect invocation may result in arbitrary undefined behavior (on the native side) or in a JVM exception (on the Java side). In this regard, we would say that JNI is more than a foreign function interface, since it automatically generates linking code for the native methods, but less than a software component architecture as its type vocabulary is quite reduced, and type-correctness is not enforced.

Figure 2.22 Example of defining and using native methods in Java

```
1  import java.util.*;
2  public class HelloGenerics<T extends Iterable> {
3      public native List<T>      foo1(T[]  arr1);
4      public native List<List<T>> foo2(T[][] arr2);
5      public native void         displayHelloWorld();
6
7      static {
8          System.loadLibrary("hellogenerics");
9      }
10     public static void main(String[] args) {
11         new HelloGenerics<List<Integer>>().displayHelloWorld();
12     }
13 }
```

One can also observe that, as the JVM features no support for parametric polymorphism (it was "statically compiled away") the *JNI does not support parameterized components*. The example that concludes this section will make this trivial to see.

Figure 2.22 presents the definition of the `HelloGenerics` parameterized Java class. The type parameter T is declared to implement the `Iterable` interface. The `HelloGenerics` class declares three native methods: `displayHelloWorld`, `foo1`, and `foo2`. It also declares a static initializer that loads the library (named `hellogenerics`) that contains the implementation of the three native methods. The main method instantiates the class and calls the `displayHelloWorld` native method.

Figure 2.23 presents the C++ header file obtained by compiling the class file corresponding to `HelloGenerics` with the *javah* tool. It is easy to see that the `List<T>` and `List<List<T>>` types that appear in the defintion of the `foo1` and `foo2` methods have been erased to the `jobject` JNI type. Similarly, `T[]` and `T[][]` have been erased to the `jobjectArray` JNI type. More than this, the signatures of `foo1` and `foo2` appearing in the comments associated with these functions (lines 4 and 12) are

Figure 2.23 The generated C++ header file for the native methods declared in Figure 2.22

```
1  /*
2   * Class:    HelloGenerics
3   * Method:   foo1
4   * Signature: ([Ljava/lang/Iterable;)Ljava/util/List;
5   */
6  JNIEXPORT jobject JNICALL Java_HelloGenerics_foo1
7    (JNIEnv *, jobject, jobjectArray);
8
9  /*
10  * Class:    HelloGenerics
11  * Method:   foo2
12  * Signature: ([[Ljava/lang/Iterable;)Ljava/util/List;
13  */
14 JNIEXPORT jobject JNICALL Java_HelloGenerics_foo2
15   (JNIEnv *, jobject, jobjectArray);
16
17 /*
18  * Class:    HelloGenerics
19  * Method:   displayHelloWorld
20  * Signature: ()V
21  */
22 JNIEXPORT void JNICALL Java_HelloGenerics_displayHelloWorld
23   (JNIEnv *, jobject);
```

generic type erased. According to them, the type of the return is the same for both methods. This shows that JNI cannot support parameterized components because the types handled by the JVM are erased of any generic information.

Finally, if the C implementation corresponding to the `displayHelloWorld` native method is `printf("Hello world!")`, and one runs the `HelloGenerics` Java application, `Hello world!` will be printed on the screen. This concludes our JNI example, and this section.

2.4 Thread Level Speculation

One of the main technique of increasing the performance of a single computation task, besides reducing the memory latency and increasing the clock speed, has been the attempt to exploit the application's inherent parallelism. For scientific applications, static (traditional) parallelization techniques, which usually involve data/control dependence analysis together with various code transformations to eliminate the loop carried dependences, have proved to be a success. Non-numeric applications, however, tend to use irregular data structures and complex control flow, factors that in most cases yield quite a "pessimistic" dependence analysis result. Code regions may reveal a rich-level of (hidden) parallelism, but the traditional techniques will usually fail to guarantee a *safe* parallelization.

Thread level speculation (TLS) architectures are software/hardware mechanisms that resolve at run-time the unknown dependences that would otherwise prevent parallelism from being extracted. This *speculative execution* of parallel threads has proved to be a promising direction in achieving the awaited breakthrough for non-scientific applications.

In a TLS framework, threads execute out of order, and use software/hardware structures, referred as *speculative storage*, to record the necessary information to track the inter-threads dependences and to revert to a *safe* point and restart the computation upon the occurrence of a *dependence violation* (*rollback recovery*). To guarantee correct execution, threads merge their changes into the *global non-speculative storage* only when the speculation is verified, which is only when it is determined that the locations it read-from and wrote-to do not generate a dependence-violation.

One of the main shortcomings of TLS is that it features high inter-thread communication costs. The emergence of chip-multiprocessors (CMP) has alleviated this problem to some degree. CMPs contain multiple tightly-coupled processor cores on a single chip, which significantly reduce the costs of interprocessor communication. Their emergence has come about as the cost-benefit ratio of instruction-level parallelism offered by superscalar VLIW processors has grown [48]. Even though commercial CMPs currently exist in the market [64], the cache coherency mechanism needed for speculation is not yet present.

TLS can be applied at the loop and method/function levels. At the loop level, speculative threads concurrently execute iterations of a loop out of sequential order even when these *may* contain a true dependence. The thread assigned to the lowest numbered iteration from all is referred to as the *master* thread since it encapsulates both the correct sequential state and control-flow. It is the job of the speculative cache coherency mechanisms to detect the occurrence of inter-thread data dependencies and initiate a rollback. In servicing a rollback the speculative state needs to be cleared and the threads affected by the violation are restarted to carry out the cancelled iterations. Method-level speculation overlaps the execution of a called method with the code downstream from the call-site. The region following the call is executed speculatively while the main thread executes the called method. In general, the downstream speculative region is quite small since data dependencies will occur between the parameters or return value of the two code segments. However, the length of a speculative region can be expanded through the use of value prediction. Simple, and efficient two-value and stride predictors can be applied to free up some possible dependencies with good results [60].

The motivation of our study in TLS technologies originates in the area of automatic library translation across a multi-language, distributed system. In such an environment, each component of an application is separately compiled, and is locally optimized, in the absence of any information regarding the other components structures. The consequence is that conservative, traditional compiler optimizations (inlining, tower type, parallelization) cannot be performed aggressively. Furthermore, existent generic library design is usually tributary to the assumption that the application operates in a single language/process space. This assumption no longer holds when, using a "black box" type translation strategy (see section 4.7), we make them available in a heterogeneous environment, greatly impacting the application's performance. For example, the running time of a client performing repeated calls to light functions of some remote component will be dominated by the network overhead.

In the light of the above observations, Chapter 5 proposes two distributed TLS models in the attempt to reduce the communication and dispatch overhead inherent to distributed applications. Section 2.4.2 introduces a high-level all software-based TLS framework, developed in collaboration with Jason Selby, Mark Giesbrecht and Stephen Watt that served as the backbone for the two distributed TLS models. Since this is still on-going research we shall present it at a high-level.

There is an extensive bibliography related to TLS. Section 2.4.1 summarizes a few contributions that we have found most useful and have inspired us in the design of our own TLS framework.

2.4.1 Related Work in Thread Level Speculation

The TLS related work reviewed in this section covers a variety of topics such as speculative architecture proposals, thread partitioning mechanisms, and speculative

related optimizations. In our oppinion, a good TLS framework needs to combine elements from all these topics in order to obtain good optimization speed-up.

The paper "Reference Idempotency Analysis" by Kim et al. [30] formally describes the structure of the software and the execution model for two architectures: hardware only speculation execution (HOSE), and compiler assisted speculative execution (CASE), and proves the correctness of the two. In addition it discovers a new program property called "reference idempotency" that allows to reduce the considerable performance loss caused by speculative storage overflow. These kind of references need not be kept in the speculative storage, thus reducing the demand for speculation. The paper also investigates the necessary and sufficient conditions for reference idempotency with respect to the CASE model.

Rundberg and Stenstrom in [59] propose an all software approach to design a thread-level data dependence speculation system targeting multiprocessor architectures. Loads and stores that can produce data dependence violations at run time are wrapped inside checking codes to ensure the correctness of the execution. Speculation is integrated at a low level, as it involves only memory references and basic types. The paper claims a speculative read/write operation overhead less than ten instructions. Our TLS approach, described in the next section, although inspired from this model, is higher-level, in the sense that we also provide specialized speculative storage for containers, objects, etc. Our approach is highly adaptive: the type of the speculative storage support, and the parameters of the speculative execution are chosen in accordance with the code properties and the available profiling information.

The paper "Master/Slave Speculative Parallelization" by Zilles and Sohi [73] proposes a master/slave speculative parallelization model (MSSP). In MSSP, one pro-

cessor (the master) executes an approximate version of the program ("distilled program"). In order to check its results, the master spawns, in program order, slave threads that execute the original program. There are no requirements on the "distilled" program necessary to ensure a correct execution, however, in order to get good speed-up, the "distilled" program should have a high prediction accuracy and be fast. To accomplish this, control flow transformations (cold path elimination), value-based (constant substitution for invariant values), and dependence-based (ignore may-aliases for load-store pairs that rarely alias) approximations, optimizations of writes with distant first uses may be applied. This work has inspired our second distributed TLS model, described in Section 5.2.3.

Two works by Chen and Olukotun [9], and Kazi and Lilja [27] target speculative architectures for the Java environment. The first article finds the Java virtual machine to be an effective environment for exploiting method-level parallelism and investigates how method-level speculation can speed up single-threaded, general purpose Java programs. Specifically, if a method is marked as speculative then the current processor executes the method, and a forked speculative task executes in parallel starting from the method return point (continuation). The second article proposes a speculative parallelization model (JavaSpMT) that combines control speculation with run-time dependence checking. Their thread model has three stages: the continuation stage (compute and forward recurrence variables, and forks and initiates a successor thread), the target store address generation stage (TSAG stage – computes the addresses of the write operations upon which successor threads may be data dependent), and the computation stage. In their model, new threads are created only from the beginning of the speculative thread, and threads are synchronized with respect to the TSAG stage.

Bhowmik and Franklin, in [2], present a compiler framework for partitioning a sequential program into multiple threads for parallel execution in an speculative multithreading (SpMT) system. The proposed framework supports a variety of threads: speculative, non-speculative, loop centric, and out of order thread spawning. For the partitioning process, the compiler uses profiling, intra-procedural pointer analysis, data dependent and control dependent informations. The thread model supports spawning a thread from anywhere in the current thread: it may wait for example for a dependence to get resolved, before spawning another thread.

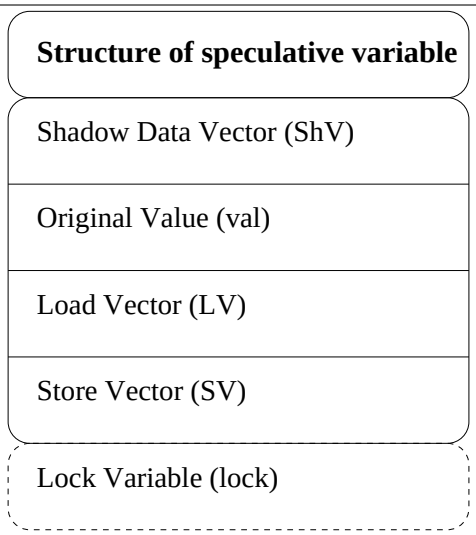
Finally, Zhai et al. observe in [72] that program performance under thread-level speculation models is severely limited due to the stalls required to forward the scalar values (that would otherwise cause frequent data dependence violations) between threads. It furtherer presents and evaluates data flow algorithms for three increasingly aggressive instruction scheduling techniques that reduce the critical forwarding path introduced by the data forwarding associated synchronizations.

2.4.2 Our Non-Distributed TLS Approach

Even though hardware support for speculation is not available yet, we set out to explore the benefits of TLS and implemented a software framework. Our approach is similar to the one of Rundberg and Stenstrom [59], in the sense that loads/stores from/in addresses that cannot be statically disambiguated (*speculative locations*) are replaced with calls to functions which simulate the data dependence checking that would be present in a speculative cache protocol.

For each variable that may be the cause of a data dependence violation, the compiler associates a *speculative variable* whose structure is presented in Figure 2.24.

Figure 2.24 The structure of a speculative variable:
Shadow Data Vector: stores the variable values for different iterations
Original Value: safe point value – usually the value before speculation has started
Load Vector: if entry with index i is set then thread i has "read" the variable
Store Vector: if entry with index i is set then thread i has "written" the variable
Lock Variable: is used to avoid race conditions



Our architecture also assumes that the compiler replaces a "read/write" of such a variable with a call to a *speculative load/store* function.

Upon a *speculative load*, a thread marks up the entry in the *load vector* corresponding to its iteration number (*id*), then searches the *store vector* to find the greatest thread id less than its own that has "written" the variable, and finally fetches the value from the corresponding entry in the *shadow data vector*. Similarly, a *speculative store* sets up the entry in the store vector corresponding to the *id* of the thread that executes the operation, then stores the value at the corresponding index in the shadow data vector, and finally checks if a later thread has "read" a value that should have been provided by the current thread. If the latter is true than a data dependence violation has occurred and a *rollback* has to be serviced. The rollback routine usually sets a barrier, and waits for all but one thread to reach the *waiting* state.

The running thread is the one with the lowest id that has a rollback to service, and consequently its state is the same one obtained after sequentially executing *thread id* iterations (correct/valid state). The *original value* is updated to the value known by the current thread, and then the rest of the vectors (load, store, shadow) are cleared, and either the speculative or the sequential execution is restarted.

Our approach, however, is at a much higher level than that of [59], which implemented a speculative framework in a mix of C, and assembly. The initial idea behind our framework was to incorporate TLS into the repertoire of an adaptive dynamic optimizer such as JikesRVM [4]. Profiling could detect situations in which speculation might be applicable and even resolve statically unsolvable distance-vector equations which rely upon run-time values. This monitoring of the run-time state could be used to possibly reduce the number of dependence violations encountered by initiating threads separated by the observed dependence distance. The addition of TLS to a traditional parallelizing compiler could provide speed-up where data dependence analysis fails to conclusively determine if dependencies exist across loop iterations. The access to the true run-time behavior of a program that a dynamic compiler has could as well be used to direct the shape of the iteration space by identifying whether a block or cyclic iteration pattern is most applicable. Further adding to the adaptability of the system, profiling can be integrated into the rollback handler. The ratio of rollbacks to commits could be monitored and if an unacceptable threshold is reached, the run-time compiler could remove the speculative code. Many hardware based schemes suffer from the inability to control the amount of memory required by speculative threads in order to keep the main state isolated from the speculative state [54]. In our software approach we can resize or set an upper bound on the size of the speculative cache as needed.

Our goal is to apply speculation to parallelize general Java programs, as opposed to the conservative parallelization techniques that work well only on very regular scientific applications. To achieve this, a dynamic compiler carrying out the speculative transformations must be able to plug in speculatively aware versions of the Java class libraries. Specifically, in order to speculate on many common code sequences speculative versions of the collection classes, such as `List`, are needed. Consider the common situation of iterating through a `List`. Given a speculative version of the `List` class, a dynamic compiler could replace the use of the sequential library with a speculative version which cuts the `List` into segments dependent upon the number of available processors. Each processor would then visit in parallel only its assigned part of the `List`, and dependence checking would be hidden behind the scenes in the implementation of the speculative `List` class.

Chapter 3

ALMA

3.1 Chapter Introduction

This chapter studies an interoperability problem between two very different computer algebra systems: Maple and Aldor. The investigation has two main goals: First, 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, and in matching the Aldor's strongly type system with Maple's weakly typed system. Second, we are interested in the practical problem of using Aldor as an extension mechanism for the popular Maple computer algebra system. This 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. The work presented here is based on the ISSAC paper "Domains and Expressions: An Interface between Two Approaches to Computer Algebra" [44], co-authored with Stephen Watt.

One of the positions held over the past two decades of mainstream computer al-

gebra system design has been that there should be one over-arching language that serves both the end user and library developer. The idea has been if the language is good enough for end-users, it should be good enough for system developers, and otherwise it needs fixing. This has led to systems that either use modified scripting languages for their libraries (e.g. Maple), or that use modified library-building languages for their user interface (e.g. Axiom). A variant of this approach has been to build much of the mathematical support in a lower-level system implementation language, such as Lisp (e.g. with Macsyma) or C (e.g. with Mathematica). The result is that large parts of the current computer algebra systems are written in languages poorly adapted to the purpose, resulting in systems that are less flexible, less efficient and less reliable than we might wish.

This chapter examines the structure required for a different approach: to write libraries in a language well-adapted to large-scale computer algebra programming, together with an environment aimed at ease of use by the general end-user.

It is not difficult to see that the style of programming for top-level problem solving and for libraries is quite different. For interactive problem solving, or for one-off scripts, it is important to be able to write commands quickly and succinctly. In this context, manipulation of some sort of general expression provides flexibility. On the other hand, to program large-scale computer algebra libraries, there are advantages to a language that allows efficient compilation, secure interfaces, and flexible code reuse. However, to achieve efficiency, safety and composibility requires more declarative structure. In this context, it is more natural to work with objects in precisely defined algebraic domains. Since libraries are used many times more than top-level scripts, programmers are more willing to provide this structure.

Extensions to computer algebra systems are not always calls to larger software components; they may equally well be collections of very fast light-weight routines. We therefore look beyond the solutions offered by loosely coupled computer algebra systems, e.g. OpenMath[51] or the software bus[55]. We choose Aldor [68] as a suitable library-building language, Maple [38] as a suitable interactive environment, and we require that Aldor libraries to be tightly coupled to Maple. That is, Aldor libraries will receive and directly operate on Maple objects in the same address space.

Our solution consists of two parts: The first part allows the low-level run-time systems of Maple and Aldor to work together. It allows Aldor functions to call Maple functions and *vice versa*, and provides a protocol whereby the garbage collectors of the two systems can cooperate when structures span the two system heaps. As any low-level foreign function interface, it holds the user responsible for correct usage. This work was conducted by Watt and was reported in [67].

The second part of our solution, reported here, implements a high-level correspondence between Maple and Aldor concepts. The aim has been to bridge the semantic differences between the two environments, to allow Aldor domains to appear to the user as Maple modules, and Maple modules to appear as Aldor domains. While our semantic correspondence works both ways, in practice we are primarily interested in using Aldor libraries in the Maple environment. We use a tool to generate Aldor, C and Maple code that wraps the Aldor library exports, as well as supporting run-time support code to do dispatch and caching.

The resulting package, which we call *Alma*, allows standard Aldor libraries to be used in a standard Maple environment [38]. More precisely, Alma can be seen as a software component architecture to achieve connectivity among two computer algebra

systems. It gracefully handles user-errors (type-checking), supports reflective features to describe components' types and functionalities, provides a user-oriented interface (Maple "look and feel"), and employs high-level optimizations. Thus, our approach is more challenging and quite different than previous work on low-level foreign function interfaces, and consequently the internal architecture of the proposed framework is more complex.

We present two validations of this architecture: First, we describe the mappings of the Aldor language features to Maple, and the Alma type-checking process (Section 3.5). Second, we present a comprehensive example, in which approximatively 1160 Aldor exports have been made available to the Maple user (Sections 3.2 and 3.7). Earlier results leading to this approach have been reported in [12, 42].

We see the following as contributions of this work:

- Aldor has been found to offer efficiencies comparable to hand-coded C++ [1]. Our approach therefore allows extension libraries to operate with efficiencies comparable to Maple kernel routines.

- These extensions are written in a high-level language, well-adapted for mathematical software. It allows the programmer to ignore lower-level details and have natural integration of dynamic components into the Maple environment.

- Aldor is designed for mathematical "programming in the large" and provides linguistic support for such concepts as generic algorithms, algebraic interface specification and enforcement, dynamic instantiation, etc. Our approach allows the Maple system to benefit from these features. Alternatives, such as `C++`, do not provide this.

- Authors of large Aldor libraries often wish to make their functionality available through a main-stream computer algebra system. Two examples are Bronstein's library for differential operators, Sumit [3], and Moreno Maza's library for triangular sets, Triade [39]. The current work makes this relatively easy.

The remainder of this chapter is organized as follows: We start with an example in Section 3.2: we show a Maple session computing the polynomial GCD over a tower of algebraic extensions using Aldor's BasicMath library. Section 3.3 briefly introduces the aspects of the Maple and Aldor programming environments needed to understand Alma. Section 3.4 presents a high level architectural view of the Alma framework, together with an example of user-Alma interaction. Section 3.5 describes the Maple mapping, together with our type-checking mechanism. Section 3.6 describes the key ideas used in the Aldor and C mappings. Section 3.7 shows the implementation side of the example started in Section 3.2. Finally, Section 3.8 presents some conclusions.

3.2 Example

This section presents an example where a Maple user employs the functionality of the Aldor BasicMath library to solve a mathematical problem in a way not supported natively in Maple. The BasicMath library was developed at NAG by Moreno Maza and others as part of the FRISCO project, and provides Aldor with a set of types and algorithms for computer algebra. In particular, our example uses its support for regular chains. It is an extensive library, comprising about 103700 lines of Aldor code.

The Maple session presented in Figure 3.1 shows the Alma interface. The example computes the greatest common divisor of two polynomials ($f2 = 6 * x * y - y^2 + 5$

Figure 3.1 A Maple session computing a GCD in $(R[x]/\mathrm{Sat}(m_x))[z, y]$ using the Alma framework

```
> read "mtestgcd-wrap.mpl":

# Construct polys
> f1   := MapleToAldorPoly(x*y^2 - 4*y + 5*x):
> f2   := MapleToAldorPoly(6*x*y - y^2 + 5):
> m    := MapleToAldorPoly(x^2 + 1):

# Form triangular set and gcd by Aldor package.
> trset := TriPack:-empty():
> rchain:= TriPack:-regularChain(m, trset):
> ggcd  := TriPack:-regularGcd(f1, f2, rchain):
> ggcd  := genstep(ggcd):
> ggcd  := TriPack:-reducedForm(ggcd, rchain):

# Get the GCD as a Maple expression.
> AldorToMaplePoly(ggcd);

      y - x
```

and $f1 = x * y^2 - 4 * y + 5 * x$) in $(R[x]/\mathrm{Sat}(x^2 + 1))[z, y]$ by invoking the Aldor BasicMath library and using its support for regular chains. The session uses the file `mtestgcd-wrap.mpl` to act as a wrapper between the Alma system and the user. The implementation of this file is explained in Section 3.7, after we have described the necessary concepts.

The example first creates the Alma objects corresponding to the given Maple polynomials. The regular chain containing the polynomial m is constructed, and the greatest common divisor of $f1$ and $f2$ with respect to the regular chain is computed. Finally, the reduced form of *ggcd* is computed, and it is converted to a Maple polynomial. As the last line of Figure 3.1 shows, the computed GCD is $y - x$.

Figure 3.2 A Maple module and its use

```
1 makeZp := proc( p )
2    module()
3       export plus;
4       plus := (a,b) -> a + b mod p;
5    end module:
6 end proc:
7
8 z5 := makeZp(5); # create the module
9 z5:-plus(2,4); # add 2 and 4 mod 5.
```

The functions `empty`, `regularChain` and `regularGcd` have the interfaces exactly as exported by the Aldor library. `TriPack` is the instantiation of an automatically generated Maple module wrapper corresponding to the Aldor package `RegularTriangularSet`.

3.3 Aspects of Maple and Aldor

This section briefly presents some of the aspects of Aldor and Maple systems that we used in our architectural design.

Maple uses a dynamically typed language that supports first class functions. Typically, functions use dynamic type tests to implement polymorphism, and name overloading is not supported. Modern versions of Maple have adopted the concept of *modules* to organize packages and libraries. A module is a first-class Maple object and provides a collection of name bindings. Some of these bindings are accessible to Maple code outside the module, after the module has been constructed; these are the *exports* of the module [38]. Figure 3.2 shows a Maple *module* and its use.

Figure 3.3 An Aldor category/domain example

```
1  -- File Example.as:
2  import from SingleInteger;
3  define Module(R:Ring) : Category ==
4  AbelianGroup with {
5     *:      (R, %) -> %;     ++ Scalar multiplication
6     coerce: R     -> %;
7     coerce: String -> %;
8  }
9
10 ++ Polynomial domain over ring R
11 Polynomial(R: Ring) : Module(R) == add {
12    (r: R) * (x: %) : %  == ...;
13    coerce(r: R) : %     == ...;
14    coerce(s: String) : % == ...;
15    ...
16 }
```

As they are first class objects, modules can be returned by functions. A module's exported functions can reference environment variables visible at the moment of their creation (i.e. it is a closure). In Figure 3.2 the module returned by the `makeZp` function references `makeZp`'s parameter `p`. It exports the `plus` operation whose functionality is to add numbers modulo `p`.

We remind the reader that the **Aldor** language has already been introduced in Section 2.1.1. An example of an Aldor program is presented in Figure 3.3. It defines a parametrized category `Module(R)`, representing a simplified version of the mathematical category of *R-Modules*. `Module(R)` declares as exports a scalar multiplication and two conversion operations from an element of the ring and from a `String` object to an object of type `%`. The type `%`, within a domain-valued expression, refers to the domain being computed. `Polynomial` has the dependent mapping type: `(R: Ring)-> Module(R)`, taking one parameter `R`, which is a domain satisfying the `Ring`

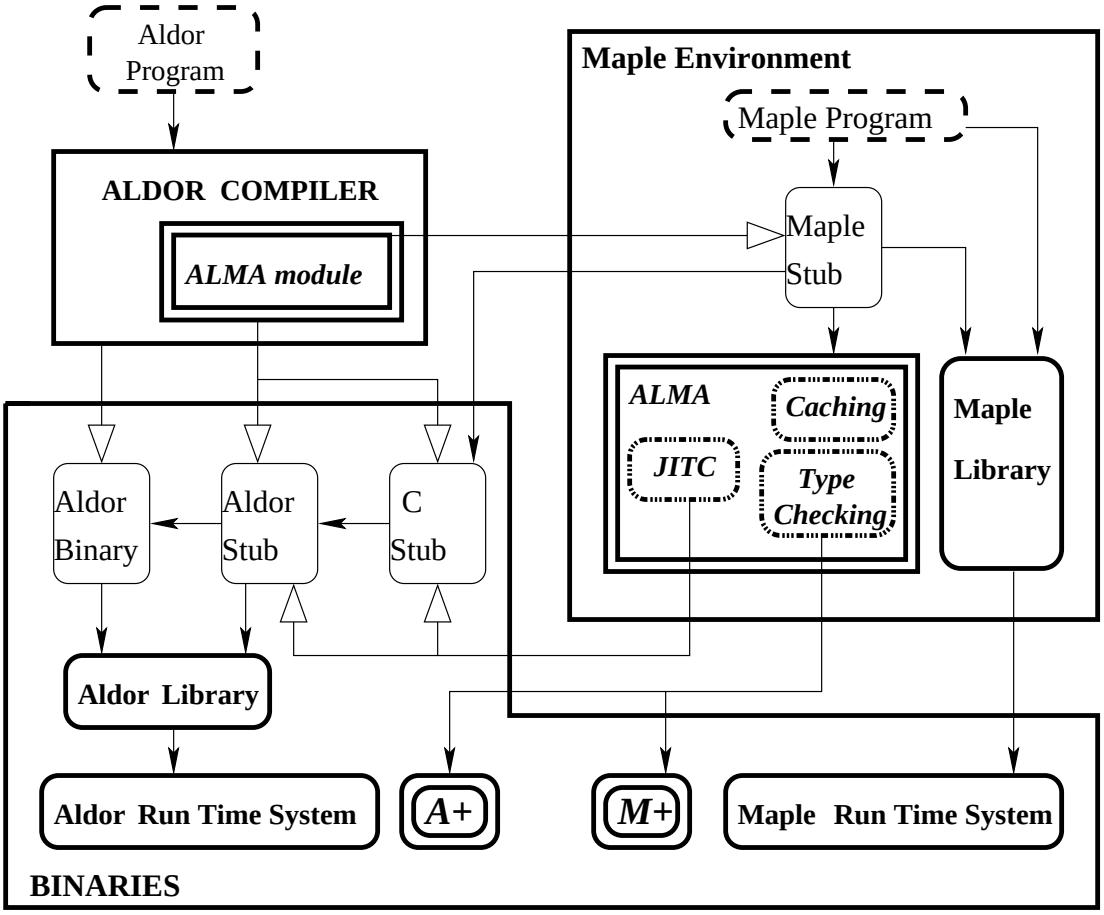
category, and produces a type beloning to the category of R-modules. Static analysis can use the fact that `R` provides all the operations required by `Ring`, thus allowing static resolution of names and separate compilation of parameterized modules. Names can be overloaded, and are resolved based on their static type. The first line in the Aldor code in Figure 3.3 makes the exports of the `SingleInteger` domain available throughout the file.

3.4 Alma Design

This presents an overview of Alma's design. The main ideas that guided the design are summarized below, and then further detailed:

- Alma should automatically generate any needed (Maple, C, and Aldor) stubs, and keep the system's internals hidden from the user.
- Alma should provide a dynamic (interactive) type-checking mechanism that gracefully handles user needs, and errors.
- Alma should allow Maple to interact with Aldor components in an efficient manner, introducing only a minimal overhead cost.
- Alma should extend the Maple language only as needed, by providing mappings for foreign programming language concepts such as overloading, domains, etc.
- Alma should be simple to use, rendering a Maple "look and feel" to Aldor code.

Figure 3.4 High-level architecture overview:
white arrows mean "generates,"
normal arrows mean "uses,"
dashed boxes are user source code,
light boxes are generated code.



3.4.1 Rationale of the Design

Figure 3.4 introduces the main components of the Alma architecture. The module that does the stub code generation is located inside the Aldor compiler. It receives as input an Aldor program, and generates the usual compiled binary representation of it, together with *Aldor, C, and Maple* stubs for the program's exports. Among these there may be exports that have their definition in some Aldor library.

The Maple stub becomes the interface between the user and the Alma system. It uses the functionality of the *type checking module*, in order to ensure a correct call to the Aldor library. Otherwise, if no type-checking is performed, an incorrect call on the user's part would most likely produce a low-level fatal error. The type-checking module is designed to provide useful feedback to the user in the case of an erroneous invocation. For example it will list the allowed export types for a given export name.

Once the program has reached a mature phase, one may want to eliminate the type-checking overhead. If a fast implementation is desired, the Alma code generation module is able to produce code in which no type checking is performed. The *type checking module* is implemented mostly in Maple, but it also uses Aldor run-time system enhancements (the "has" operation that tests if a given domain satisfies a given category).

Our architecture allows Maple to share a single address space with optimized Aldor components from a library. However, the cost of calling an Aldor function can be somewhat expensive: The initial use of the Aldor stub may require a number of expensive runtime operations, such as domain instantiation, that cannot be statically optimized by the Aldor compiler. (These operations may involve parameters that are known only at runtime.) Thus, for performance reasons, the Maple stub uses a *cache* module (implemented based on Maple's *remember* option) to store previously computed domain/category types. Aldor-closure objects corresponding to functional Aldor exports are also cached, so that they can be invoked directly, thus by-passing the Aldor stub. Alma supports function-level just in time re-compilation of the C and Aldor stubs (JITC module). More precisely, if an export is found to be "hot" and depends on type-parameters known only at run-time, Alma will build, and re-compile a specialized C and Aldor stub corresponding to that export. Since most

type-parameters are now instantiated, the Aldor compiler will find better opportunities for aggressive optimizations (like inlining), thus improving the application's performance.

In order to successfully complete a foreign Aldor invocation, the Maple stub calls the C stub that forwards the request to the Aldor stub, and this invokes the correct Aldor export on valid Aldor parameters, returning a value to the C stub. The C stub creates foreign Maple objects and returns them to the Maple stub. The functionality of the Aldor and Maple run-time system enhancements modules (`A+` and `M+` in the figure) is to synchronize *Maple's* and *Aldor's* garbage collectors (see [67]).

Alma's objects expose rich reflective features that can be queried by the user. This allows one to find the functionality of the corresponding Aldor component, its type, etc. Alma's foreign objects can also be manipulated in the same way as any ordinary Maple objects: They may be used in Maple operations (such as `map, apply`), while Alma's internal invocation mechanism is completely transparent to the user.

3.4.2 Example of Correspondence

We present a simple interaction between the Maple-user and the Alma framework. Assume the user wants to use the functionality of the Aldor code in Figure 3.3, and the stubs have already been generated by calling the Aldor compiler with the appropriate options on the `Example.as` file.

The first line in Figure 3.5 imports the Maple stub into *Maple's* environment. On line 3, the user asks for information about the `Polynomial` domain. Alma answers by providing the type information, exports, and the comments associated with the `Polynomial` domain (see Figure 3.3). Similarly, on line 4, the user asks about the `*`

Figure 3.5 User-Alma interaction.
Lines starting ">" are user input; the others are Maple output

```
>read("MapleExampleStub.mpl"):                                  # line 1

>with(Example);                                                 # line 2
  module() export Polynomial, ... end module

>Polynomial("help");                                            # line 3
  Domain Type: Polynomial(R: Ring) : Module(R)
  Exports:
     *     : (R, %)    -> %;
     coerce: (R)       -> %;
     coerce: (String)  -> %;
  Comment: Polynomial domain over ring R

>Polynomial("help", "*");                                       # line 4
  Functional Type: *: (R, %) -> %;
  Comment: Scalar multiplication

>SI_dom := SingleInteger:-Info:-asForeign;                      # line 5
  ["d", 1856856, module() export ...]

>int_obj := Alma:-AldorInt(5);                                  # line 6
  ["o", 5, module() export ... ]

>poly_si_dom := Polynomial(SI_dom);                             # line 7
  ["d", 1848300, module export ...]

>poly_obj := poly_si_dom:-coerce(int_obj);                      # line 8
  module() export ... end module

>wrong_obj:=Polynomial(SI_dom):-coerce(SI_dom);                 # line 9
  no function with this signature! candidates:
    coerce:(SingleInteger)->Polynomial(SingleInteger)
    coerce:(String) -> Polynomial(SingleInteger)
```

export of the `Polynomial` domain. All Aldor domains/categories are translated into Maple modules, or functions producing modules, if parameterized. They export an `Info` module that encapsulates the type's reflective features. The `asForeign` export of the `Info` module stores a Maple foreign object corresponding to the Aldor domain it represents. At present, our implementation represents a foreign Maple object as a list that contains a classification identifier ("d" means domain, "c" means category, "f" means function, "o" means object, etc.), a pointer to the Aldor object (for primitive types this will be the value), a Maple structure representing the Aldor type, and some additional information used to synchronize the garbage collectors This is illustrated by Alma's response to the user command at lines 5, 6, 7.

Line 5 creates a *foreign domain type object* corresponding to the `SingleInteger` Aldor domain. Line 6 creates an object of type `SingleInteger` which in fact is just a primitive integer value, as one can see in the Maple representation of the `int_obj`. Next, on line 7, another domain-type object is created, corresponding to the `Polynomial(SingleInteger)` Aldor type. If the user would like to verify first that the `SingleInteger` domain satisfies the `Ring` category, he can look in the `SingleInteger:-Info:-supertypes` export. Note that the interaction with our framework is quite intuitive, as our mapping closely follows *Aldor's* specification structure and semantics. Types are run-time values both in Aldor and in our mappings: the user has to construct them first in order to use their exports. Types are also first class values, therefore they are constructed and used in the same way a regular object is used. Finally, on line 8, the `coerce` function is called, and as result, a foreign Maple object of `Polynomial(SingleInteger)` Aldor type is returned.

The last line in our example (line 9) shows how our framework reacts to an erroneous input: The type checking module detects that the parameter to the coerce

export is neither of type `SingleInteger` nor of type `String`, so the incorrect Aldor library invocation is aborted. In addition, feedback is provided to the user with respect to the valid type signatures of the `coerce` function. Also note that while Maple does not support overloading, our mapping behaves as though it does. To the user it seems as though one can call two functions with the same name and with different parameter types, as they appear in the Aldor specification.

3.5 The Maple Stub

We now turn our attention to the internals of the system, starting with the generated Maple stubs.

The Maple mapping addresses the issues that arise from matching the Aldor's strongly typed system with Maple's dynamically typed system. In particular, one of the challenges is in matching the compile-time parametric polymorphism of Aldor's dependent types with the dynamic polymorphism of Maple's module-producing functions. For a rich connectivity between Maple and Aldor to exist, Aldor's features, such as run-time domain types, overloading, dependent types and mapping types, need to be mapped to Maple. The key for the translation of these features is to create, via the Maple stub, dynamic types corresponding to the hierarchy of available Aldor types, and to design a dynamic type-checking mechanism for the foreign Maple objects. Alma's type-checking phase is greatly simplified, compared to the static Aldor type-checking, as it happens at the application's run-time where most parameters have completely instantiated types.

3.5.1 Mapping Rules

The code in Figures 3.6 and 3.7, is an extract of the Maple stub corresponding to the Aldor `Polynomial` domain defined in Figure 3.3. We use this to help present the high-level concepts ideas used to interface Maple with Aldor. To keep the figures simpler, we have excluded the code for the `coerce` exports, the "help" option, or some of the exports of the `Info` module.

An Aldor domain-producing function (e.g., `Polynomial`) is translated into a Maple function which at run-time yields a module. In addition it encapsulates the necessary information for type-checking its parameters and exports. This is done on lines 7, 36, and 37 in Figure 3.6; `type/TC` is the Alma type-checker that ensures the consistency of the mapped Maple code with the Aldor type system. Aldor's nested domains are mapped into nested Maple modules. The rest of the Aldor domain exports are mapped to Maple module exports. Name overloading in our mapping is achieved by concatenating the different implementations for the same name and using a single function in which dynamic type tests identify the right code to be executed.

Modules corresponding to Aldor's domains and categories export an `Info` module containing metadata (reflective features and profiling information) associated with that type. These standardized exports of the `Info` module are computed at the domain/category-type module creation time.

Our mapping exploits Maple's support for closures. Each of the generated functions that produces a type will set a variable with a unique name to point to its parameters list, thus guaranteeing access to its parameters from a function declared in a nested scope. Line 36 and 37 in Figure 3.6 type-check the `*: (r:R, x:%)->%` export of the `Polynomial: (R:Ring)->Module(R)` parameterized domain. Notice that here

Figure 3.6 Part of the MapleExampleStub.mpl file

```
1  `Polynomial` := proc() option remember;   ## cache for domain types

2    local ret, tmp_fct, b, ret_param, ALMA_getObject, args4;
3    if(args[1]="help") then ... return; fi;
4    args4 := args;

5    if nargs=1 then
6      b := true;
7      if b then   b := type( args[1], TC(Ring()) );   fi;

8      if b then                                  ## TYPE: Module(R:Ring)
9        ret := module()

10         export `*`, Info, fcts;
11         Info := module()        ## metadata: reflective + profiling

12           export GenExports, GenInfo, hash, self, asForeign,
13                  type, asForeign, supertypes, printExports,
14                  domArgs, domArgsOpt, optimizeOn, profile;

15           GenInfo := ["Polynomial",[["Ring"]],
16                                    ["Apply","Module",[args4[1]]]];
17           GenExports := [["*",[args4[1],"%"],["%"]], ...];
18           domArgs := args4; domArgsOpt := [];
19           optimizeOn := [0]; profile := [[0]];
20         end module;

21         fcts := module()
22           export `*`, `*clos`, `coerce`, `coerceclos`;
23           local `*cstubname`; `*cstubname` := "starFrMyPolyT";

24           `*clos` := proc(arg)  option remember;   ## closure cache
25             local tmp_fct, ret;
26             tmp_fct := define_external( convert(`*cstubname`,
27                    symbol), 'MAPLE', 'LIB = "./libctestJIT.so"):
28             ret := tmp_fct( Alma_map(lst->lst[2], [op(arg)]),
29                      ["f",[args4[1,3],Info:-self],[Info:-self]]);
30             return ret;
31           end proc;
```

Figure 3.7 Part of the MapleExampleStub.mpl file – continuation

```
32          `*` := proc()
33            local ret, cached_clos, b, ret_param;
34            if nargs = 2 then
35              b := true;
36              if b then b := type( args[1], TC(args4[1,3]) ); fi;
37              if b then b := type( args[2], TC(Info:-self) ); fi;
38              if b then
39                if (Info:-optimizeOn[1] = 1) then
40                  cached_clos := `*clos`(domArgsOpt);
41                else cached_clos := `*clos`(domArgs); fi;
42                Info:-profile[1,1] := Info:-profile[1,1] + 1;
43                if( Info:-profile[1,1] = Alma:-JITtreshold ) then
44                  `*cstubname` := "starFrMyPolySpec";
45                  Info:-optimizeOn[1] := 1;
45                  OptimizeAldor(Info:-self, 1);  fi;
46                ret := callAldorClosure(cached_clos,
47                             map(lst->lst[2],[args]),[Info:-self]);
48                return ret;
49            fi;   fi;

50            print("Context: Polynomial(R:Ring);");
51            print("Candidates: *(R,%)->(%)");
52            error "No function with this signature";
53        end proc; ... end module;                ## end fcts module
54        `*` := fcts:-`*`;
55      end module;                                 ## end ret module

56      ret:-Info:-self := ret;
57      Alma_getObject := proc()     local tmp_fct, ret1;
58        tmp_fct := define_external('cPolynomialOfT',
59                                'MAPLE', 'LIB' = "./libctestJIT.so");
60        ret1 := tmp_fct(map(lst->lst[2],[op(args4)]),
                                                [ret:-Info:-self]);
61        return ret1;
62      end proc;
63      ret:-Info:-asForeign  := `Alma_getObject`();
64      ret:-Info:-type       := Module(args[1]);
65      ret:-Info:-supertypes := [Type];    return ret;
66  fi; fi;

67  print("Context:");
68  print("Candidate:Polynomial:(R:Ring())->Module(R)");
69  error "No function with this signature";
70 end proc;
```

`R` is a type variable, as it is given as a parameter to the `Polynomial` domain, and is used as a type in its implementation. `R` can be accessed by means of the `args4` variable in the `Polynomial`'s function outer scope. The `type(args[1],TC(args4[1,3]))` call invokes the Alma type-checker (`type/TC`) to verify if `r` is of type `R`. This uses the representation knowledge that the third element in *Aldor's domain foreign object* representation is the Maple module object that maps the corresponding Aldor domain). Both `r` and `R` are known only at run-time, and are accessible through the closure's environment.

Modules corresponding to *Aldor's* domains and categories export an `Info` module containing metadata (reflective features and profiling information) associated with that type. The `Info` module provides a set of standardized exports that are computed at the domain/category-type module creation time:

- an `asForeign` member that stores a *Maple* foreign object corresponding to the Aldor domain it represents;
- a `self` member that keeps a reference to itself, so that it can type-check a parameter whose type is *Aldor's* `%`;
- a `hash` member that uniquely identifies a type (as it is computed based on its parameter hashes - if any, and on its *Aldor* static hash) - this is used by type-checking;
- a `flag` member specifying whether or not the module represents a category/domain type, or other "special" types (such as `Category`, `Type`, a "Join" of several categories, etc);

- a `type` member that stores a module object corresponding to the *Aldor* type of the current object;
- a `supertypes` member which, for a category module stores its super-types (inherited categories) represented as Maple module objects;
- a `GenInfo/GenExports` member that records name and type information of itself/its exports; a `printExports` member pretty prints all or part of this information;
- `optimizeOn` and `profile` members are to be used in the context of just in time (JIT) re-compilation optimizations. For each domain-export `optimizeOn` shows whether or not it has been already recompiled, while `profile` stores profiling information (gathered during the applications run-time) for each domain-export;
- `domArgs/domArgsOpt` expose the parameters of the original/ JIT type-specialized domain/category.

Lines 32-53 in Figure 3.6 show the implementation of the `*` export of the `Polynomial` domain. Lines 34-37 verify that the number and type of the parameters are consistent with the Aldor definition. If the `optimizeOn` entry associated with the `*` export is set, then this export has already been type specialized and re-compiled (see Section 3.6). In this case, the `*clos` function is invoked on the non-inlined parameters (`domArgsOpt`, line 40). Otherwise it is invoked on all domain parameters (`domArgs`, line 41). The `*clos` function returns an Alma closure-object corresponding to the Aldor `*` export. This is invoked with valid parameters on line 46 by means of the `callAldorClosure` Alma's system-function. Its first parameter is the Alma closure object, the second is a list of Aldor valid arguments, while the third is the Alma type

of the result. If the profiled information corresponding to the `*` export shows that it is advantageous to JIT recompile the C/Aldor stub (line 43), the Alma system-function `OptimizeAldor` is called (line 45).

In order to return the Alma closure-object corresponding to the Aldor `*` export, the `*clos` function requires access to the C stub via the *Maple's* `define_external` function. The call to `define_external` links in an externally defined function, and produces a Maple procedure that acts as an interface to this external function [38]; the `tmp_fct`, computed on line 26 of Figure 3.6, is such an interface. The first parameter of the `tmp_fct` is a list of Aldor parameters on which the Aldor stub is to be invoked, while the second argument (`["f",[args4[1,3],Info:-self],[Info:-self]]`) is the Alma type of the closure object (`*` receives two parameters: one of type `R` – where `R` is a type-parameter of the `Polynomial` domain, and another one of type `%`, yielding an object of type `%`).

We note that, the type/closure cache of the Alma system is easily implemented with Maple's *option remember* (lines 1 and 24).

3.5.2 Foreign Object Layout

We now briefly describe the Alma foreign object layout. Where necessary we shall provide details on *Aldor's* type system semantics.

In Aldor, types and functions are first class values: they can be created at run-time, and can be passed as parameters/returned to/by functions. Therefore, besides "regular" objects, we have to define proper formats for foreign Alma type/closure objects, and to design proper Maple types for them.

Figure 3.8 Aldor specification

```
1  -- any Aldor domain satisfies Type
2  -- any Aldor category type satisfies Category
3
4  define MyCat(T:Type): Category == with;   -- Type: Category
5
6  MyDom(T:Type): MyCat(T) == add;           -- Type: Domain
7
8  fun( A:Type, o:A, obj:MyDom(A) ): A == o; -- Type: Function
9
10 SI == SingleInteger; a: SI := 3::SI;       -- Type: Object
```

Figure 3.9 shows the foreign object layout for the Aldor expressions: `MyCat(SI)` (category-type object), `MyDom(SI)` (domain-type object), `a` (object of `SingleInteger` type), and `fun` (function), which have all been defined in Figure 3.8. For objects that do not correspond to domain/category Aldor types, the third element of Alma's foreign object layout (rows 3 and 4, column 2) is their Alma type. The layout of the domain/category-type objects does not include their types, but rather themselves (see rows 1 and 2 in Figure 3.9). For example `MyCat(SI)` is the Alma module-type associated with the Aldor `MyCat(SingleInteger)` category. This is because their Alma type is readily accessible by means of their reflective features (the `Info:-type` export).

Row 4 in Figure 3.9 shows the Alma type corresponding to the `fun` function. It is composed from a classification identifier "f", a list containing the Alma types of its parameters and another list containing the Alma types of its returns. A list whose first argument is the "l" tag (link) indicates that the parameter's type is itself passed as a parameter to this function, the remaining list's arguments giving the index in the current type where the type-parameter was introduced. The `"a"` tag stands for

Figure 3.9 Foreign object layout. The Aldor expressions in the first column are defined in Figure 3.8

Aldor Expr.	Associated Alma Foreign Object Layout
MyCat(SI)	["c", ptr_to_obj, MyCat(SI)]
MyDom(SI)	["d", ptr_to_obj, MyDom(SI)]
a	["o", 3, SI]
fun	["f",ptr_to_clos,f_tp] where l1:=["l",1], f_tp:=["f",[Type,l1,["a",MyDom,[l1]]],[l1]];

"apply the second argument of the set to the rest of the set's arguments." It is used only if the type expression involves a type-parameter that has not yet been computed (thus the type of the "apply" cannot be computed yet in this case).

3.5.3 Type Checking

Let us now consider Alma's type-checking mechanism. In Aldor, every value is a member of a *unique domain* that determines the interpretation of its data. For the current version of the language, only the domain of all domains, and the domain of all functions produce non-trivial subtype lattices [68][70]. This means that user-developed domains cannot create subtypes, the only non-trivial sub-typing lattices for our type system are the lattices of categories and functions; a non-function object is of a unique type, and cannot satisfy any other type.

To type-check that a foreign Maple object `o` is of Alma type `d` (*i.e.* a Maple module corresponding to an Aldor domain type), the Aldor type of `o` (found through the foreign object layout), and the Aldor object representation of `d` are compared, either directly or by hash codees. To verify that an Alma domain-object belongs to

a certain category type, the Aldor run-time system is invoked ("has" operation) via Alma's Aldor stub.

To test that a foreign Maple functional object `S1->T1` is of a functional type `S2->T2`, one has to verify that `S2` is a subtype of `S1`, and `T1` is a subtype of `T2`. 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.

3.6 The C and Aldor Stubs

The role of the Aldor and C stub is to re-direct the user's call to the Aldor library. These are not necessarily accessible to the user, and do not resemble the structure of the mapped Aldor specification. The C and Aldor mappings, if not used inside our framework form un-safe code, as they assume that the type-checking has already been performed at the Maple stub level.

The C stub is the glue between the Maple and Aldor stubs, as both languages expose a basic interoperability layer with C. The C stub for the Aldor export `*:(R, %)->%` is presented in Figure 3.10. When invoked, the C stub (`starFrPolynomialT`) identifies the Aldor objects to be passed as parameters to the Aldor stub (`list_args`), and calls the Aldor stub (represented by `astarFrPolynomialT`) on these arguments (casted to void pointers). The resulting Aldor object (`ret`) is combined with its Alma type (`ret_type`, also received as a parameter by the C stub) to form an Alma foreign closure-object that is returned to the Maple stub.

This is accomplished through the use of the `makeForeignObject` Alma system function. The created closure-object, when called (via `callAldorClosure` Alma

Figure 3.10 C stub mapping

```
/************ C stub for *$Polynomial(T) ************/
extern FiClos astarFrPolynomialT(void* D);

ALGEB starFrPolynomialT(MKernelVector kv, ALGEB args) {
  ALGEB list_args, ret_type, result; FiClos ret;
  list_args = (ALGEB)args[1];
  ret_type = (ALGEB)args[2];
  ret = astarFrPolynomialT((void*)MapleToInteger32(kv,
                  MapleListSelect(kv,list_args,1)));
  result = makeForeignObject(kv,"f",ret,ret_type);
  return result;
}

/****** C stub for *$Polynomial(SingleInteger) ******/
/** Code generated by the JIT re-compilation module **/
extern FiClos astarFrPolynomialSpec();

ALGEB starFrPolynomialSpec(MKernelVector kv,ALGEB args)
{
  ALGEB ret_type, result; FiClos ret;
  ret_type = (ALGEB)args[1];
  ret = astarFrPolynomialSpec();
  result = makeForeignObject(kv,"f",ret,ret_type);
  return result;
}
```

system-function), uses the Aldor-C interoperability layer for executing a closure call (`CCall`). The C stub generated by the JIT re-compilation module looks very much like the standard one. The only difference is that it invokes a different Aldor function (`astarFrPolynomialSpec`) which takes fewer parameters. In our case it takes no parameters as the `SingleInteger` parameter of the `Polynomial` has been already inlined in the `*` Aldor stub export generated by the JIT re-compilation module.

Figure 3.11 illustrates the main ideas employed in the Aldor stub generation. The Aldor stub exposes the parametric polymorphism of the Aldor specification/library

Figure 3.11 Aldor stub mapping

```
1  astarFrPolynomialT(T:Ring) :
2    (Ring,Polynomial(T)) -> Polynomial(T) == {
3      _*$Polynomial(T); -- (***)
4  }
5
6  -- Code generated by the JIT re-compilation module --
7  SI == SingleInteger;
8  astarFrPolynomialSpec() :
9    (SI,Polynomial(SI)) -> Polynomial(SI) == {
10     _*$Polynomial(SI);
11     -- this can be aggresively optimized --
12 }
```

to the Maple user who can now instantiate Aldor types at application's run-time and call their exports. A domain functional export is represented as a function that takes as parameters all the parameters of the domains in which it is nested (starting with the uppermost one), and that returns the desired closure to the C stub. All the other exports shall return an object (for example domain/category types).

As can be seen, the Aldor stub is quite simple. We employ the type inference mechanism to do the difficult work of identifying which of the possible overloaded `*` functions we return. The Aldor compiler will identify more opportunities to aggressively optimize the `astarFrPolynomialSpec` Aldor stub export (generated by JIT re-compilation) – for example it can inline all the `SI` operations (`+, -, *`) that appear in the `*` export of the `Polynomial(SI)` domain.

Figure 3.12 Aldor specification used as input to the Alma framework

```
1  -- File testgcd.as:
2  #include "basicmath"
3
4  N == NonNegativeInteger; R == Integer;
5  lv: List Symbol == [+"z",+"y",+"x"];
6  V == OrderedVariableList(lv); Q == Fraction(Integer);
7  P == SparseMultivariatePolynomial(Q, V);
8  gcdPack==GcdOverTowersOfAlgebraicExtensionsPackage(lv);
9  T  == RegularTriangularSet(Q, lv);
10 VT == ValueWithRegularChain(P, T);
11 BB == Boolean; SI == SingleInteger;
```

3.7 Example Implementation

We now show the details a library author must be aware of to use Alma. This completes the example of Section 3.2, which showed only the end-user's point of view.

Figure 3.12 shows the Aldor specification that must be provided as input to the Alma compiler to make available *part* of BasicMath's exports to the Maple environment. The compilation generates Maple, C, and Aldor stubs that each have about 1160 exports. It is, in our opinion, easy to see why a naive, non-automatic integration of this library in the Maple environment is not a practical solution: It requires a good deal of effort, not to mention the maintenance cost. If the exports of the library are changed, the Maple mapping must be altered as well.

Figure 3.13 presents the hand-written wrapping code that will create the necessary Alma types and functions and will ease the use of the Alma system. This is not, strictly speaking, necessary and could be done by the end user. However it is likely that it needs to be created only once, and may be used in different

Figure 3.13 Maple wrapper used in Figure 3.1

```
# Import from generated Aldor file.
read "mtestgcd.mpl": with(testgcd):

# Problem-independent abbreviations.
STR  := String: CHAR := Character: S := Symbol:

# Wrapper for this package.
TriPack := RegularTriangularSet(Q:-Info:-asForeign,lv):
GenP    := Generator(P:-Info:-asForeign):
GenCHAR := Generator(CHAR:-Info:-asForeign):
GenVT   := Generator(VT:-Info:-asForeign):

MapleToAldorPoly :=
   almaPolyToAldor(P,SI,N,R,Q,S,CHAR,STR):
AldorToMaplePoly :=
   almaPolyToMaple(P,GenP,Q,N,R,S,STR,GenCHAR):

# Utility function.
genstep := proc(ggcd0)
   local ggcd, vgcd, str;
   ggcd := GenVT:-‘step¡(ggcd0):
   vgcd := GenVT:-value(ggcd):
   str  := AlmaNewString(SI,CHAR,STR)("val"):
   VT:-apply(vgcd, str):
end:
```

programs. We underscore that the Maple and Aldor generated stubs are generic and can be instantiated over various types. The Alma user may also work with `SparseMultivariatePolynomial(R,V)`, not only with the `P` defined in Figure 3.12.

The code in Figure 3.13 constructs various Alma types and constants corresponding to BasicMath types/constants which will be needed in the computations already presented in Figure 3.1. Note that Alma has also generated Maple exports corresponding to the Aldor constants `lv, N, R, Q, T, ...` (Figure 3.12), and these can now be directly manipulated in the Maple file. The utility function (`genstep`) receives

as parameter a generator object containing $(gcd_i, tower_i)$ pairs, obtained by calling the `TriPack:-regularGcd` function, and returns gcd_1.

The functions `almaPolyToAldor` and `almaPolyToMaple` are Alma system components that return to the user Maple to Aldor and Aldor to Maple polynomial conversion closures.

3.8 Chapter Conclusions

We have described an approach to using efficient, externally defined, high-level mathematical libraries within Maple. These can extend Maple in an effective and natural way. The Alma framework implements an interoperability solution between two languages with very different type models: strongly typed, higher order type system (Aldor) *versus* dynamically typed system (Maple). Our implementation allows Aldor domains to appear as Maple modules, and allows Aldor programs unfettered direct access to Maple objects. This allows very efficient interaction between the two environments.

At this point Alma is most useful in two settings: The first setting is to allow kernel-like efficiency in core mathematical extensions of Maple. The difference between Alma and using C code via Maple's foreign function interface is that it is possible to work at a high mathematical conceptual level and not worry about details such as garbage collection. The second setting is to allow complex Aldor packages to be used naturally from Maple. These packages typically have their own internal representation for the mathematical objects they manipulate. We forsee a third setting where Alma will be used: as an alternative for writing new libraries for Maple. New programs can work naturally with both Maple and Aldor native objects, while the

Aldor compiler enforces mathematical interface requirements and generates efficient code.

Most importantly, we have re-examined one of the most basic assumptions of modern computer algebra system design: that algebra code should be written either in the top-level user language or in the low-level systems implementation language. We believe that we have demonstrated that top-level problem solving and library development can successfully use different mathematical programming languages.

Chapter 4

GIDL

4.1 Chapter Introduction

This chapter examines what is required to have multi-language parameterized components interoperate to create applications, and how to access existing generic libraries across language boundaries. We propose a common model for parametric polymorphism that accommodates a representative range of different object semantics and binding times from various languages, and use it to design a "generic" software component architecture extension that can be applied on top of most component architectures in use today. The work presented here is based on the OOPSLA paper "Parametric Polymorphism for Software Component Architectures" [45], co-authored with Stephen Watt.

Software component architectures provide mechanisms for software modules to be developed independently, using different programming languages, and for these components to be combined in various configurations to construct applications. To provide the richest environment, these architectures have historically attempted to

capture the intersection of features of the programming languages for which they have bindings. Common programming practice has evolved a great deal, however, since the component architectures in common use today were established. Notably, parametric polymorphism has evolved from a beautiful property of research-oriented programming languages to become a standard feature of languages used in main-stream applications. The concept of multi-language, multi-platform components must similarly evolve if we wish our components to enjoy the benefits of parametric polymorphism.

Parametric polymorphism is one mechanism by which programming languages may provide support for generic programming. By associating all behavior of parameter values with the types of the parameters, it becomes possible to write generic programs. These types can either be stated explicitly as parameters to a module, or inferred, depending on the setting. Explicit parametric polymorphism has become more widely used, in practice, and has certain theoretical benefits, including termination of type inference in some higher order languages [35]. Parametric polymorphism increases the flexibility, re-usability, and expressive power of the programming environment, avoids the need for down-casting, and allows a compiler to find more programming errors.

There are quite a few popular programming languages with support for parametric polymorphism, albeit with differing semantics. Section 2.2.2 summarized a few, to give an idea of the range a general facility must be able to map onto. Our conclusion is that a mechanism to combine modules in different programming languages must be able to accommodate both *compile-time* and *run-time* instantiation of modules, and both *qualified* and *unqualified* type variables. Here, we use the term *qualified* as a synonym for *bounded quantification* [7].

Our work described in this chapter was inspired by an early experiment [10], briefly presented in Section 2.1, where two languages with different parametric polymorphism semantics and different binding time models were made to work together. The experiment linked C++, with compile time template instantiation, and Aldor [69, 70], with run time higher-order functions producing dependent types. This experiment motivated the present, more general, approach.

We have developed an extension to CORBA's Interface Definition Language (IDL) to support parameterized interfaces. We have dubbed this extended specification language *Generic IDL*, or GIDL for short. In this chapter we present our implementation of GIDL, which consists of a GIDL to IDL compiler and code generators implementing C++, Java and Aldor bindings. GIDL encapsulates a common model for generics and provides efficient implementation under a wide spectrum of requirements for specific semantics and binding times of the supported languages. Our component architecture extension does *not* assume a homogeneous environment. Its design, which constitutes in our view a novel application of the type-erasure technique to implement generics in a heterogeneous environment, allows it to be easily adapted to work on top of most software component architectures in use today: CORBA is just our working study case.

To test the effectiveness of our model for generics, we have investigated how to use GIDL as a vehicle to access *two generic libraries* beyond their original language boundaries. The first library experiment implements a server incorporating part of the *C++ Standard Template Library (*STL*)* functionality. We have not re-written the STL: our implementation uses the STL as a black box, wrapping it in a manner that can easily be automated. We find that GIDL is perhaps more suitable than C++ to express the STL "orthogonality" semantics. Our specification is self-explanatory and self-contained, in the sense that it does not need free language annotation to

explain type safety constraints. The second library experiment explores the high-level conceptual ideas involved in mapping the semantics of the Aldor BasicMath library to a GIDL specification. We see that Aldor's functional model of polymorphism can be mapped naturally into GIDL.

We see the contributions of our approach as:

- recognition of parametric polymorphism as important to support in a multi-language environment,
- identification of polymorphism semantics suitable for use in this setting,
- the definition of an interface language, GIDL, with mappings for three very different target languages, and
- a report on our implementation experience.

The remainder of this chapter is organized as follows. Section 4.2 compares our generic model with the different parametric polymorphism flavors in various programming languages, and defends the generality of the proposed design. Section 4.3 describes GIDL's semantics, and the GIDL to IDL translation. Section 4.4 presents the high level ideas used in the mapping of the generic type qualifications. Section 4.5 introduces the general architecture of the GIDL base application. Sections 4.6 describes the GIDL bindings of the supported languages (C++, Java, Aldor). Section 4.7 examines the effectiveness of the GIDL generic model, by exposing parts of *two* existing generic libraries to a multi-language environment via GIDL. Finally, Section 4.8 presents some concluding remarks.

4.2 Motivation and Design Point

The initial motivation for our work arose building a linkage between Aldor and C++ in the context of a European project for symbolic-numeric computation. The two main background items brought into the project were (1) a complex, heavily template-based 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 the interesting challenge to make two languages with very different binding time models and parametric polymorphism semantics work together (C++ with compile-time template class instantiation and Aldor with run-time higher-order functions producing dependent types). A summary of this work was already given in Section 2.1, while a detailed account is given elsewhere [10, 11].

This experiment established that we could overcome the C++/ Aldor semantic gap and motivated our search for a systematic solution for parametric polymorphism for components, encompassing more languages in a simpler way.

Since the semantics of generics are different in various programming languages, we have been forced to identify a common ground that can be suitably mapped to these languages efficiently. GIDL supports mappings to Java, C++ and Aldor, thus handling a wide spectrum of parametric polymorphism and binding time semantics. (see Section 2.2.2). We are not aware of any current mainstream languages that would pose substantially new issues. For example, it would be straightforward to provide C# bindings once its template support is finalized.

In our opinion, a language targeted to describe parameterized components should support qualified (rather than only unqualified) type parameters, and compile-time (rather than only run-time) module instantiation. This allows the compiler to find more programming errors. It also allows the component specification to gain in expressivity (with more invariants expressed in type signatures) and in clarity (from the semantics of qualification), enforcing a cleaner separation of specification and implementation.

Our generic model supports a type of bounded polymorphism, in which restrictions can be placed on type variables in terms of both inheritance relations (*extension-based qualification*), and expected functionality (*export-based qualification*). The distinction between the two types of qualification is the standard one between named and structural subtyping. The latter will provide a natural mapping for programming languages that allow type variables to be bounded by a list of exports, and will be useful in cleanly describing the semantics of orthogonally designed libraries (see C++'s STL, Section 4.7.1), or in mapping functional types to GIDL (see Section 4.7.2). Both qualifications are implemented in an uniform manner over the targeted languages (C++, Java and Aldor), with almost no run-time overhead introduced by the generics mechanism.

For our implementation of the generic model, we chose an erasure technique (as in **Java/GJ** [47, 46]), rather than syntax expansion (as in **C++**) or type-valued parameters (as in **Aldor** [70]). The generic type information is "erased" to types that are understood by the underlying component architecture, our mapping to the targeted languages (Java, C++, Aldor) being responsible for recovering the lost (generic type) information. The application of the type erasure technique [47] for implementing generics has allowed us to design a *generic software component architecture extension*

that can work on top of most component architectures in use today (different CORBA implementations, DCOM, JNI), modulo modifications in the targeted language stub code generation phase. In addition, this design enforces the backward compatibility with the non-generic applications written for the underlying component architecture and with applications written in programming languages with no support for parametric polymorphism. GIDL is also comprehensive with respect to binding times, requiring the particular language binding (C++, Java, Aldor) to determine appropriate implementations (see Section 4.6). Code is generated to allow normally static environments to provide dynamic types through virtualization, and to allow normally dynamic environment to be more static through specialization.

4.3 Generic IDL

CORBA–IDL [49] is a declarative language used to describe the interfaces that client objects call and object implementations provide, separating the specification and the implementation aspects of a module. It defines basic types (`short`, `byte`, `float`, `double`, `string`, ...), 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. These services may be provided by local or remote objects and are in principle transparent to the client program. CORBA–IDL's usefulness for language-independent specification has lead to its use outside of its initial CORBA setting. For example, the World Wide Web Consortium provides IDL definitions for its document object models for XML, SVG, MathML, etc.

In this section we present the syntax and semantics of Generic Interface Definition Language (GIDL), our extension to CORBA–IDL that supports parametric polymor-

phism. We have developed a corresponding GIDL compiler, consisting of about 33,500 lines of code in 133 Java classes.

We emphasize from the outset that we do not aim at writing a compliant OMG–CORBA extension; for example we have not as yet modified the CORBA interface repository to handle generic types. We have focused on adding parametric polymorphism at the static IDL level of CORBA so the ideas involved in our design can be applied in a straightforward manner to extend other software component architectures. Reflective features and type repositories are architecture specific and thus not the subject of this paper. However, these type (interface) repositories mirror the IDL specification and therefore similar ideas can be employed to enhance them with support for parametric polymorphism.

4.3.1 Rationale of the Design

We summarize the main principles that guided the design of our GIDL extension. We required that the GIDL's model for generics should:

- be "general" enough to allow a similar extension for various SCAs, and preserve the backward compatibility with non-generic applications
- have the property that the type of an expression be context independent (i.e. be determined solely by the type of its constituents),
- be powerful enough to make specifications written in GIDL clear, precise and easily extensible, allowing qualifications to be placed on generic types,
- allow mappings to languages supporting parametric polymorphism in a natural way, within a small overhead cost.

In the light of the above assumptions we constructed a generic model for GIDL in some ways similar to that of Java (GJ [66]). We are using a homogeneous implementation approach, based on a type erasure technique which ensures the backward compatibility with the non-generic applications written for the underlying SCA. Briefly, the GIDL compiler generates an IDL specification file by erasing the generic type information, and generates wrapper code in the desired programming language (C++, Java, Aldor) to retrieve the erased information.

4.3.2 The GIDL Parametric Polymorphism Semantics

GIDL defines a generalized model of parametric polymorphism that allows us to support a range of languages through various mappings. One consequence is that GIDL is neutral to whether the type parameters are created statically or dynamically; this depends on the targeted language. From a type-system point of view, GIDL supports F-bounded quantifications [5] based on named and structural subtyping. Type variables can be restricted to explicitly extend a given interface, or to implicitly implement all the functionality (methods) of a given interface. The latter addresses the code extensibility and re-usability issue, allowing the programmer to design a clean and precise specification, and to avoid unnatural inheritance relations between interfaces. (This is useful, for example, in rendering the correct semantics of orthogonal-based libraries as the C++ STL.) Furthermore, there are languages like Aldor that can allow type variables to be bounded simply by a list of exports, without demanding a subclassing relationship: `f(A:with{op:(SI)->SI},a:A): SI ==...;`. This type of restriction is discussed further in Section 4.4.

The following example introduces 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 4.1.

The interface `PriorQueue1` specifies a priority queue of objects whose types have to be the `PriorElem` interface or to explicitly extend it (be a subtype of it). We call this an *extension-based qualification*. A type instantiation of an *extension-based qualified* generic type will be validated by the compiler only if it actually inherits from the qualifier, in our case `PriorElem`.

The `PriorQueue2` interface accepts as valid candidates for the generic type all the interfaces that implicitly, fully implement all the operations present in the definition of the `PriorElem` interface. We call this an *export-based qualification*. Note that this definition requires exact matching of method signatures, and does not accommodate functional subtyping (contravariant parameter types, covariant return type).

To illustrate, at line 33 in our example, the type checker will accept the `Test<Foo_extend, Foo_export>` scoped-name, because the interface `Foo_extend` inherits from `PriorElem`, and the `Foo_export` interface implements the whole functionality of the `PriorElem` interface. Line 34 will generate a type error since `Foo_export` does not inherit from `PriorElem`, and therefore violates the extension based qualification of the `A: PriorElem` generic type.

A type instantiation of an *export-based qualified* generic type is valid only if it is found to implement the whole qualifier's functionality. In this example, a call such as `PriorQueue2<Interf>` is valid only if `Interf` contains the operations:

Figure 4.1 Generic interfaces with different generic type qualifications

```
module GenericStructures {

  interface PriorElem {
    short getPriority();
    short compareTo(in Object r);
  };

  interface Foo_extend : PriorElem { /* ..... */ };

  // Assume Foo_export is not in a "isA" logical relation with
  // PriorElem so we did not want to inherit from it
  interface Foo_export{
    short getPriority();
    short compareTo(in Object r);
    //...
  };

  interface PriorQueue1<A: PriorElem> {
    void    enqueue(in A a);
    A       dequeue();
    boolean empty();
    short   size();
  };

  interface PriorQueue2<A:-PriorElem> {
    void    enqueue(in A a);
    A       dequeue();
    boolean empty();
    short   size();
  };

  interface Test<A: PriorElem, B:- PriorElem>{
    Test<Foo_extend, Foo_export> op1(); // OK
    Test<Foo_export, Foo_export> op2(); // ERROR
    Test<Foo_extend, Foo_extend> op2(); // OK
  };
  // ...
};
```

Figure 4.2 Export-based qualification example

```
1
2  interface Elem {
3    Elem op(in string str, in Object o);
4  };
5
6  interface TElem<A, B> {
7    A op(in B b, in Object o);
8  };
9
10 interface Test<A:-Elem>{ };
```

```
short getPriority()

short compareTo(in Object r)
```

This check is not trivial, as shown below:

```
interface Elem {

   Elem op(in string str, in Object o);

};

interface TElem<A, B> {

   A op(in B b, in Object o);

};

interface Test<A:-Elem>{  };
```

Both `Elem` and `TElem<Elem, string>` are valid candidates for the generic type `A` in the definition of the `Test` interface, but this is not true for `TElem<Object, string>` for example, because `Object` is not a subtype of `Elem` and `op` is required to return an `Elem`.

GIDL also supports a *unqualified* or *universally qualified* generic types, similar to templates in C++ (e.g. `PriorityQueue3<A>`). This allows the instantiation to be any GIDL type. GIDL also supports type parameterized methods, common to all three mapped languages (e.g. as inner template function). The GIDL-level type checking and the language bindings necessary to implement this feature are similar to those for parametric polymorphism at the interface type level. However, a delicate problem arises when ensuring the correct invocation of such a method. Due to their static implementation of parametric polymorphism, both C++ and Java expect the method-level generics to be instantiated at the call site. In our case, the code is split between the caller and callee and separately compiled, thus the server has no way of knowing the type parameter instantiations. To handle this, we pass extra reflective-parameters that encapsulate the type-information of the generic type instantiations. The server-side generates code for a small method, which invokes the parameterized method on properly instantiated type-parameters, *just-in-time* compiles it and *links* it to the application. The *generated method* is finally called to complete the *original* invocation. The *generated methods* corresponding to different instantiations of the exposed type parameters can be cached for later reuse. A similar mechanism can be found in [44].

4.3.3 GIDL's Grammar and Consistency Checks

To provide syntax for parametric forms, we have modified the OMG IDL grammar as shown in Figure 4.3. Specifically, we have modified the derivation rule for the `scoped_name` non-terminal so that we can manipulate template types inside the GIDL specification (we can have sequence, arrays, structures, unions, interfaces, regular-values, etc. making use of generic types).

Figure 4.3 Adding support for parameterized interfaces to the IDL grammar

```
//...
<forward_dcl> ::= ["abstract"] "interface" identifier
                    ["<" <template_dcl> ">"];

<template_dcl> ::= <template_dcl_unit>
            | <template_dcl> "," <template_dcl_unit>
            ;

<template_dcl_unit> ::= <identifier> [{":"|":-"}
                          <scoped_name>];

<template_call> ::= <template_call_unit>
            |   <template_call> "," <template_call_unit>
            ;

<template_call_unit> ::= <const_type>
;

<scoped_name> ::= ["::"] <identifier>
                    ["<" <template_call> ">"]
            |     <scoped_name> "::" <identifier>
                    ["<" <template_call> ">"]
;

//...
```

We discuss a few details, with examples referring to the GIDL specification in Figure 4.4. We define the visibility scope of a generic type parameter to be throughout the interface in which it is defined. Following the same approach as in Generic Java [47, 66], we consider the subtyping to be invariant for parameterized types. For example, even if `Elem` is a subtype of `Object`, `Comp<Elem>` is not a subtype of `Comp<Object>`. In Figure 4.4, the type-checking of the `Comparator<Comp<B>, Comp<A>>` type (with mutual-recursive bounds) shall fail. This is because `Comp<B>` should extend `Comp<Comp<A>>` and, since the subtyping

Figure 4.4 Scopes and type-checking

```
1  interface Base<C> {
2    typedef struct BaseStruct {
3      Base<C> field;
4    };
5  };
6
7  interface Comp<A> : Base<A>{
8    void op1(in BaseStruct s);
9  };
10
11 interface Double : Comp<Float> {...};
12 interface Float : Comp<Double> {...};
13
14 interface Comparator<A : Comp<B>, B : Comp<A> > {
15   Base<B>::BaseStruct op2();
16   Comparator<Comp<B>, Comp<A> > op3(); // ERROR
17   Comparator<Double, Float> op4();     // OK
18 };
```

is invariant for parameterized types, this implies that `B` and `Comp<A>` are precisely the same type, which is not true. Using a similar reasoning, one will find that the `Comparator<Double, Float>` type is well-formed. Since the *export-based qualification* can be reduced to an *extend-based qualification* at GIDL level (see Section 4.4.2), the type checking mechanism in this case will be similar to the one presented above.

We turn now to the validity of the `op1`/`op2` operations of the `Comp`/`Comparator` interfaces. The `op1` method takes a parameter of type `BaseStruct`. The latter makes use of the generic type `C` and is defined inside the `Base` interface, which is a superclass of `Comp`. It follows that `BaseStruct` is also in the scope of `Comp`, its signature in this context, determined by up-traversing the inheritance tree of `Comp`, being `Base<A>::BaseStruct`. In the case of the `op2` method, all the information is stored inside the `scoped_name` of the returned type: `Base<B>::BaseStruct`.

We explicitly note that the *extension-based qualification* is stronger that the *export-based qualification*. For example, the GIDL specification below should generate a compile error.

```
interface Test0<C:Type1> { ... };
interface Test1<A:-Type1> : Test0<A> { ... };
```

This is because the type variable `A` in the `Test1<A>` scoped name is not required to extend `Type1`, as requested by the `Test0` definition, but only to implicitly implement its functionality.

4.3.4 Well-Formedness Type Rules

This section discusses the issues that arise from the combination of both named and structural subtyping in the definition of the qualification semantics. Figure 4.5 shows some of the type rules for well-formedness and subtyping in the presence of qualified type variables. We do not discuss the *unqualified* generic type, as its formal integration does not pose any challenges.

In this discussion, the metavariable X ranges over type variables; T, R and P range over types; N and O range over types other than type variables (non-variable types). I and m range over interface and method names respectively, while M ranges over method signatures. We write $\overline{X}$ as a shorthand for X_1,...,X_n and $\overline{X} \overline{\triangleleft} \overline{N}$ as a shorthand for $X_1 \triangleleft_1 N_1, ..., X_n \triangleleft_n N_n$. The length of the sequence $\overline{X}$ is $\#\overline{X}$ and we assume that the sequences of type variables contain no duplicate names. An interface table IT is a mapping from interface names to interface declarations. A type environment Δ

Figure 4.5 Type rules for two varieties of qualification

(Well-formed types ":" and ":-" qualifications)

$$\frac{IT(I) = interface\ \ I < \overline{X} \overline{\lhd} \overline{N} >: \overline{O}\{...\} \quad \lhd_i \in \{\ :\ ,\ :\text{-}\ \} \quad \Delta \vdash \overline{T} \quad \Delta \vdash T_i\ \ \bigtriangledown_i\ \ [\overline{T}/\overline{X}]N_i \quad \forall i \in \{1,..,\#\overline{X}\} \quad where\ \bigtriangledown_i = \ <:\ if\ \lhd_i = \ :\ and\ \bigtriangledown_i = \ <:\text{-}\ if\ \lhd_i = \ :\text{-}}{\Delta \vdash I < \overline{T} >}$$

(Named subtyping "<:")

$$\frac{IT(I) = interface\ \ I < \overline{X} \overline{\lhd}\ \ \overline{N} >: \overline{O}\{...\}\ \ \lhd_i \in \{\ :\ ,\ :\text{-}\ \}}{\Delta \vdash I < \overline{T} > \ \ <:\ \ [\overline{T}/\overline{X}]O_i \quad \forall i \in \{1,..,\#\overline{O}\}}$$

(Structural subtyping "<:-")

$$\frac{Methods(O_1) = \{M_{11},..,M_{1k}\}\ \ Methods(O_2) = \{M_{21},..,M_{2\ell}\}\ \text{where}\ \ \ell \leq k \quad \Delta \vdash O_1\ \ \Delta \vdash O_2 \quad \Delta \vdash M_{2i} \preceq M_{1i}\ \ \forall i \in \{1,..,\ell\}}{\Delta \vdash O_1 <:\text{-}\ O_2}$$

(Method inclusion "$\preceq$" – II)

$$\frac{M_1 = R_1\ \ m(\overline{P_1}) \quad M_2 =< \overline{X} \overline{\lhd} \overline{N} > R_2\ \ m(\overline{P_2}) \quad \lhd \in \{:\ ,\ :\text{-}\} \quad \exists \overline{T} \quad \Delta \vdash \overline{T} \quad \Delta \vdash \overline{P_1} = [\overline{T}/\overline{X}]\overline{P_2} \quad \Delta \vdash R_1 = [\overline{T}/\overline{X}]R_2}{\Delta \vdash M_1\ \ \preceq\ \ M_2}$$

(Method inclusion "$\preceq$" – III)

$$\frac{M_1 =< \overline{X_1} \overline{\lhd_1} \overline{N_1} > R_1\ \ m(\overline{P_1}) \quad M_2 =< \overline{X_2} \overline{\lhd_2} \overline{N_2} > R_2\ \ m(\overline{P_2}) \quad \Delta \vdash \overline{P_1} = [\overline{X_1}/\overline{X_2}]\overline{P_2} \quad \Delta \vdash R_1 = [\overline{X_1}/\overline{X_2}]R_2 \quad \lhd_1, \lhd_2 \in \{\ :\ ,\ :\text{-}\ \} \quad \Delta \vdash \overline{N_1}\ \ \overline{\psi}(\overline{\lhd_1}, \overline{\lhd_2})\ \ [\overline{X_1}/\overline{X_2}]\overline{N_2}}{\Delta \vdash M_1\ \ \preceq\ \ M_2}$$

$$\psi(\lhd_1, \lhd_2) = \begin{cases} \lhd_1 = \lhd_2 = : & then\ : \\ \lhd_1 = \lhd_2 = :\text{-} & then\ :\text{-} \\ \lhd_1 = : \ and\ \lhd_2 = :\text{-} & then\ :\text{-} \\ \lhd_1 = :\text{-}\ and\ \lhd_2 = : & then\ \eta\ where \\ \quad O_1 \eta O_2 = true\ if\ \{I | I <:\text{-} O_1\} \subseteq \{I | I <: O_2\}, & \\ \quad \text{and } false \text{ otherwise} & \end{cases}$$

is a finite mapping from type variables to pairs of bounds and qualification relation, written $\overline{X} \overline{\triangleleft} \overline{N}$ where $\triangleleft_i$ is one of the *extend* or *export* based qualifications. For brevity, some obvious rules are omitted from Figure 4.5: A type variable X is well formed in the type context Δ if it belongs to the domain of Δ. The type *Object* (the root of the IDL inheritance hierarchy) is well formed in any type context. Both subtyping relations are reflexive and transitive. Also, a type variable belonging to a type context is known to be in the corresponding subtyping relation with its bound.

The well-formedness rule in Figure 4.5 simply says that if the declaration of interface I begins with $interface I < \overline{X} \overline{\triangleleft} \overline{N} >$, then a type $I < \overline{T} >$ is well formed only if all the components of $\overline{T}$ are well formed and if, in addition, substituting $\overline{T}$ for $\overline{X}$ respects the bounds $\overline{N}$. Also, note that the simultaneous substitution enables recursion and mutual recursion between variables and bounds [24]. The *named subtyping* rule ("<:") in Figure 4.5 is also straight-forward: the inheritance hierarchy is dictated by the interface table IT.

Intuitively, the type-rule for structural subtyping ("<:-") says that O_1 is a structural subtype of O_2 if "it exports all the methods" of O_2. (IDL attributes are seen as a pair of methods: a getter and a setter). Note that O_1 and O_2 are instantiated types, in a given type context Δ. To formalize this property we introduced the *inclusion relation* ("$\preceq$") between methods. If M_1 and M_2 are not type parameterized then $M_1 \preceq M_2$ if the method names and signatures are identical. It follows in this case that also $M_2 \preceq M_1$.

Type-parameterized functions can be viewed as a set of functions: one for each different instantiation of their generic types. If M_2 is type parameterized ($\overline{X} \triangleleft \overline{N}$), but M_1 is not, then $M_1 \preceq M_2$ if the method names are identical and there exist a set of well-formed types $\overline{T}$ such that the substitution/instantiation $[\overline{T}/\overline{X}]$ applied on

M_2 yields a signature identical with that of M_1. The last case is when both M_1 and M_2 are type parameterized. Let us assume only one type parameter for M_1 and M_2: X_1 and X_2 respectively. (The generalization is straight-forward.) In order to have $M_1 \preceq M_2$ we need to have that the set of valid instantiation for X_1 is included in the set of valid instantiations for X_2. Assume an *extend-based qualification* $X_1 : O_1$ for X_1 and an *export-based qualification* $X_2 :\!- O_2$ for X_2. The set of interfaces that extend O_1 should be included in the set of interfaces that *implement* O_2 and the necessary and sufficient condition is $O_1 :\!- O_2$. A similar line of reasoning leads to the definition of the ψ operator in Figure 4.5. The last case leads to an overly technical result, which requires the type-checker to work hard. We prefer the more elegant alternative that excludes this case: if $X_1 :\!- O_1$ and $X_2 : O_2$ then M_1 is not $\preceq$-included in M_2.

4.3.5 GIDL to IDL Transformation

The implementation of our generic model employs a type erasure mechanism, based on the subtyping polymorphism supported by IDL. This preserves the interoperability between programs written over different implementations of the same software component architecture and allows our model to be easily adapted to enhance several software component architectures.

To achieve this, we constructed a translator from our GIDL to OMG IDL, accepting both regular IDL and GIDL specifications. When generating the IDL file, we first delete the generic type declarations from the GIDL file (delete the `template_dcl` productions in the GIDL grammar). Then the *unqualified*/*export-based qualified* type variables are substituted by the `any`/`Object` IDL type, while the *extend-based-qualified* ones are substituted by the (type variable erased) interface type they are supposed to extend.

Figure 4.6 The generated IDL specification

```
1  module GenericStructures{
2    // ...
3    interface PriorElem{
4      short getPriority();
5      short compareTo(in Object r);
6    };
7
8    interface PriorQueue1{
9      void enqueue(in PriorElem a);
10     PriorElem dequeue();
11     boolean empty();
12     short size();
13   };
14
15   interface PriorQueue2{
16     void enqueue(in Object a);
17     Object dequeue();
18     boolean empty();
19     short size();
20   };
21   // ...
22 };
```

The result should be a valid OMG IDL file, which can be compiled with a regular IDL compiler.

It is obvious that during this transformation we are losing the generic type information encapsulated in the GIDL specification. We recover this information by generating skeleton/stub wrapper classes in the target languages that make use of the specific characteristics of the parametric polymorphism in these languages. If we run the GIDL translator over the specification shown in Figure 4.1, it will generate the IDL specification in Figure 4.6.

4.4 High Level Ideas for Mapping Qualified Generic Types

Our generic type mechanism unifies the semantics of parametric polymorphism from different programming languages. In the implementation of our software tools we do as much work as possible at the unified level and in the GIDL to IDL translation, to minimize the language specific details.

4.4.1 Basic Ideas

Type-erasure for an *extend-based qualified* generic type is achieved by substituting it with the bounding-interface specified in the corresponding `template_dcl_unit` production. The Java and Aldor mappings are quite natural since this type of qualification is already supported. For the C++ language, due to its static binding time, the mapping can be achieved simply by casting an instance of the generic type to its corresponding qualifier. Note that this code is never executed at run-time, as shown in line marked "`//*`" in Figure 4.7.

We show below that the *export-based qualified* generic type can be reduced to an *extension-based qualification* relation at the GIDL level. The idea here is to find, for each *export-based qualified* generic type, all the possible interfaces that may (for a proper instantiation of their generic types, if any) implement the functionality of the associated qualifier.

The next step is to construct an interface that:

- implements the whole functionality of the qualifier (for a proper instantiation of its generic types, if any),

Figure 4.7 *Extend-based qualification* mapping to C++

```
1  // GIDL specification:
2  interface Foo { /*...*/ };
3  interface Test<T1:Foo> { /*...*/ };
4
5  - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
6
7  // C++ mapping:
8  template<class T1> class Test :
9  virtual public ::GIDL::GIDL_Object {
10   private:
11
12     virtual void implTestFunction() {
13       if(1) return;                  //*
14       T1 a_T1; Foo a_Foo = (Foo)a_T1;
15     }
16
17   public:
18
19     Test(::Test_var ob) {
20       implTestFunction(); //...
21     }//...
22 }
```

- becomes a "natural" parent for the interfaces identified in the previous step (in the sense that the inheritance does not actually introduce new functionality),
- defines a minimal number of generic types

We call the constructed interface the *most specific generic antiunifier* (*MSGA*) of the export-based qualification. The MSGA can be seen as the most specific antiunifier [56] or equivalently the least general generalization [53] of the types that satisfy the *export-based qualifier*.

Section 4.3.4 has already introduced and explained the GIDL type rules related to well-formedness and subtyping in the presence of qualified type variables. Next

Figure 4.8 MSGA example

```
1  interface Element{
2      tp0 op(in tp1 a1, in tp2 a2, in tp0 a3, in tp3 a4, in tp1 a5);
3  };
4
5  interface TemplEl1<T1, T2>{
6      T1  op(in T2 a1, in tp2 a2, in T1 a3, in tp3 a4, in T2 a5);
7  };
8
9  interface TemplEl2<T1, T2, T3>{
10     T1  op(in tp1 a1, in T2 a2, in T1 a3, in tp3 a4, in T3 a5);
11 };
12
13 interface Test<A:-Element> {
14     //use A ...
15 };
```

we discuss the main stages involved in the *MSGA* construction and we present an example. This *MSGA* could be used as the erasure type for its corresponding generic type. We have chosen not to do so, however, due to CORBA's IDL limitations and we use `Object` instead. The second subsection concludes the chapter.

4.4.2 Mapping the Export-Based Qualification

The algorithm for computing the *MSGA* associated with an *export-based qualification*, presented here, works under the assumption that the *extend-based qualification* has already been mapped to the target language. Each GIDL-interface that may satisfy the *export-based qualification* in certain circumstances (for a given instantiation of the generic type for example), shall be made to implement the *most specific generic antiunifier* (*MSGA*) interface associated with that *export-based qualification*.

As an example, consider the GIDL file shown in Figure 4.8. The `Test` interface uses an *export-based qualified* generic type. Among the valid candidates for the type instantiation one can list:

`Element, TemplEl1<tp0, tp1>, TemplEl2<tp0, tp2, tp1>.`

Being given the methods in the `Element` interface and the set of interfaces defined in a GIDL specification, our task is to construct the most specific generic antiunifier (*MSGA*) of these candidates. First, we construct a new parameterized interface, with as many generic types as the number of parameters in all the methods of the "to be implemented" interface, plus the number of methods, as the return types should also be taken into account. In our example, the *MSGA* initially looks like:

```
interface MSGA<G0, G1, G2, G3, G4, G5> {
  G0 op(in G1 a1, in G2 a2, in G3 a3, in G4 a4, in G5 a5);
}
```

Left like this, the interface created can make use of many different generic types, so we may want to simplify it. We create a matrix as below, in which the types that have to match will share the same column. If there is an interface that we can prove cannot implement the required functionality, it should not appear in the matrix.

G0	G1	G2	G3	G4	G5	MSGA
tp0	tp1	tp2	tp0	tp3	tp1	Element
T1	T2	tp2	T1	tp3	T2	TemplEl1
T1	tp1	T2	T1	tp3	T3	TemplEl2

The first thing to do is to identify the columns formed by the same non-generic type. This occurs in `G4`'s column in the above table. The next step is to remove the

Figure 4.9 The result of the MSGA algorithm

```
1  interface MSGA<G0, G1, G2, G5> {
2    G0 op( in G1 a1, in G2 a2, in G0 a3, in tp3 a4, in G5 a5 );
3  }
4
5  interface Element: MSGA<tp0,tp1,tp2,tp1> {...};
6
7  interface TemplEl1<T1, T2>: MSGA<T1,T2,tp2,T2> {...};
8
9  interface TemplEl2<T1, T2, T3>: MSGA<T1,tp1,T2,T3> {...};
10
11 interface Test<A : MSGA<tp0, tp1, tp2, tp1> > { //use A...};
```

corresponding generic type from the template declaration part of the *MSGA* interface and substitute it with the non-generic type throughout the *MSGA*'s interface definition. In our example this would be substituting `tp3` for `G4`. A second simplification can be made if two columns are found to be equal. This occurs with columns 0 and 3 of our example. In this case we can also remove one of the generic types in the template declaration part of the *MSGA* interface and substitute it with the other generic type throughout the interface definition. Special care should be taken for the `void` return type, since it cannot be matched by any generic type instantiation.

Finally, all the interfaces found to be valid candidates to instantiate the *export-based qualified* generic type, are made to implement the simplified *MSGA* interface, as shown in Figure 4.9. It is clear that only `Element`, `TemplEl1<tp0, tp1>` and `TemplEl2<tp0, tp2, tp1>` will not be signaled with a compiler error when substituted for the generic type `A` in the `Test` generic interface. Notice also that the *MSGA* is using only *unqualified* type parameters in order to cover all possible type instantiations and that the generic type qualifications of the candidate interfaces (`TemplEl1`, `TemplEl2`) do not influence the algorithm in any way.

Figure 4.10 More MSGA issues

```
1  //A. GIDL specification//
2
3  // Eg. 1
4  interface Type1< A:-Type1<A> > {...};
5  interface Type2< B:-Type2<B> > : Type1<B> {...};
6
7  // Eg. 2
8  interface Elem< C > {...};
9  interface Test1< D:-Elem<D> > {...};
10 interface Test2< E:-Elem<E> > {...};
11
12
13 //B. MSGA constructs for the GIDL specification in A.//
14
15 // Eg. 1
16 interface MSGA1< A > {...}; //A:-Type1<A>
17 interface MSGA2< B > : MSGA1<B> {...}; //B:-Type2<B>
18 interface Type1< A: MSGA1<A> > : MSGA1<A> {...};
19 interface Type2< B: MSGA2<B> > : Type1<B>, MSGA2<B> {...}; //***
20
21 // Eg. 2
22 interface MSGA3< T > {...}; //D:-Elem<D> and E:-Elem<E>
23 interface Elem < C > : MSGA3<C> {...};
24 interface Test1< D: MSGA3<D> > {...};
25 interface Test2< E: MSGA3<E> > {...};
```

Type parameterized functions are accommodated in a straightforward manner in the algorithm presented. Section 4.3.4 has provided the details: If at least one type instantiation of a function satisfies the signature of another function that appears in the export-based qualifier, then we consider that the type parameterized function satisfies the qualifier's function. Conversely, if the export-based qualifier exports a type parameterized function, then only another type parameterized function will satisfy it and only if its set of valid type instantiations includes the one of the qualifier's function.

There are two additional points to mention with respect to MSGAs. Figure 4.10 presents a legal GIDL specification, together with its corresponding MSGA bindings. The first example in Figure 4.10 shows that we must preserve the inheritance hierarchy among MSGAs. If this were not done, the compiler would find an error while checking the correctness of the `Type1<B>` type in line 19. The `B` bound is `MSGA2<B>`, but `B` should also be bounded by `MSGA1<B>` from the definition of `Type1` in line 18. If no inheritance relation were defined among `MSGA2` and `MSGA1` interfaces, a compile-time error would be signaled.

In order to keep the number of generated MSGAs to a minimum, a simple unification algorithm is employed among *export-based qualification* relations. The second example in Figure 4.10 shows that only one MSGA (`MSGA3`) is constructed for the `D` and `E` *export-based qualifications* (lines 23, 24).

4.4.3 Discussion

We should remember that during the GIDL to IDL translation, we are loosing the generic information present in our GIDL specification. We recover the lost information by generating wrapper classes corresponding to the constructs in the GIDL specification. We use the *MSGA* interfaces as a general approach for mapping the *export-based qualification* to both the C++ and Java programming languages (once we have implemented the mapping for the *extend-based qualifications*). They are also used at the GIDL level for type checking. One can notice that the *MSGA* construct introduces little run-time overhead, as it is used in the type-checking phase at compile time. The generated *MSGA* interfaces may look ugly, involving many generic type parameters. This is not a concern since their use is transparent to the user, as their

Figure 4.11 Incorrect C++ mapping of the *export-based qualification*

```
1  template<class T1> class Test : virtual public ::GIDL::GIDL_Object
       {
2
3    private:
4      virtual void implTestFunction() {
5        if(1) return;
6        T1 a_T1; tp1 var1; tp2 var2; tp3 var3; tp0 var0;
7        var0 = a_T1.op(var1, var2, var0, var3, var1);
8      }
9
10   public:
11     Test(::Test_var ob) {
12       implTestFunction(); //...
13     }
14
15   //...
16 }
```

only task is to ensure a correct translation of the GIDL semantics to the language bindings.

At a first sight, it might appear that in the case of the C++ mapping, an easier solution can be found for translating the *export-based qualification*. Namely, the `A:-Element` in Figure 4.8, can be **wrongly** mapped as in Figure 4.11.

This calls all the qualifier's functions on the generic type object. This translation is not consistent with the *export-based qualification* semantics, however, as the generic type instantiation may export a method that is a subtype of the qualifier's method (in the usual functional lattice) and type-checking will succeed when it should not.

It might be desirable to have functional subtyping encapsulated in the *export-based qualification* semantics, especially since one could then easily map the semantics of functional subtyping through generics. In this case the code above will do. We are still investigating this but it appears to be difficult: If the generic model assumes

invariant parameterized types subtyping, the current MSGA algorithm is insufficient. Conversely, if covariant and contravariant parameterized type subtyping is assumed, the MSGA algorithm can be made to work, but enforcing the correctness of the GIDL semantics in the mapped languages complicates the language bindings and the user interaction with the framework. This discussion is beyond the scope of this thesis. At present, as described, our choice has been for invariant subtyping.

4.5 The GIDL Base Application Architecture

We now present a high level view of our GIDL architecture: that is how the architecture components are created and how they interact to accomplish an invocation successfully. We then show how a programmer may use our architecture. We argue the transparency of our design, in the sense that the programmer need not know the internal architecture, but only the mapping rules from GIDL to a specific programming language.

4.5.1 The GIDL Extension Architecture

Figure 4.12 illustrates the design of our proposed architecture. The circles stand for user's code. The rectangular boxes represent components in the standard OMG-CORBA architecture. This includes the IDL specification, the stub and skeleton, and the object request broker (ORB). The hexagons represent the components needed by our generic extension, including the GIDL specification and generated GIDL wrappers. The dashed arrows represent the *compiles to* relation among components. A GIDL specification compiled with our GIDL compiler will generate an IDL specification file,

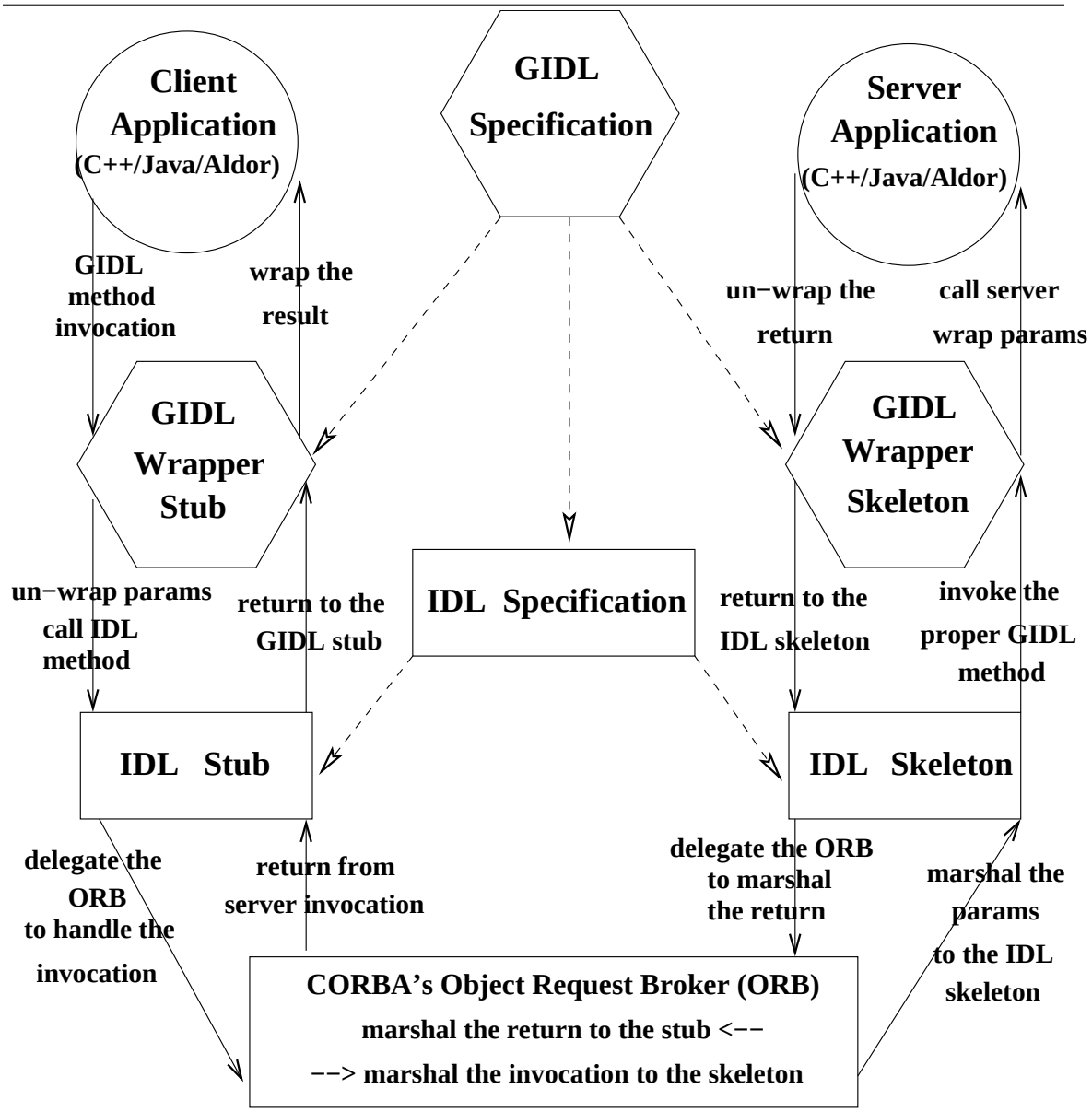
together with GIDL wrapper stub and skeleton bindings, which recover the lost generic type information.

The bottom part of the figure represents CORBA's internals. When compiling the IDL file with any vendor's IDL compiler, client stubs and skeletons will be generated and these serve as proxies for clients and servers respectively. Because the IDL defines interfaces so strictly, the stub on the client side will have no trouble matching perfectly with the skeleton on the server side, even if the two are compiled to different programming languages, or are running on different ORBs from different vendors, under different operating systems or hardware [49].

The solid arrows in Figure 4.12 depict method invocation. In CORBA, every object has its own unique object reference. The client must obtain an object's reference in a string representation. This is used by the ORB to identify the exact instance that must be invoked. As far as the client is concerned, it is invoking a method on the object instance. However, it actually calls the IDL stub that acts as a proxy and forwards the invocation to the ORB. It is the ORB's job to find the server, to pass the parameters, make the invocation and eventually to return a result to the client [49].

As stated previously, our generic extension for CORBA introduces an extra level of indirection in the original mechanism; in order to recover the generic type information lost by the GIDL to IDL transformation, stub and skeleton wrappers are generated to match the original GIDL specification. Basically, for every type in our GIDL specification, we construct C++/Java/Aldor wrapper stubs that reference the CORBA-stub objects generated by the IDL compiler. When the client invokes an operation, it actually calls a method on a GIDL stub wrapper object. The GIDL method implementation retrieves the CORBA-objects hidden by the wrapper-objects taken as parameters, in-

Figure 4.12 GIDL architecture for CORBA
circle – user code
hexagon – GIDL *component*
rectangle – CORBA *component*
dashed arrow – is compiled to
solid arrow – method invocation flow



vokes 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. The wrapper skeleton method encapsulates the erased IDL objects with generics erased as GIDL ones, adding back the generic type's erased information. It invokes the user-implemented server method with these parameters, retrieves the CORBA IDL-object or value from the returned object and passes it to the IDL skeleton.

Clearly, for our implementation to be CORBA compliant, CORBA's *Interface Repository* (IR) model would have to be changed to handle parameterized interfaces. Two new IR–IDL interfaces for `TemplateDclUnit` and `TemplateCallUnit` extending the `IRObject` interface should be added to the IR meta model and the `InterfaceDef` IR-IDL interface should be modified to contain a sequence of `TemplateDclUnit` and a list of `TemplateCallUnit`. The definition of `ScopedName` would also have to be made to deal with templates. The `TypeCodes` and the string representation of references would also be extended to contain parameterized type information.

But as we stated earlier, *it is not our goal to write a* CORBA *compliant extension of* IDL. Thus a detailed investigation of the changes that would have to be done at the CORBA interface repository level is orthogonal to the goal of this thesis. We interested exclusively in adding genericity to IDL; the OMG's IR is a mirror of IDL's specifications, thus the same ideas apply.

A more delicate problem is modifying a software component architecture's run-time to deal with dynamic invocation on parameterized methods, as by the use of generics the number of GIDL types is potentially infinite. This is handled by run-time re-compilation techniques, similar to that described in [44]. We start with a set of can-

didates for the instantiation of the generic method and expand this set incrementally as invocations with different instantiation for generic types occur.

With minimal modifications to the wrapper code generation, our generic extension architecture can sit on top of other software component architectures such as DCOM or JNI. Targeting DCOM is straight-forward, as its design is similar to CORBA.

Enhancing JNI is more subtle: Given a GIDL specification file, wrapper stubs are generated on the C++ and Java sides. These make use of parametric polymorphism and will ensure that the GIDL semantics are statically enforced in both mappings, similar to our design for CORBA. What differs is the implementation of the erased stub (*IDL stub* box in Figure 4.12). On the C++ side, this corresponds to the mechanism provided by JNI to invoke the JVM; it can be mangled inside the wrapper classes and hidden from the user. To call Java code from C++, the C++ parameterized wrapper classes use the JNI mechanism to invoke, through JVM, the parameterized Java wrapper classes. To call C++ from Java, the parameterized Java wrapper classes, containing only native methods, are compiled ("javah" utility) and, as a result, the C++ generic erased stub is generated. The latter re-directs the invocation to the parameterized wrapper class.

In summary, the generic extension for our CORBA case study can be applied on top of any CORBA-vendor implementation, while maintaining backward compatibility with standard CORBA applications. Moreover, with minimal changes, our architecture can be applied to various heterogeneous systems. Our approach has been to design a general and clean extension architecture and then to apply aggressive optimization techniques to reduce the overheads incurred by casting, and the extra indirection in invocation. One can anticipate that a combination of optimizations, including

Figure 4.13 GIDL code for a simple priority queue

```
1  interface PriorElem {
2     short getPriority();
3     short compareTo(in Object r);
4  };
5
6  interface PriorQueue2<A:-PriorElem> {
7     void    enqueue(in A a);
8     A       dequeue();
9     boolean empty();
10    short   size();
11    A       creatNewA(in short s);
12 };
```

pointer aliasing, scalar replacement of aggregates, copy propagation and dead code elimination, will achieve this in most cases.

4.5.2 The User's Perspective

Consider the GIDL specification shown in Figure 4.13. When implementing the server side, the programmer should extend the generated skeleton wrapper classes `PriorQueue2` and `PriorElem`, implementing the operations that appear in the GIDL specification. This is the usual CORBA procedure for writing servers, so the user will find no difficulty here.

An excerpt from a C++ client program that makes use of the types defined in this GIDL specification is shown in Figure 4.14. Suppose the server is represented by a `GIDL::PriorQueue2<GIDL::PriorElem>` object. Suppose the server is represented by a `GIDL:: PriorQueue2<GIDL::PriorElem>` object. The client obtains a string representation of a reference to the generic type erased object, i.e. `::PriorQueue2` from the server (line 1). It creates a generic wrapper stub (line 2) together with an IDL

Figure 4.14 Code excerpt from a C++ client

```
1  CORBA::Object_var obj = orb->string_to_object(s);
2  GIDL::PriorQueue2<GIDL::PriorElem> gpq(pq_orig);
3  GIDL::PriorElem gPEobj = gpq.createNewA(GIDL::Short_GIDL(1));
4
5  gpq.enqueue(gPEobj); // OK
6
7  // Obtain a reference to a CORBA::Object - obj ...
8
9  gpq.enqueue(obj);  // Error
10
11 GIDL::PriorElem gPEobj = gpq.dequeue();
12 GIDL::Short_GIDL sh    = PEobj.getPriority();
13
14 cout<<sh<<endl; //prints "1"
```

stub proxy. The latter is implemented inside the wrapper class constructor to hide the internal architectural design. From this point on, the user can transparently invoke the server functionality (lines 5, 11, 12). In Figure 4.14, `GIDL::Short_GIDL` is the C++ mapping type for GIDL's `short`. Line 9 generates a compile-type error, signaling the user that his code does not obey the GIDL specification semantics. If we look at the GIDL specification in Figure 4.13, the `enqueue` operation is supposed to take a parameter of type `A`. In our case the parameter is substituted by `GIDL::PriorElem`, since we are working with `GIDL::PriorQueue2<GIDL::PriorElem>`. Therefore the parameter of the `enqueue` function is expected to be of type `GIDL::PriorElem` and not `CORBA::Object`.

To conclude, our architecture places little burden on programmer's shoulders, as most of our implementation details are hidden. The steps in application design are the same as those required for a standard CORBA application, but now the implementation can use generic programming. The details of the language bindings for C++, Java and Aldor are given in the next section.

4.6 Language Bindings

In Section 4.5 we discussed in general terms the high-level ideas involved in the design of our framework. Now, by seeing the mapping specifics, we complete the description of the overall architecture. There are two reasons for presenting aspects related to the language bindings: First, as the targeted languages cover a wide-range of parametric polymorphism semantics and binding time models, it is informative to understand how the mappings work. Second, it is of practical interest to mention some of the less obvious details that are important in achieving an effective mapping.

We do not give a formal proof for the correctness of the translation schemes for the language bindings. This would be a tedious task as none of the targeted languages, to our knowledge, have a complete formal model. An approach similar to that adopted for *Featherweight GJ* [24], working with only a small functional subset of the languages considered, might be used to prove that our *extension* does not introduce any run-time errors.

4.6.1 GIDL to C++ Mapping

This section describes how the stub and skeleton wrappers, presented in the previous section as high level components of our architecture, are implemented when the targeted language is C++. We first introduce the high-level mapping ideas, including the correspondence between GIDL and C++ types. We then elaborate on the wrapper object-model and on the C++ implementation of GIDL's export- and extend-based qualifications.

4.6.1.1 High-Level Mapping Ideas

The mapping from GIDL to C++ is for the most part quite easy and natural, as the IDL syntax and semantics are quite close to those of C++. We closely follow the same conventions used in the standard IDL to C++ mapping, so the user will not feel any major conceptual difference when using our generic architecture.

GIDL modules are translated into C++ namespaces; GIDL interfaces into C++ (possibly template) classes, encapsulating all the functions that appear in the GIDL interface together with getter and setter functions for every attribute in the GIDL interface. A GIDL structure is mapped to a C++ class, with setter and getter functions for each field in the GIDL structure. GIDL basic types (`short`, `long`, *etc*) are mapped to corresponding C++ types, providing the expected functionality by means of operator overloading. GIDL's arrays and sequences are mapped by type instantiating a C++ generic array/sequence class in which the "[]" operator is overloaded. In our implementation, the relation between the wrapper objects and the associated CORBA-objects is many to one: There can be several wrappers storing the same CORBA-object. Memory management is simple, creating our wrapper objects on the stack only. Thus there is no need for explicit de-allocation.

Our GIDL-C++ stub and skeleton wrappers are encapsulated within the "GIDL" and "GIDL_implem" namespaces. The inheritance hierarchy at the GIDL specification level is preserved in the C++ mapping. GIDL scopes directly create C++ scopes, as the C++ semantics allows the definition of nested classes. A side-effect of this is that the generic types defined by a generic GIDL interface stay in the same position after the C++ translation and do not create generic type duplicates for the nested GIDL structures (as happens in the Java mapping case).

Figure 4.15 Nested structures

```
1  // GIDL:
2  interface GenericInterf<A> {
3     struct GenericStruct {
4        typedef A A_array[5][5];
5        A_array field;
6     };
7  };
8
9  - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
10
11 // C++:
12 template<class A> class GenericInterf: GIDL::GIDL_Object{
13    struct GenericStruct : GIDL::GIDL_Object {
14       typedef Array_GIDL<...,A,...> A_array;
15       public: A_array field;
16       //...
17    } //...
18 } //...
```

In the example shown in Figure 4.15, a GIDL specification containing a structure type nested inside an interface type is similarly translated to C++ as a nested definition of classes. The generic type parameter `A` is shared inside the nested scope.

Our wrapper objects, no matter what GIDL type they represent, can be seen as an aggregation of a reference to the erased CORBA value they represent, the generic type information associated with them and the casting functionality they define. They also inherit the functionality provided by the corresponding GIDL type. This is similar to the "reified type" pattern of Johnson [25], where objects are used to carry type information or to some uses of dependent product types.

4.6.1.2 Wrapper Stub Object Model

Figure 4.16 shows A piece of the generated wrapper stub for the following GIDL specification.

```
interface Foo { /*...*/ };
interface Test<T1:Foo, T2:-Foo, T3>
{ Foo op(in T1 t1, in T2 t2, in T3 t3, in Foo f); };
```

As stated in Section 4.6.1.1, the type casting functionality is common to all the stub wrapper types. This is represented in Figure 4.16 by the `_lift` and `_narrow` methods. The `_lift` method returns a new instance of the wrapper class encapsulating the CORBA-object received as parameter, while `_narrow` returns the CORBA-object encapsulated by the wrapper object. The `_any_lift` and `_any_narrow` functions have a similar functionality, but they are used in conjunction with the CORBA::Any type, our erasure for the *unqualified* generic type.

The implementation of the `op` function (in Figure 4.16) illustrates the method invocation mechanism. All the wrapper objects received as parameters are unboxed to IDL stub objects. Following the type erasure rules, a wrapper interface type object is unboxed to the CORBA stub object it encapsulates (line `//6`), a *unqualified* generic type is erased to the `CORBA::Any_var` type (line `//5`), an *extend-based qualified* generic type is unboxed to the IDL-stub type associated with its qualifier (line `//3`) and finally an *export-based qualified* generic type is erased to the `CORBA::Object` type (line `//4`). The IDL stub method is invoked on the object reference that our wrapper encapsulates (line marked 7) and finally the returned CORBA stub object/value is boxed inside a stub wrapper object and is returned to the client application (line `//8`).

Figure 4.16 Excerpt of C++ wrapper stub code

```
1  template<class T1,class T2,class T3> class Test :
2                        virtual public ::GIDL::GIDL_Object {
3    protected: ::Test_var* obj;
4    private:
5      virtual void implTestFunction() {
6        if(1) return;
7        T2 a_T2; MSGA_Foo msga = (MSGA_Foo)a_T2;      //1
8        T1 a_T1; Foo a_Foo = (Foo)a_T1;               //2
9      }
10
11   public:
12     Test(::Test_var ob) {
13       obj = new ::Test_var(ob);
14       implTestFunction();
15     } //...
16
17     ::Test_var getOrigObj()     { return *obj; }
18     void setOrigObj(::Test_var o) { *obj = ::Test::_duplicate(o); }
19     static ::Test_var _narrow( Test<T1,T2,T3> corba_obj_TIDL) {
20       return *corba_obj_TIDL.obj;
21     }
22     static Test<T1,T2,T3> _lift(CORBA::Object_var corba_obj_TIDL){
23       return Test<T1,T2,T3>( ::Test::_narrow(corba_obj_TIDL) );
24     }
25
26     static Test<T1,T2,T3> _any_lift(CORBA::Any_var a) { /*...*/ }
27     static CORBA::Any_var _any_narrow(Test<T1,T2,T3> w) { /*...*/ }
28     //...
29
30     virtual GIDL::Foo op(T1 a1_G,T2 a2_G,T3 a3_G,GIDL::Foo a4_G) {
31       ::Foo_var a1 = a1_GIDL._narrow(a1_G);             //3
32       CORBA::Object_var a2 = a2_GIDL._narrow(a2_G);     //4
33       CORBA::Any_var a3 = a3_GIDL._any_narrow(a3_G);    //5
34       ::Foo_var a4 = a4_G._narrow(a4_G);                //6
35       ::Foo_var a0_GIDL = (*obj)->op(a1, a2, a3, a4);   //7
36       GIDL::Foo retGIDL; return retGIDL._lift(a0_GIDL); //8
37     }
38 };
```

The last thing to note here is the C++ mapping of GIDL's export- and extend-based qualifications for generic type parameters. We remind the reader that C++ does not support restrictions on generic type parameters. We achieve this through the `implTestFunction()` function (in Figure 4.16), which is called from the wrapper class constructors.

As discussed in Section 4.4.1, our implementation relies on C++'s static binding. In the case of the *extend-based qualified* generic type, a simple cast to the qualifier's type suffices (shown on line `//2`). This enforces the condition that the substituted type has to inherit from the qualifier (`GIDL::Foo` in our case). The mapping of an *export-based qualification* requires the construction of the *MSGA* associated with that generic type declaration, as discussed in Section 4.4. The generic type instantiation is valid if the cast to the associated *MSGA* succeeds (line `//1` in Figure 4.16). Otherwise a compile-time error is generated during type checking.

Our mapping of parameter qualifications adds no run-time overhead, as our verification code (lines `//1` and `//2`) follows the statement `if(1) return;` so is never reached. After the type-checking phase is completed, any reasonable compiler will discover this and all the calls to `implTestFunction()` will be eliminated.

4.6.2 GIDL to Java Mapping

The Java mapping follows the same main lines as the C++ mapping. We create wrappers objects that encapsulate CORBA object references and recover the generic type information lost during the GIDL to IDL erasure transformation. We follow the same translation rules defined in the standard IDL to Java mapping. The GIDL inheritance hierarchy is translated to a corresponding inheritance hierarchy among

Java interfaces, the root of the hierarchy being the `GIDL_Value_Interf` interface. We do this because Java classes do not support multiple inheritance.

One drawback of the Java mapping is that it requires the user's help. Java does not support object instantiation of a generic type parameter, e.g. `new A()`. Neither does it provide 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. This is needed because otherwise we cannot enforce an exact boxing/unboxing mechanism between our wrapper objects and stub objects. The *virtual call* on such an object will invoke the correct boxing/unboxing function for the instantiated type, otherwise the `lift/narrow` methods will be called on the Java erased type and this is not correct.

We omit the implementation details and touch only upon the constructs that are mapped in a conceptually different manner than in the C++ case. The remainder is similar to the C++ mapping. We focus on the Java mapping of the implicitly parametric structures, that is GIDL structures that are nested in the scope of a generic interface, and that use some of the interface's generic type parameters. An example of such structure is the following:

```
interface Base<C: Object, D, E>{
  typedef struct BaseStruct{
    C field_c;   E field_e;
  };
};
```

Since we have defined that the scope of a generic type parameter is throughout the interface in which it is declared, the example is perfectly legal GIDL code. In order to

Figure 4.17 Java wrapper stub mapping

```
1  package GIDL.Base; import GIDL.*;
2  public final class BaseStruct
3      <C extends GIDL.GIDL_Object, E extends GIDL.GIDL_Value_Interf>
4      implements GIDL.GIDL_Value_Interf {
5
6    private org.omg.CORBA.Object obj; //the encapsulated CORBA object
7    private C c;       private E e;
8
9    public BaseStruct(C c, E e, org.omg.CORBA.Object ob) {
10       obj = ob;   this.c = c;   this.e = e;
11   }
12   public BaseStruct(C c, E e) { this.c = c; this.e = e; }
13
14   public BaseStruct<C, E> lift(org.omg.CORBA.Object b)
15   {    return (new BaseStruct<C, E>(c, e, b)); }
16
17   public Base.BaseStruct narrow(BaseStruct<C, E> t) {return t.obj;}
18
19   public BaseStruct<C, E> any_lift(org.omg.CORBA.Any a) {
20       try{
21           Base.BaseStruct ob = Base.BaseStructHelper.extract(a);
22           return (new BaseStruct<C, E>(c, e, ob));
23       }catch(Exception exc){ /* ... */ }
24   }
25
26   public org.omg.CORBA.Any any_narrow(BaseStruct<C, E> o){
27       try{
28           org.omg.CORBA.Any a = orb.create_any();
29           Base.BaseStruct bb = o.obj;
30           Base.BaseStructHelper.insert(a, bb);
31           return a;
32       }catch(Exception exc){ /* ... */ }
33   }//.....
34
35   public C get_field_c()     { return (C)c.lift(obj.field_c); }
36   public void set_field_c(C co) { obj.field_c = c.narrow(co); }
37   public E get_field_e()     { return (E)e.any_lift(obj.field_e); }
38   public void set_field_e(C eo) { obj.field_e = e.any_narrow(eo); }
39 }
```

perform the mapping, we need to know which are the generic parameters used in the structure definition, and also any constraints that apply to them. The Java mapping for the `BaseStruct` parameterized structure, presented above would be that shown in Figure 4.17.

As we have seen with the C++ mapping, each wrapper stub class implements two methods: `lift` and `narrow`, which are used to encapsulate and retrieve a CORBA-object. However, since Java does not support any run time information with respect to type variables, we cannot declare the `lift` and `narrow` methods statically. We ask the user to provide a trivial object for each type variable in the declaration of an interface. This allows dynamic creation of new instances of the variable type using *virtual calls* to `lift`, `any_lift` on the *trivial* objects. The `any_lift` and `any_narrow` methods are similar to `lift` and `narrow` and are used for the unqualified generic types (as their erasure is the IDL `any` type). In addition, the GIDL wrappers provide an implementation for each method in the declaration of the corresponding GIDL interface and for any the `get` and `set` methods corresponding to fields in the structure definition.

4.6.3 GIDL to Aldor Mapping

Aldor was not one of the programming languages for which CORBA provides standard mappings. Michael Loyd has conducted the work of linking Aldor to the CORBA framework. We mention here the main mapping ideas, in which I was also involved. As usual we avoid implementation details and keep the discussion at a high level.

Both GIDL and Aldor provide a set of low-level types, for example fixed size integers and floating point numbers, strings, etc. The correspondence between these low-level

types is straightforward. We use Aldor's *ex post-facto* domain extension on the basic types to extend them with the functionality needed by our framework. That is, the existing domains are extended to satisfy new type categories supporting GIDL in an aspect-oriented manner.

A GIDL interface is mapped to an Aldor domain/category pair. The category specifies the exports present in the GIDL interface, together with the casting functionality needed to link it to the CORBA environment and the domain provides the implementation. Because Aldor is not based on classes of objects the mappings of the exported operations all receive one extra parameter corresponding to the implicit "self" parameter of the GIDL methods. Multiple inheritance among GIDL interfaces is matched by multiple inheritance among the Aldor proxy categories. The `Join` operation on categories is used when multiple inheritance is required. Inner GIDL structures and interfaces are directly mapped into inner Aldor domains. Aldor directly supports both types of qualifications present in the GIDL model for generics.

It is interesting to notice that languages with dynamic bindings (like *Java*) support parametric polymorphism through a qualification mechanism (otherwise an improper instantiation of the generic type will lead to a run-time error and not to a compile-time as preferred). Aldor is not an exception from this rule and it directly supports both types of qualifications present in the GIDL model for generics.

Figure 4.18 provides an example of these ideas. Note that GIDL *export-based qualification* is directly mapped to Aldor by means of a type parameter qualified by an un-named category. This is specified by means of the "`with`" Aldor construct, which implies that a specific list of exports need to be provided by the type parameter.

The Aldor mapping easily accommodates type-parameterized functions (no need

Figure 4.18 Mapping GIDL qualifications to Aldor

```
1  // GIDL specification:
2  interface Monomial<R: Ring> : Ring, Module<R> {
3      Monomial<R> *(R r, Monomial<R> mon);
4  }
5  interface Comp<A> { boolean compare(A a); }
6  interface Comparator<A:-Comp<B>, B:-Comp<A> > {...}
7
8  - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
9
10 -- Aldor stubs:
11 define MonomialCat(R:Ring) : Category ==
12 Join(RingCat, ModuleCat(R)) with {
13     *: (R, %) -> %;
14 }
15 Monomial(R:Ring): MonomialCat(R) == add {
16     -- ...
17     (r:R) * (poly:%) : % ==
18     -- CORBA remote invocation of the server ...
19 }
20 define ComparatorCat(
21     A: with{compare:(B)->Boolean},
22     B: with{compare:(A)->Boolean} ): Category ==
23 with {
24     -- ...
25 }
```

for recompilation techniques), as Aldor supports types as first class values. (Types can be passed as parameters to functions and are constructed at run-time.) Since, given a GIDL specification, the number of type constructors is finite, it is possible to write a function that receives as parameter a formal description of a type, and its implementation constructs that type and returns it. The resulting type is passed as parameter to the original type-parameterized function.

4.7 GIDL and Library Translations

This section explores how our generic model and architectural design may be used to expose facilities from a language to a multi-language environment and discuss our mappings in the context of automatic library translation.

Section 4.7.1 presents an experiment in which we have translated part of the C++'s Standard Template Library (STL) functionality into a GIDL server. We conclude that GIDL is able to express the STL semantics and, furthermore, that the library-translation process can be automated. As the GIDL and Aldor languages are very different, Section 4.7.2 investigates the high-level ideas involved in mapping the semantics of an Aldor library to a GIDL specification and finds that GIDL is effective in rendering these semantics. The Aldor libraries are sophisticated mathematical libraries for exact algorithms for linear and non-linear algebra and they use a high-density of complex type constructors. It is not uncommon for the mathematical types expressed in Aldor to be several levels deep, as in:

```
Polynomial(Matrix(Complex(Fraction(Integer))))
```

Furthermore, libraries typically have relationships between type parameters, declared, e.g., as:

```
SimpleAlgebraicExtension(R: CommutativeRing,
                         P: UnivPolynomCategory(R),
                         p: P): Algebra(R) == ...
```

With their rich nature, the Aldor BasicMath library and the C++ Standard Template Library make two ideal candidates for experimentation.

4.7.1 Accessing the C++ STL in a Multi-Language Environment

We now describe an experiment in which we have exposed the C++ Standard Template Library to a multi-language setting. This tests both our generic model design, as we are able to express STL's fundamental requirements at the GIDL's level and our architectural model as we were able to translate it with minimal programming effort.

4.7.1.1 Key Features in the Design of STL

STL [17] comprises six major kinds of components: containers (e.g. vectors, lists), generic algorithms (e.g. find, merge, sort), iterators, function objects (classes containing one method that overloads the `()` operator), adaptors (components that modify the interface of another component), and allocators (memory management encapsulation).

One can identify two key differences between STL and many the other C++ libraries: First, STL containers are not built within an inheritance relation — they are not assumed to be derived from some common ancestor. Second, STL components are designed to be *orthogonal*, unlike traditional container class libraries where algorithms are associated with classes and implemented as member methods. This keeps the source code and documentation small, as for m containers and n algorithms that are applicable to all m containers, only n generic algorithms have to be written and not $m \times n$. On the other hand, the components orthogonal structure addresses the extensibility issue, as it allows user's algorithms to work with the library's containers, or user's containers to work with the library's algorithms. The orthogonality between the algorithm domain and the containers domain is achieved, in part, by the use of

iterators; the algorithms in STL are not specified in terms of the data structure on which they operates, but in terms of iterators, which are data structure independent.

However, because of performance guarantees, it might be not possible to plug together any algorithm with any container: For example an efficient *generic* sort algorithm may require random access to the data and in the *list*'s case this is not possible. Thus, STL specifies for each container, which iterator categories it provides and for each algorithm, which iterator categories it requires. These are both defined as English annotations in the standard [17]. Thus, one may observe that C++ does not have sufficient formalism to express the set of requirements STL imposes on its iterators and containers. Next section shows how we can do better with GIDL.

4.7.1.2 STL's GIDL Specification

In our mapping, we have preserved the main design characteristics of STL. At the GIDL level, the design of iterators and containers is not intrusive: It does not assume any kind of inheritance from a common ancestor. This is achieved by the use of the *export-based qualification*. We also preserve the orthogonal structure of containers and algorithms. As our goal is to make available the STL library in a multi-component environment through a minimal coding effort, our server relies on the STL's functionality in its implementation.

We have abstracted the STL iterators' functionality, making it context independent. We have done this using additional generic types, bounded by a mutual recursive export-based qualification, as shown in Figure 4.19. It follows that `InpIt<T>` exports the method `boolean ==(InpIt<T> i)` while `RaiIt<T>` exports the method `boolean ==(RaiIt<T> i)`.

Figure 4.19 Export-based qualification for iterators

```
1  interface InputIterator< T, It:-Iterators::InputIterator<T,It> >
2          : Iterators::BaseIterator<T,It> {
3      boolean ==(in It it);
4      /*...*/
5  };
6
7  interface InpIt<T> : InputIterator<T, InpIt<T> >{};
8
9  interface STLvector
10   < T,
11     Ite:-Iterators::RandAccessIterator<T, Ite>,
12     II:-Iterators::InputIterator<T, II> > {
13     /* ... */
14 };
```

The code excerpt above shows how things work together. The `STLvector` container does not expect the iterators to be built within any inheritance relation, but only for them to implement the functionality described in the STL specification. We use inheritance among iterators just because this provides a better expressivity and keeps the code short. This is not a requirement, however, as seen from the `STLvector` and `InputIterator` definitions. One can extend our specification for a set of iterators without using any inheritance relation among them.

We believe that our STL specification is reasonably expressive and self-describing. A generic algorithm is mapped to a parameterized function, where the type parameters are qualified to enforce the semantics of the GIDL specification. An excerpt is given in Figure 4.20. This shows how the `find` algorithm uses two type variables in its specification. One, `T`, is unqualified, for the type of the values the iterator is holding. The other, `It`, uses export-based qualification and shares "structural similarity" [5] with its F-bounded qualifier, `InputIterator`.

Figure 4.20 GIDL specification for STL algorithms

```
1  interface Algorithms {
2    <T, It:-Iterators::InputIterator<T,It> >
3    It find(in It first, in It last, in T val);
4
5    <T, II:-Iterators::InputIterator<T, II>,
6        OI:-Iterators::OutputIterator<T, OI> >
7    OI copy(in II first, in II last, in OI result);
8
9    <T, FI:-Iterators::ForwardIterator<T, FI> >
10   void replace(in FI first, in FI last, in T x, in T y);
11   ..................................................
12 };
```

4.7.1.3 Implementation Issues

Our implementation uses the STL as a black box, since the goal is not to rewrite STL but rather to export its functionality in a multi-language environment. The C++ GIDL stubs make heavy use of overloading, as STL exposes these features in the specification of iterators and containers. The operations `++`, `--`, `+=`, `-=`, `[]`, `*` are exported by certain types of iterators, and `==`, `!=`, `>`, `<` are exported by both iterators and containers.

Our GIDL objects for STL are simple wrappers for the STL library constructs. For example, our implementation of `STLvector` keeps an STL `vector` as instance variable and, upon invocation, it calls the appropriate method of the STL `vector` and wraps the returned object to give the corresponding GIDL type. Generic algorithm and function objects are mapped to parameterized functions and interfaces, each containing only one method: the function call operator `()`. Their implementation simply calls the STL function and again wraps the result to a valid GIDL type.

The previous subsection has shown that our GIDL specification has enforced the required STL semantics for iterators. The iterators, together with the functional objects are of central importance in ensuring the STL *orthogonal* design (the first represent abstract data accessing methods, while the latter allows generic algorithms and some container classes to vary their computation in other ways than those exposed by iterators). One consequence of our mapping is that the GIDL stub wrappers corresponding to iterators and functional objects are themselves valid STL types and therefore they can be uniformly manipulated by "native" STL exports such as algorithms and containers, if so desired. An invocation on such a GIDL iterator will redirect the call to the (possibly remote) server side. Thus it is not only possible to have a "black box" automatic library translation strategy, but also to have a distributed implementation for iterators and functional objects.

An interesting and challenging problem is to optimize the usage of the generic library in a multi-language, distributed setting, as many of the implicit assumptions taken in the original design of the library are no longer true. These include the assumptions of a single space and language environment. For example, if in a distributed environment one is traversing an iterator, the performance will greatly suffer, since in order to obtain each value, a foreign, possibly remote call is performed. A thread level speculation approach to reduce the communication and dispatching overheads is presented in Chapter 5, and [43].

4.7.2 Accessing Aldor's BasicMath Library in a Multi-Language Environment

This section investigates the high-level ideas involved in translating the semantics of the Aldor library to GIDL. This includes the two level type system, functional and category subtyping, dependent types and other issues. As Aldor and GIDL are quite different, the design of a translation scheme that enforces the semantics of the Aldor type system in GIDL is a test of our generic model.

Our previous work in the context of automatic library translation, described in Chapter 3 and [44], has studied what is required to use Aldor libraries to extend Maple [38], a dynamically typed, functional, computer algebra language, in an effective and natural way. The resulting framework, called *Alma*, implements a high-level correspondence between Maple and Aldor concepts, and is able to automatically generate Maple stubs corresponding to the functionality of an Aldor library. The user could manipulate Aldor and Maple objects in a completely uniform manner. Work is in progress to enhance the *Alma* framework with support for generating GIDL specifications (corresponding to Aldor libraries) together with automatic *server implementation* (that will use the Aldor library as a black box).

The main challenge in mapping Aldor semantics in GIDL is to achieve proper functional subtyping constraints for the GIDL interfaces representing Aldor functions. We are do this using semantics constructors for subtype, supertype and set membership relations. These are implemented as the trivial parameterized interfaces `SubType<T>`, `SuperType<T>` and `InstanceOf<T>`. Fluet and Pucella [21] employ a similar "phantom types" technique that uses free type variables to encode subtyping information together with a Hindley-Milner based type system [16] to enforce it.

Figure 4.21 Excerpt from Aldor Integer and List

```
1  define Ring       :  Category  ==             with  {...};
2  define CatA(R:Ring) : Category ==             with  {...};
3  define CatB(R:Ring) : Category ==  CatA(R)    with  {...};
4
5  fun( b: CatB(Integer), a: CatA(Integer) ): Boolean == {...};
6
7  define IntegerNumberSystem : Category == Ring with {
8      greater :  (%, %)          -> %;
9      coerce :   (BInt$Machine)  -> %;
10 }
11 Integer : IntegerNumberSystem == add { ... };
12
13 ListCategory(S: Type) : Category == with {
14     nil      : () -> %;
15     isEmpty  : (%) -> boolean;
16     first    : (%) -> S;
17     rest     : (%) -> %;
18     sort     : ((S, S) -> Boolean, %)   -> %;
19     merge    : ((S, S) -> Boolean, %, %) -> %;
20 }
21 List(S: Type) : ListCategory(S) == add { ... }
```

We have introduced the Aldor language to the reader in Section 2.1.1. Suppose we want to expose the Aldor exports shown in Figure 4.21 for use in a multi-language environment. Figures 4.22 and 4.23 show the corresponding GIDL specification.

To simulate Aldor's two level (domain/category) type system, we have introduced the trivial GIDL generic interface `InstanceOf<T>`. If the Aldor domain `DomA` $\in$ `CatA`, then in GIDL we make `DomA` to inherit from `InstanceOf<CatA>`. To correctly handle functional and category subtyping we have introduced the trivial GIDL generic interfaces `SubType<T>` and `SuperType<T>`. To express an Aldor subtype or supertype relation between the categories `CatA` and `CatB`, in GIDL we make `CatA` extend `SubType<CatB>` or `SuperType<CatB>` as appropriate (see *Categories* part in

Figure 4.22 GIDL for Aldor exports of Figure 4.21

```
1
2  /************************ Root Types ************************/
3
4  interface Type                              {};
5
6  interface InstanceOf<T>                     {};
7  interface SubType<T>                        {};
8  interface SuperType<T>                      {};
9  interface FunctionalType                    {};
10
11 interface Category : InstanceOf<Type>   {};
12 interface Domain : InstanceOf<Type>     {
13     <C:Category> boolean has(in C c);
14 };
15
16 /************************ Categories ************************/
17
18 interface Ring: Category, SubType<Ring>,
19   SuperType<Ring>, SuperType<IntegerNumberSystem>  {};
20
21 interface CatA< R:InstanceOf<Ring> > :
22   SubType< CatA<R> >, Category,
23   SuperType< CatB<R> >, SuperType< CatA<R> >       {};
24
25 interface CatB< R:InstanceOf<Ring> > :
26   SubType< CatB<R> >, Category,
27   SubType< CatA<R> >, SuperType< CatB<R> >         {};
28
29 interface IntegerNumberSystem :
30   Category, SubType<Ring>,
31   SubType<IntegerNumberSystem>,
32   SuperType<IntegerNumberSystem>                   {};
33
34 interface ListCategory< S:InstanceOf<Type> > :
35   Category, SubType< ListCategory<S> >,
36   SuperType< ListCategory<S> >                     {};
```

Figure 4.23 GIDL for Aldor exports of Figure 4.21 – continuation

```
/***************** Domains: Integer and List<S> *****************/
interface GlobalExports {
    Boolean fun(in CatA a, in CatB b);
};

interface Integer : Domain, InstanceOf< Ring >, SubType<Integer>,
      SuperType<Integer>, InstanceOf< IntegerNumberSystem > {
    Boolean greater (in Integer i1, in Integer i2);
    Integer coerce (in long l);
};

interface List< S:InstanceOf<Type> >: Domain, SubType<List<S>>,
      SuperType<List<S>>, InstanceOf< ListCategory<S> >    {
    List<S>   nil      ();
    Boolean   isEmpty  (in List<S> l);
    S         first    (in List<S> l);
    List<S>   rest     (in List<S> l);

  <S1: SuperType<S>, S2: SuperType<S>, F:-Sign_SS_Bool<S, S1, S2> >
    List<S>   sort1    (in F f, in List<S> l);

  <S1: SuperType<S>, S2: SuperType<S>, F:-Sign_SS_Bool<S, S1, S2> >
    List<S>   merge    (in F f, in List<S> l1, in List<S> l2);
};

/************************ Functional Types ************************/
interface fun                  :             FunctionalType {
    Boolean GIDLapply( in CatB<Integer> b, in CatA<Integer> a );
};
interface greater              :             FunctionalType {
    Boolean GIDLapply( in Integer i1, in Integer i2 );
};

interface Sign_SS_Bool < S : InstanceOf<Type>,
                         S1 : SuperType<S>, S2 : SuperType<S>
                      >          :          FunctionalType {
    Boolean GIDLapply( in S1 s1, in S2 s2 );
};
```

Figure 4.22). This is needed because if we make `CatA` directly extend `CatB`, we cannot express the supertype relation. This does not introduce any significant run-time overhead. Categories are used in Aldor only to specify properties of domains, such as which operations they export and to qualify type parameters. Domains provide the implementation. Whereas type categories will normally form an interesting subtype lattice, Aldor libraries have only trivial subtype relations among domains. We therefore provide no sub/super-typing relation between them at the GIDL level.

In Aldor, all domains and categories satisfy the `Type` type. The `Domain` and `Category` interfaces are the base classes for the GIDL interfaces corresponding to the Aldor domains and categories. They both thus inherit from `InstanceOf<Type>`. An Aldor declaration such as `R:Ring` is given at the GIDL level as a type qualification `R: InstanceOf<Ring>`. For example, `Integer` should be a valid instantiation for the generic type `R`.

Figure 4.22 demonstrates how function parameters and functional subtyping mapped into GIDL. The definition of `ListCategory` in Figure 4.21 expresses that both `merge` and `sort` functions receive as first parameter a function whose signature is `(S,S)->Boolean`, where `S` is qualified by `Type` in Aldor and `InstanceOf<Type>` in GIDL. Corresponding to the signature of the functional object, the GIDL compiler generates the `Sign_SS_Bool` interface containing only one function, generically named `GIDLapply`. The signature of the function `GIDLapply` illustrates functional subtyping, with contravariant subtyping for parameter types. Note that the types of the parameters `S1` and `S2` are supertypes of the original parameter type `S` of the original declarations for `sort` and `merge`.

The `sort` and `merge` methods of the `List` interface are parameterized with the qualifications

```
< S1: SuperType<S>, S2: SuperType<S>,
  F:- Sign_SS_Bool<S, S1, S2> >
```

and `F` is used instead of the functional type. The export based qualification of `F` ensures that all the possible candidates will be taken into account (see MSGA in Section 4.4.2). Ultimately, the GIDL compiler will find all the Aldor exports that may have a functional subtype of `(S,S)->Boolean` and will generate interfaces, each containing one method named `GIDLapply`. For example the `greater` and `fun` GIDL interfaces correspond to the Aldor functions with the same names. The `fun` function is a valid first parameter for the `sort` function in `List<CatB<Integer>>` as `CatA<Integer>` extends `SuperType <CatB<Integer>>`. Similarly, `greater` is a valid first parameter for the `sort` and `merge` functions in `List<Integer>` interface as the two signatures are identical.

Finally, we note that in Aldor functional subtyping is most often trivial. Assume that the qualification for `S` is a defined category instead of `Type`. It follows that the extra parameters `S1` and `S2` are not needed because no non-trivial super-type exists. Similar reasoning explains why the mapping does not introduce an extra qualified type:

```
S3: SubType<Boolean>
```

for the return type of the functional object.

To conclude, Figures 4.22 and 4.23 present a legal GIDL specification that enforces the semantics of the original Aldor code in Figure 4.21, and the translation process described here has been generalized and automated.

4.8 Chapter Conclusions

Previous interoperability experiments, described in Chapter 3 and Section 2.1, have shown that we can match the semantics of different flavors of parametric polymorphism: Aldor's dependent types *vs.* C++'s static templates *vs.* Maple's module-producing functions. This chapter has proposed GIDL: a more general and systematic solution that encompasses more languages in a simpler way.

The first step was to define a model for generics that could support the interface semantics for generics in a range of different programming languages. We have seen from our implementation of GIDL that qualification 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 qualifications can be supported effectively. Their implementation introduces almost no run-time overhead.

We have shown that this parameterization in GIDL can be supported by translation to IDL, with the generation of appropriate wrappers. This allows such code to be used with existing CORBA implementations and to take advantage of the usual support for distributed applications. Applications which are not distributed, may make use of GIDL simply to support multi-language use of generic modules. This use involves minimal overhead. We have also shown that, with little modification, GIDL can be used to extend other interconnection architectures such as DCOM and JNI.

Finally, we have demonstrated that it is feasible to export parametric polymorphic libraries to a multi-language environment via GIDL: We have implemented a component that accesses a significant part of the C++ STL functionality. From this, we have seen how imposing qualification restrictions can improve the precision and safety of the STL library interface. We have also presented the main ideas involved in mapping the Aldor language features to the GIDL level. This allows Aldor libraries to be used across language boundaries via GIDL.

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. We view our work as a contribution to this area.

Chapter 5

Distributed Models of Thread Level Speculation

5.1 Chapter Introduction

This chapter applies thread level speculation to an area in which it has not been previously attempted, namely distributed systems, and finds that, besides the obvious parallelization benefit, this may effectively reduce the communication and dispatch overhead inherent to such architectures. The work presented here is based on the PDPTA paper "Distributed Models of Thread-Level Speculation" [43], co-authored with Jason Selby, Mark Giesbrecht and Stephen Watt.

Distributed Software Component Architectures (DSCA) provide a mechanism for software modules to be developed independently, using different programming languages. These components can be combined in various configurations, to construct distributed applications. Chapter 4 proposed a generic component architecture "ex-

tension" that provides support for parameterized (generic) components, and can be easily adapted to work on top of various SCAs (CORBA [50], DCOM [36]).

There is increasing interest in the subject of automatically exporting generic libraries across their initial language boundaries. Our experiments, already described in Section 4.7.1 and Chapter 3, have exposed part of C++'s STL and Aldor's [68] BasicMath libraries for use across the Generic IDL (GIDL) [12, 45] and Alma [42, 44] frameworks respectively. This work has also revealed several performance issues. First, the overhead associated with inter-component communication stalls can be quite significant. In the context of a distributed application, the network and dispatching overhead may become dominant. This is especially true for object-oriented languages since they expose smaller average method length. Second, separate compilation of components hinders traditional compiler optimizations such as inlining.

This chapter explores the novel application of speculative techniques to a distributed environment that address the afore-mentioned issues. We propose two models of Thread-Level Speculation (TLS) that can discover parallelism that is not exploitable using traditional parallelizing compiler techniques. Their application can yield substantial performance benefits, even in the case when the underlying hardware is not a multiprocessor.

The first model attempts to overlap the client-server communication overhead with useful computation performed on the server side in the form of speculation. This allows multiple remote invocations to be replaced with fewer calls that the server expands in multiple speculative iterations of the same code. We obtained speed-ups of about 2× when the client and server share the same machine, and about 3.5× in the distributed case.

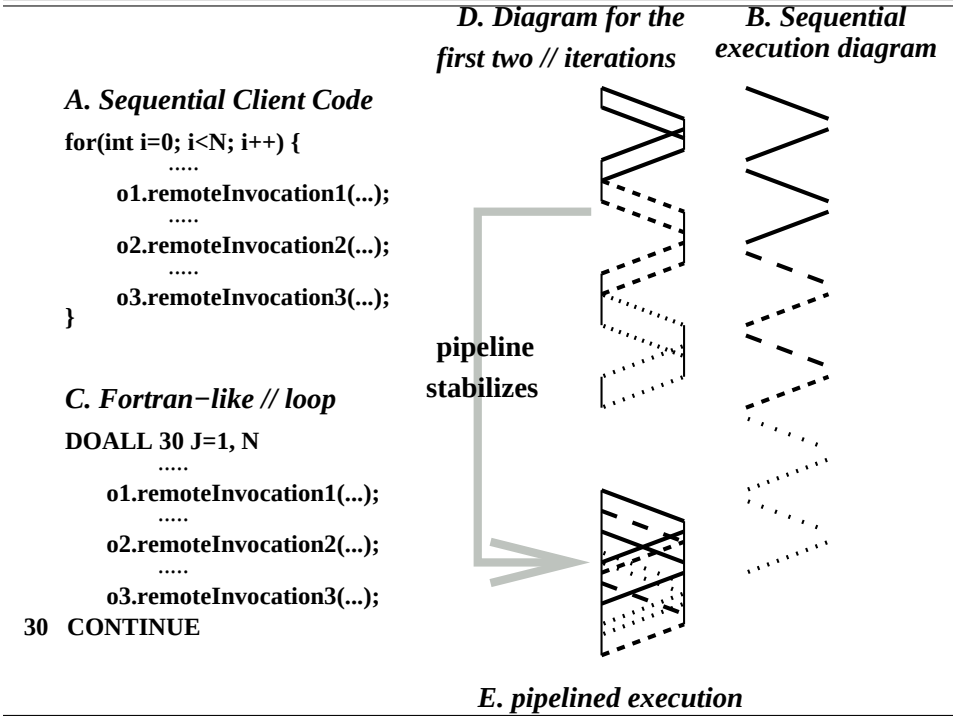
The second model simulates "procedure inlining". The server (master) runs a predictor program that approximates the code that was supposed to be executed by the client. The client validates the correctness of the predicted version of the program using results sent back by the server. This model obtains speed-ups of about 11× when the client and server share the same machine, and about 21× for the distributed case.

Section 2.3 and Section 2.4 have already provided an overview of the mainstream distributed component technology in use today (CORBA, DCOM, .NET), and of the current TLS approaches respectively. The remainder of this chapter is organized as follows. We describe the application of TLS to a distributed heterogeneous environment in Section 5.2. Afterward, in Section 5.3 we report and analyze the performance benefits of exploiting the parallelism enabled by TLS in order to speed-up client-server applications. Finally, we conclude with the contributions of the work presented in this chapter in Section 5.4.

5.2 Distributed Applications of Thread-Level Speculation

This section introduces two TLS models, inspired by [59] and [73], which can be applied in a potentially multi-language, distributed environment. Performance improvements are derived from two aspects. First, the communication overhead is reduced by eliminating stalls between the client and the server, and secondly, by taking advantage of the server/client support for parallel execution. In most situations the second model yields better speed-ups compared to the first. However, in environments where

Figure 5.1 An example of a simple object-oriented client program.



security is of concern, the code migration aspect of the second approach might forbid its use.

Throughout this chapter we assume that the server's throughput is reasonable low (that is, the server has some idle time and is not over-run with clients requesting its services). Section 5.2.1 presents an overview of our approach, while Sections 5.2.2 and 5.2.3 introduce the two speculation models respectively.

5.2.1 Overview

Figure 5.1.A presents an example of a general, object-oriented, client program, and Figure 5.1.B displays its normal (sequential) execution. However, if the loop can be executed concurrently, as evident in Figure 5.1.C, then the speed-up can be quite substantial. Figure 5.1.D shows the diagram's temporal execution of the first two

concurrent iterations. After some number of iterations, the *pipeline* stabilizes. Examining Figure 5.1.E, we see that the costs of the communication is ameliorated. The communication costs could be further decreased by "inlining" the client code into the server. Additionally, server-side parallelism can be effectively exploited. This becomes more important as the granularity of a method increases.

Figure 5.1 displays an ideal Fortran `DOALL` parallelization of the program. However, this is not possible since the code is split and separately compiled between the client and the server. To achieve this, we employ our distributed TLS models that are discussed in Sections 5.2.2 and 5.2.3.

5.2.2 Distributed Speculation Model

This section provides an overview of our TLS framework and describes its application to a distributed environment. Our model differs from that of a typical TLS scheme by the fact that the speculative variables may reside on a remote machine and therefore are not directly accessible by the client. However, the remote object whose methods uses these variables can act as a proxy for them. If the method's parameters are also remote objects, than recursively, their server is required to provide parallelization support for the operations that are invoked upon them. If support for speculative parallelization is unavailable, and the code cannot be proven to be free of data-dependencies then speculation is not applied.

Figure 5.3 presents part of a two-client program that uses the services provided by a server that implements the functionality of the GIDL specification presented in Figure 5.2 (ignore for the moment the lines marked with `*` and the `TLSPackage` module). Assuming that the server's code is available for analysis, note that the

Figure 5.2 GIDL specification. Lines marked with * denote TLS support

```
1  module TLSPackage {
2    exception TLS_Dependence_Violation { long thread_num; };
3    interface Speculative_Variable {
4      void reset(in long tid, in long max_tid);
5      void commitValueInFront(in long tid);
6      void start_speculation();
7    };
8    interface Splitable_Variable<T:Splitable_Variable<T> > :
9        Speculative_Variable {
10     typedef sequence<T> Seq_T;
11     Seq_T splitSpeculativeVariable(in long nr);
12     void recombineIterators(in Seq_T s);
13   };
14 };
15
16 interface GetValueObject {
17     long getValue();   void setValue(in long val);
18 };
19
20 module IteratorPackage {
21   interface Iterator<T> :
22         TLSPackage::Splitable_Variable<Iterator<T>>{ // *
23     long  isEmpty();    void  step();
24     T     value();      void  resetIterator();
25   };
26 };
27
28 module ContainerPackage {  //...
29   interface Vector<T:GetValueObject, C:Comparator<T> > :
30     Container<T,C>, TLSPackage::Speculative_Variable { // *
31     T elementAt(in long i);
32     void setElementAt(in T obj, in long i);
33     T Spec_elementAt(in long i, in long thread_num);  // *
34     void Spec_setElementAt(                           // *
35         in T obj, in long i, in long thread_num
36     )raises (TLSPackage::TLS_Dependence_Violation); //....
37   }; //....
38 }; //....
```

Figure 5.3 Two client code regions which are rich in speculative parallelism.

```
1  // A)
2
3  for(int i=0; i<dim[0]; i++) {
4
5      GetValueObject gvo = vect.elementAt( new Long_GIDL(i) );
6      int elem = gvo.getValue().getValue(); elem *= ...;
7
8      if(elem>(-1)) gvo.setValue(new Long_GIDL(elem));
9      else {
10         GetValueObject gvo1;
11
12         if(i>0) {
13             gvo1 = vect.elementAt( new Long_GIDL(i-1) ); //***
14             elem = (long)gvo1.getValue().getValue(); elem*= ...;
15         } else elem = ...;
16
17         gvo1 =
18            factoryImpl.createComparableObject(new Long_GIDL(elem));
19         vect.setElementAt(gvo1, new Long_GIDL(i));
20     }
21 }
22
23 // B)
24
25 for(; index_it.isEmpty().getValue()!=0; index_it.step()) {
26
27     Long_GIDL ind = index_it.value();
28     GetValueObject gvo = vect.elementAt(ind);           //***
29     int elem = gvo.getValue().getValue(); elem *= ...;
30
31     if(isValidElement(elem)) {
32         GetValueObject gvo =
33            factoryImpl.createComparableObject(new Long_GIDL(elem));
34         vect.setElementAt(gvo, ind);                    // ***
35     }
36 }
```

client code cannot be conservatively parallelized due to the loop-carried true data-dependence of distance 1 in client A, and due to the indirect access of the vector's `vect` elements in client B (see the lines marked `***`). In both cases, profiling information combined with code analysis performed on the client may (non-conservatively) suggest that a region of rich-parallelism has been discovered. Suppose the `if` branch is *cold*, considering the *hot* path the code "resembles" a data-dependence free loop (modulo the data dependences introduced by possible object aliasing). Given these hindrances to parallelization our speculative framework can be employed.

The client announces to the server that speculation is about to commence, and provides the required information regarding the speculative region. The TLS module used by the GIDL stub will invoke the target-language compiler (Java in our example) to compile the respective methods with support for speculation, thus generating some new (speculative-related) methods on the server side. While it is clear how this transformation would be implemented we are currently performing it by hand. Furthermore, it will modify the GIDL specification to also include speculation (lines marked with `*` together with the `TLSPackage` module in Figure 5.2), and re-compile it to update the client and server stubs.

Each interface that is found to contain at least one speculative method is required to inherit from the `TLSPackage::Speculative_Variable` interface (see Figure 5.2). Essentially, such an interface functions as a proxy for the speculative variables identified in its speculative-methods (as they do not have distributed support). Information received from the client will aid the server-side compiler to prune the number of variables that are considered speculative. However, if this is the only modification, the client-code labelled `B` in Figure 5.3 will generate many rollbacks due to the iterator `step` operation. To solve this, `Iterator` extends the `Splittable_Variable` interface,

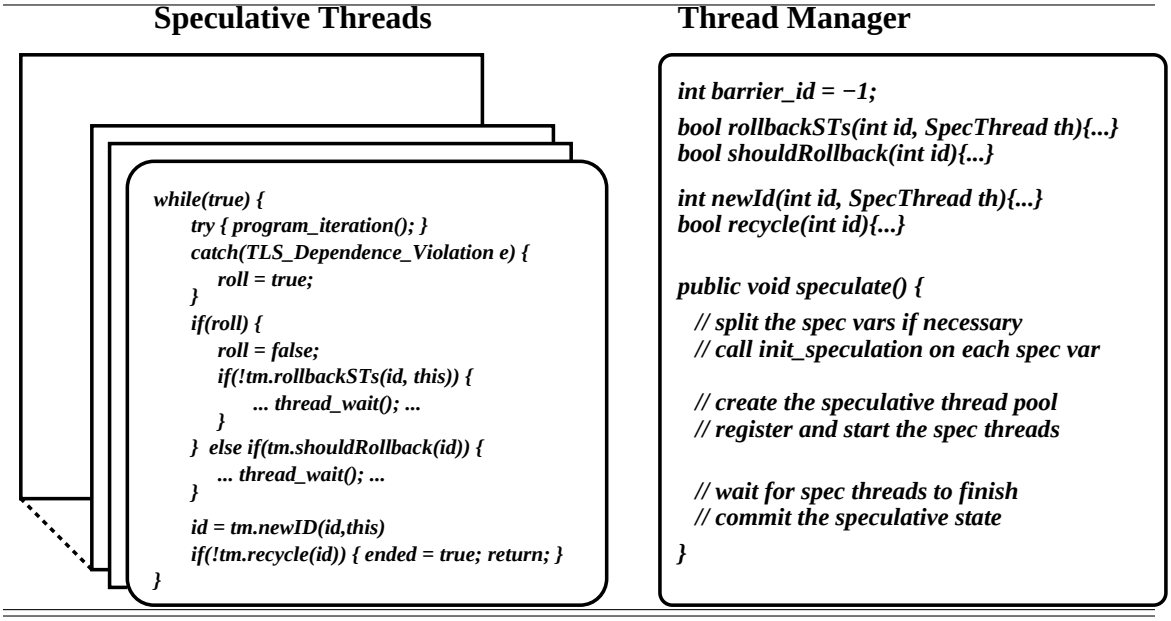
Figure 5.4 Part of the server-side speculative code for ContainerPackage::Vector

```
1  T[] arr; TLS.Arrays.Spec_Arr_RefU1D<T> spec_arr;
2  ArrayList<GIDL.TLSPackage.Speculative_Variable> Spec_Vars;
3
4  final public void start_speculation() {
5      spec_arr=new TLS.Arrays.Spec_Arr_RefU1D<T>(arr,1,1,ob_T);
6      Spec_Vars.add(spec_arr);
7  }
8
9  final public void Spec_setElementAt(T ob, Long_GIDL a1) {
10     arr[a1.getValue()] = ob;
11 }
12
13 final public void Spec_setElementAt
14        (T ob, Long_GIDL a1, Long_GIDL th)
15                  throws _TLSPackage.TLS_Dependence_Violation {
16     int th_num = th.getValue();
17     try {
18         spec_arr.Speculative_Store(a1.getValue(), th_num, ob);
19     } catch(TLS.Dependence_Violation exc) {
20         throw new _TLSPackage.TLS_Dependence_Violation(th_num);
21     }
22 }
```

allowing each speculative thread to work with disjoint (separate) iterators (refer to Section 2.4.2 for speculative support for container classes).

Figure 5.4 presents the `setElementAt` method of the `Vector` interface, and its speculative version `Spec_setElementAt`. Notice that the generated speculative code differs very little from the original. Specifically, it receives an extra parameter, the id of the thread executing the method (`th`). Second, the speculative operation is guarded by a `try-catch` block. If a violation is detected than the exception is forwarded as a GIDL exception onto the client. Finally, the container that may be the source of a data-dependence violation (`arr:T[]`) is replaced with a speculative

Figure 5.5 The interaction between the speculative threads and the thread manager



version (in this case the `spec_arr:TLS.Arrays.Spec_Arr_RefU1D<T>`). Figure 5.4 displays the implementation of the `start_speculation()` method exported by the `GIDL.TLSPackage.Speculative_Variable` interface. It initializes the variables on which data-dependence violations might occur, and stores them in a container. The `reset` and `commitValueInFront` methods (omitted from Figure 5.4 due to space constraints) traverse the list of speculative variables encapsulated by this class (`Vector`) and re-initializes them, or updates the original location that they shadow, respectively. These methods are invoked when handling a rollback or when speculation has succeeded and the speculative state should be merged with the true non-speculative state, respectively.

As depicted in Figure 5.5, the client starts speculative execution by creating a thread-manager, and calling the `speculate` method on it. The thread manager calls the `start_speculation` method on all local speculative variables, and on all the remote objects that act as proxies for the speculative variables identified on the server.

Furthermore, it creates a pool of speculative threads (registered to itself) and starts them. A speculative thread executes iterations corresponding to the sequential code, except that it now references local speculative variables and invokes the speculative handler methods. At the end of an iteration the speculative thread checks to see if any violations were detected by the other threads. If so, the thread transitions into the waiting state. Otherwise it is assigned a new `id` (sequential execution iteration number), and checks to see whether the terminating condition was met. If a thread catches a data-dependence violation exception (thrown by local code or by the server), it invokes the `rollbackSTs` method on its thread manager, which will set the manager's `barrier_id` flag. In the end, only the lowest id thread that has detected a rollback will be alive. At this time, for each speculative variable the value generated by the thread with the highest `id` less than or equal to the `id` of the running thread is committed. Finally, all the speculative variables are committed, and cleaned up. Adaptability is built into the system by monitoring the ratio of rollbacks to commits. If a predefined threshold is passed then speculation is abandoned for sequential execution, otherwise the speculative threads are awakened and speculation continues.

5.2.3 Distributed Speculative-Inlining Model

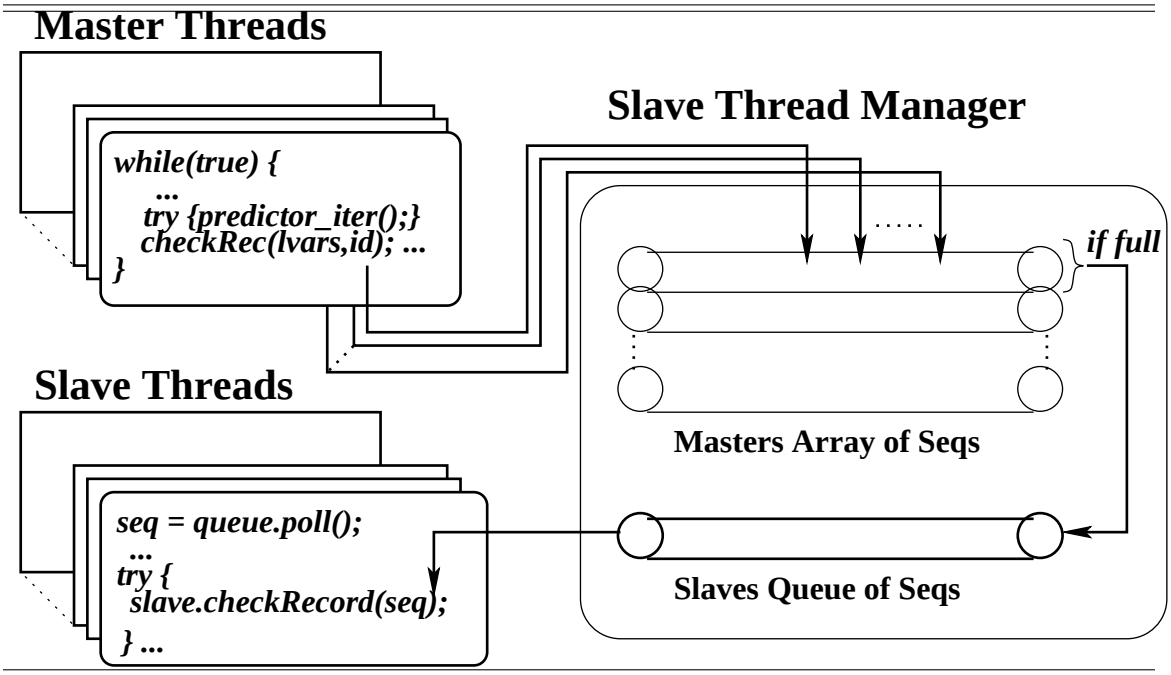
The second speculative model presented here, inspired by [73], achieves a speed-up in a similar manner as procedure inlining. More precisely, the client provides the server (or vice versa) with a *predictor* program that approximates the code executed by the client. There are no constraints associated with the distilled program. However, in order to produce a good speed-up, it needs to achieve a high prediction accuracy. The

server (*master*) runs the predictor program and sends back to the client, records of the live variables computed along the anticipated path through the client's code. It is the client's responsibility to validate the correctness of the master's execution.

Our model differs from [73] in several ways. First, [73] expects the distilled program to be much faster (a straight line code segment of the dominant path) than the slave's verification code. In our case, we prefer the *approximate* program to be as close as possible to the original (and hence less likely to contain a violation), because of the high cost associated with a rollback. Second, our implementation is adapted to a distributed environment, and therefore, is geared toward other goals, such as network, and dispatching overhead elimination. The parallelization of the predictor program becomes more important for us as the iteration granularity increases.

There are two situations when program distillation is most beneficial inside of our framework. The first is when a method returns a predictable value. Consider a local object which is used as a branch condition, for an example see Figure 5.3.B: `if(client_obj.IsValidElement(...))`. In this case the *hot* branch will be added to the predictor but without the test (the test will be a remote invocation from the server point of view, and thus expensive). The second case, is when the deletion of a *cold* branch causes the number of speculative variables to drastically decrease, or the predictor code becomes conservatively parallelizable. In such a situation the server may even employ a standard parallelization model to achieve the greatest speed-up. In Figure 5.3.A, if the *true* branch from `if(elem>-1) ...` is found to be *hot* then a predictive program can be constructed by keeping the target, and removing the cold path. Further analysis by the server-side compiler of the predictor may conservatively discover that the vector's element holder (`arr` in Figure 5.4) will not generate any data dependence violations.

Figure 5.6 "Inlining" - like speculative model. This figure presents the interaction between the master/slave threads and the slave thread manager



The server side of the inlining speculative model is mainly composed from two communicating instances of our TLS framework, as shown in Figure 5.6.

Master threads, registered to a master thread-manager, execute out of order iterations of the distilled program. At the end of every iteration, the live variables of the master threads are packed into a record residing in a predefined location in an array of sequences of records indexed by the thread's `id` (viewed as a bi-dimensional array – the *Masters Array of Seqs* in Figure 5.6). Master threads are not permitted to over-write non-null records since this means that the record has not yet been committed because at least one thread is lagging behind. When a sequence is filled up, it is inserted into the *slave* queue (*Slaves Queue of Seqs* in Figure 5.6) and a new, empty sequence is placed in the table. The terminating condition of the master threads is dictated by the client's code.

The slave threads poll a sequence from the slave-queue (if not empty, otherwise yield and try again). They request the client (that now acts like a server) to verify the current sequence containing several live-variable records. A slave-thread's exit condition is reached when all of the master-threads are dead and no data in the slave-queue requires verification. No explicit synchronization is required between the master and slave threads except for guarded access to the slave-queue.

The client is responsible for verification. If any of the instructions that were not part of the *predictor* program (branch conditions excluded) are reached, or a *cold* branch excluded from the predictor is taken, then a violation has occurred. The client throws a dependence-violation exception that will be caught by the corresponding slave thread on the server-side. The slave thread manager will handle the rollback as described in the previous section, additionally it will set the `barrier_id` flag of the master thread manager to the `id` of the thread that detected the violation. Thus all of the master-threads are going to be in a waiting-state (all have an `id` greater than `barrier_id`, otherwise the corresponding sequence wouldn't have reached the client), and finally, only one slave-thread (the thread with the lowest id that detected a rollback) is running. Only then are the speculative variables committed and reinitialized. Control is then handed to the client which sequentially performs the iterations corresponding to the records in the received sequence.

Figure 5.7 presents the GIDL specification, corresponding to the client program displayed in Figure 5.3.A, that is needed by our "inlining speculative model". When a client discovers a suitable code region for speculation, it locally creates and runs a slave checking-server (type `Slave1<...>`). The method:

```
Master1<E, C> createMaster1(Slave1<E> slave)
```

creates a remote-object that upon invoking the `runMaster` method will create the

Figure 5.7 GIDL specification support for the inlining speculative model

```
1  module MasterSlavePack {
2
3    interface Master1 <
4                        T: GetValueObject,
5                        C: ContainerPackage::Comparator<T>
6                       > {
7        void runMaster(in long i, in long j,
8            in long s, in long l,
9            in long sps, in long ms,
10           in ContainerPackage::Vector<T, C> vect
11       );
12   };
13
14   interface Slave1<T: GetValueObject> {
15       struct LiveVariables {
16           T elementAt_result; long thread_nr;
17           long getValue_result;
18       };
19       typedef sequence<LiveVariables> seq_LV;
20
21       void checkRecord(in seq_LV lv)
22           raises(TLSPackage::TLS_Dependence_Violation);
23
24       void performRollbackIteration(in seq_LV lv);
25   };
26 };
27 //...
```

server-side two-level TLS architecture described previously. The `checkRecord` method in the `Slave1` interface performs the speculation validation. If a dependence violation exception is thrown the client is requested to sequentially execute several iterations (`performRollbackIteration(...)`).

As noted in the beginning of this section the inlining model almost always yields better speed-ups compared to the first approach. This is due to the fact that the

number of remote calls performed by the two models is $1/(MasterCheckingSeqSize * NrOfRemoteCallsPerIt)$ in favor of the inlining speculative model. However, client code may reference many objects distributed across many servers, among which some may not support code exchange via a common intermediate representation (IR). Moreover, security issues may disallow the sharing of certain pieces of code or data. In this case, a combination of the two models is the preferred solution (if the code possesses high-level parallelism). The *master* is selected by identifying the remote object that is invoked most frequently. Predictive programs corresponding to the functionality of the servers that support a common communication IR and allow code migration will be also inlined into the master. If the code exposes parallelism, the execution time may be further decreased by concurrently executing speculative iterations of the master thread. We can see that one application may create a hierarchy of inlined speculations and overlapping speculative iterations (first model).

5.3 Results

Automatic library translation across language boundaries is an area yet to be explored. Unfortunately, it is lacking in formal benchmarks that can accurately measure the performance effects associated with porting a non distributed application into a distributed environment. We implemented a GIDL-server which exhibits functionality similar to that found in the STL of C++ (for example, containers, iterators, etc). Our tests are based on variations of the two examples used throughout this paper. The "remote" method granularity was varied from `10` to `10000` instructions (notice that each iteration performs between 3 and 5 remote calls). Our tests were carried out on two configurations. One configuration ran on a single machine which acted

Table 5.1 Distributed TLS 1st Architecture (overlapping communication)
Nr = client thread pool size,
G = "remote" method granularity (instructions)
nMc speed-up compared to sequential.
n = no. machines, c = client version
nMcR as above, but with 1% rollback rate.

Nr	G	1M1	1M1R	1M2	1M2R	2M1	2M1R	2M2	2M2R
4	10	1.35	1.30	1.30	1.23	2.23	2.05	2.05	1.98
8	10	1.55	1.51	1.56	1.52	3.01	2.72	3.24	2.71
16	10	1.65	1.53	1.62	1.53	3.36	2.76	3.36	2.68
32	10	1.91	1.47	1.69	1.44	3.22	2.37	3.46	2.27
4	10^3	1.31	1.28	1.30	1.28	2.09	2.03	2.13	2.03
8	10^3	1.51	1.45	1.53	1.48	3.12	2.72	3.16	3.07
16	10^3	1.62	1.46	1.62	1.46	3.29	2.94	3.47	2.66
32	10^3	1.73	1.48	1.70	1.35	3.53	2.31	3.53	2.17
4	10^4	1.25	1.23	1.32	1.26	2.25	2.03	2.04	1.86
8	10^4	1.36	1.27	1.50	1.38	2.71	2.35	2.78	2.39
16	10^4	1.41	1.24	1.55	1.32	2.83	2.35	3.17	2.41
32	10^4	1.44	1.25	1.63	1.24	2.73	2.01	3.41	2.05

as both client, and server (2.4GHz P4/512 Mb). Another configuration employed two machines on the same local network (both 800MHz P3/256Mb RAM). All the machines we have used are running Linux.

We applied our TLS framework to distributed programming in the anticipation that speed-ups could be obtained by overlapping network stalls with speculative computation, thereby minimizing idle times. Table 5.1 shows the speed-ups obtained by employing our first distributed TLS model compared to sequential program execution. In a rollback-free ("ideal") execution, employing a higher number of client threads generates a better speed-up (`32` client threads achieve a `1.91, 1.69, 3.22, 3.46` times speed-up). Our framework is rollback-tolerant in the sense that it gracefully accommodates a 1% rollback probability. In examination of the cost of a rollback, we notice that the performance difference with respect to the ideal case decreases

Table 5.2 Distributed TLS 2nd Architecture ("inlining"-like speculation)
G = "remote" method granularity (instructions)
SS = slave sequence size,
nMc speed-up compared to sequential.
n = no. machines, c = client version
nMcR as above, but with 1% rollback rate.

G	SS	1M1	1M1R	1M2	1M2R	2M1	2M1R	2M2	2M2R
10	1	3.02	2.31	4.69	3.27	5.86	4.70	8.96	6.58
10^3	1	2.88	2.22	4.20	3.06	4.96	4.67	10.22	9.21
10^4	1	1.96	1.32	2.86	1.88	3.76	2.26	5.19	2.99
10	10	9.59	3.20	11.54	3.65	15.57	4.75	21.10	6.18
10^3	10	7.35	1.77	9.33	2.54	14.05	2.52	14.83	2.86
10^4	10	2.97	0.71	4.13	0.89	3.83	1.10	5.62	1.57

with the size of the thread pool. This is due to the greater number of inter-thread dependencies resulting in redundant work and increased synchronization overhead. The observed number of threads that provided the best speed-up was either 8 or 16.

Our second model clearly yields substantial performance benefits compared to the the first model as demonstrated in Table 5.2. There are two main reasons for this. First, we have eliminated CORBA's inherent remote-call dispatch costs by "inlining" the client code into the server. All remote calls in the initial code are now handled locally. Second, the network overhead is reduced by batched communication of the live variables. The server is configured to use `15` concurrent slave threads in order to "pipeline" the remote-client checking phase.

In an ideal (rollback-free) execution scenario, the application of this model obtains impressive speed-ups. On a single machine, execution time was `9.6` and `11.5` times faster, and `15.6` and `21.1` times faster over a distributed network with a method granularity, and slave sequence size of `10` (slave sequence size represents the number of records sent in a batch for the client to check for correctness). However, for a 1% rollback probability, the corresponding speed-up decreases dramatically (`3.20` –

6.18). This is because, in our implementation, the rollbacks are handled by asking the client to sequentially execute the iterations associated with the sequence of records that have generated the violation (10 in our case). We are currently working on enhancing our architecture to better handle the rollback situation by sequentially executing only the "guilty" iteration. However, the rollback handling will remain expensive (see results in Table 5.2 for sequence-size 1) and influence our predicted program to be more "correct" than "distilled".

Table 5.1 and Table 5.2 show that for both our models, the speed-up decreases when the method-granularity increases. However, in this case, taking advantage of the machine's (potential) parallelism becomes very important.

To summerize this section, the performance gain for our first model (with respect to the sequential client program execution time) depends on the size of the thread pool, on the remote method granularity, and on the rollback ratio. The best speed-ups, for a rollback-free execution, are obtained with 32 client threads and range from 144% to 191% when the client and server share the same machine, and from 353% to 341% for the distributed case, when the method granularity varies from 10000, to 10 respectively. For a 1% rollback rate, the best speed-ups are obtained using a number of threads between 8 and 16. They range from 127% to 153% for the single machine case and from 235% to 276% when the client and server are across a local network, for a method granularity of 10000 and 10 respectively.

The second model mimics "procedure inlining" and is very effective in eliminating the distributed system overhead. For a rollback-free execution we obtained speed-ups between 297% and 1154% for a single machine space, and between 383% and 2110% for the distributed case, for a method granularity of 10000 and 10 respectively. We also

notice that a 1% rollback rate will substantially decrease these speed-ups, therefore we prefer a more "correct" rather than a more "distilled" predictor.

5.4 Chapter Conclusion

This chapter has examined the potential for thread level speculation in a new area: the environment of distributed software components. We have found that substantial speed-ups may be achieved from this level of parallelism.

We propose two TLS models employed in a distributed setting that substantially reduce the network and call dispatch overhead. Additional speed-up is achieved when the underlying hardware is a multiprocessor. This becomes more noticeable as the remote method granularity increases.

The first model performs concurrent speculative iterations on the client, overlapping with communications. The second model mimics procedure inlining to eliminate distributed system overhead.

The performance gain depends on many factors. For the first model speed-up ranges from 1.4× to 1.9× on a single machine, and about 3.5× when distributed. For the second model speed-up ranges roughly between 3× and 11.5× on one machine, and between 3.8× and 22.1× when distributed. Allowing a 1% rollback rate gives a somewhat smaller speed up for the first model, and substantially decreases speed-up for the second model.

The creation of speculatively aware container classes proved to be a highly beneficial idea and warants futher investigation to determine other commonly used libraries where thread-level speculation can be exploited.

Chapter 6

Conclusions

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

The first step has been to invesitgate an interoperability solution between two languages with very different type models: the weakly typed Maple, against the strongly typed, higer order type system Aldor. The resulting framework, named *Alma*, is higher-level than the "usual" foreign function interface approaches, and it fosters a deeper connectivity between the two environments. As a consequence, the facilities of the Aldor language are natuarally integrated in the Maple environment. *Alma* also represents a non-traditional approach to structuring computer algebra software: using an efficient, compiled language, designed for writting complex mathematical libraries together with a top system based on user interface priorities and ease of scripting.

The experiment described in Sections 2.1 has motivated us to explore a systematic solution to languages interoperability in the presence of parametric polymorphism. We have investigated how to resolve different binding times and parametric polymor-

phism semantics in a range of representative programming languages, and have identified a common ground that can be suitably mapped to different language bindings. We have presented GIDL: a generic component architecture extension that provides support for parameterized components, and can be easily adapted to work on top of various software component architectures in use today (CORBA, JNI, DCOM).

We have introduced the semantics of the GIDL parametric polymorphism model, and the language bindings for C++, Java, and Aldor. We have shown that qualification of type parameters can improve the precision and safety of the specification interface, and it can be effectively supported and enforced even when the target language does not allow it. We have described our implementation of GIDL, consisting of a GIDL to IDL compiler and tools for generating linkage code under the language bindings, and have shown that our architecture preserves the backward compatibility with non-generic applications.

Furthemore, we have described how GIDL can be used to export C++'s STL and Aldor's BasicMath libraries to a multi-language environment, and have discussed our mappings in the context of automatic library interface generation. These translations tested our generic model and the GIDL architecture and they have shown that complex language concepts such as *orthogonality* and *functional subtyping* can be expressed at GIDL level.

Our library translation experiments have revealed performance issues related to the inter-component communication stalls. To address these, we have proposed and evaluated two speculative models that attempt to reduce some of the method call overhead associated with the distributed objects. Thread-level speculation exploits parallelism in code which is not provable free of data dependencies. Our application of

thread-level speculation attempts to overlap the client-server communication overhead with useful computation performed on the server side in the form of speculation. Our evaluation of applying thread-level speculation to client-server applications resulted in substantial performance increases, on the order of 3 times for our initial model, and 21 times for the second.

Parametric polymorphism is an important programming language idea that has achieved widespread acceptance over the past decade. Making it available to multi-language components is an important problem, together with developing applicable compiler optimization techniques. Our goal has been to make a contribution in this area.

References

[1] Laurent Bernardin, Bruce Char, and Erich Kaltofen. Symbolic Computation in Java: An Appraisement. In *Proc. ISSAC 1999*, pages 237–244. ACM, 1999.

[2] Anasua Bhowmik and Manoj Franklin. A General Compiler Framework for Speculative Multithreading. In *SPAA'02 Proceedings*. ACM, 2002.

[3] M. Bronstein. SUM-IT: A Strongly-Typed Embeddable Computer Algebra Library. In *Proceedings of DISCO'96, Karlsruhe*. Springer LNCS 1128, 1996.

[4] M. G. Burke, J. D. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M. J. Serrano, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeno Dynamic Optimizing Compiler for Java. In *JAVA '99: Proceedings of the ACM 1999 conference on Java Grande*, pages 129–141. ACM Press, 1999.

[5] Peter Canning, William Cook, Walter Hill, and Walter Olthoff. F-Bounded Polymorphism for Object Oriented Programming. In *ACM Symposium on Functional Programming Languages and Computer Architecture (FPCA)*, pages 273–280, 1989.

[6] Luca Cardelli. Type Systems. In *Handbook of Computer Science and Engineering, Chapter 103*, 1997.

[7] Luca Cardelli. Basic Polymorphism Typechecking. In *Science of Computer Programming*, pages 8 (2): 147–172, April 1987.

[8] R. Cartwright and G. L. Steele. Compatible Genericity with Run Time Types for the Java Programming Language. In *OOPSLA'00 Proceedings*. ACM, 2000.

[9] Michael K. Chen and Kunle Olukotun. Exploiting Method Level Parallelism in Single Threaded Java Programs. In *PACT'98 Proceedings*. IEEE, 1998.

[10] 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.

[11] 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.

[12] Y. Chicha, M. Lloyd, C. Oancea, and S. M. Watt. Parametric Polymorphism for Computer Algebra Software Components. In *Proc. 6th International Symposium on Symbolic and Numeric Algorithms for Scientific Comput.*, pages 119–130. Mirton Publishing House, 2004.

[13] James P. Cohoon and Jack W. Davidson. *C++ Program Design: An Introduction to Programming and Object-Oriented Design, 2nd Edition.* Boston: McGraw-Hill, 1999.

[14] Microsoft Corporation. DCOM Technical Overview. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndcom/html/msdn_dcomtec.asp, 1996.

[15] Haskel B. Curry and Robert Feys. *Combinatory Logic, volume 1.* North Holland, second edition, 1968.

[16] Luis Damas and Robin Milner. Principal Type-Schemes for Functional Programs. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 207–212, 1982.

[17] Atul Saini David R. Musser, Gillmer J. Derge. *STL Tutorial and Reference Guide, Second Edition.* Addison-Wesley (ISBN 0-201-37923-6), 2001.

[18] James Donahue and Alan Demers. Data Types are Values. In *ACM Transactions in Programming Languages and Systems*, pages 426–445, 1985.

[19] 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. World Scientific Pub, 2002.

[20] Jim Farley. *Java Distributed Computing.* O'Reilly, 1998. "Wiley computer publishing.".

[21] Matthew Fluet and Riccardo Pucella. Phantom Types and Subtyping. In *IFIP TCS*, pages 448–460, 2002.

[22] Jean Yves Girard. Interpretation Fonctionnelle et Elimination des Coupures de l'Arithmetique d'Ordre Superieur. In *Proceedings of the Second Scandinavian Logic Symposium*, pages 63–92. North-Holland, 1971.

[23] William A. Howard. The formulas-as-types notion of construction. In *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*, pages 479–490. Academic Press, New York, 1980.

[24] Atsushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: A Minimal Core Calculus for Java and GJ. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 1999.

[25] Ralph E. Johnson. Type Object. In *EuroPLoP*, 1996.

[26] M. P. Jones. A System of Constructor Classes: Overloading and Implicit Higher-Order Polymorphism. In *Proc. Functional Programming Languages and Computer Architecture*, pages 52–61. ACM, 1993.

[27] Iffat H. Kazi and David J. Lilja. JavaSpMT: A Speculative Thread Pipelining Parallelization Model for Java Programs. In *PPOPP'01 Proceedings*. ACM, 2001.

[28] Kate Keahey. A Brief Tutorial on CORBA, http://www.cs.indiana.edu/ kksiazek/tuto.html.

[29] Andrew Kennedy and Don Syme. Design and Implementation of Generics for the .NET Common Language Runtime. In *Proceedings of the ACM SIGPLAN 2001 conference*, 2000.

[30] Seon Wook Kim, Rudolf Eigenmann Chong-Liang Ooi, Babak Falsafi, and T. N. Vijaykumar. Reference Idempotency Analysis: A Framework for Optimizing Speculative Execution. In *PPOPP'01 Proceedings*. ACM, 2001.

[31] Henry Ledgard. *ADA An Introduction/ADA Reference Manual*. Springer-Verlag, New York, 1981.

[32] Barbara Liskov, Russell Atkinson, Toby Bloom, Elliott Moss, J. Craig Schaffert, Robert Scheifler, and Alan Snyder. *CLU Reference Manual*. Springer-Verlag, 1981.

[33] D. MacQueen. An Implementation of SML Modules. In *Conference on Lisp and Functional Programming*, pages 212–223. ACM, 1988.

[34] Donis Marshall. *.NET Security Programming*. Wiley, 2003.

[35] N. McCracken. The Typechecking of Programs with Implicit Type Structure. In *Semantics of Data Types, LNCS n.173*, pages 301–316. Springer-Verlag, 1984.

[36] Microsoft. An Introduction to Microsoft .NET Remoting Framework. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndotnet/html/introremoting.asp.

[37] Robin Milner. A Theory of Type Polymorphism in Programming. In *Journal of Computer and System Sciences, Vol. 17*, pages 348–375, 1978.

[38] M. B. Monagan, K. O. Geddes, K. M. Heal, G. Labahn, S. M. Vorkoetter, J. McCarron, and P. DeMarco. *Maple 9 Advanced Programming Guide*. Maplesoft, 2003.

[39] M. Moreno Maza. Technical Report TR 4/99, On Triangular Decompositions of Algebraic Varieties. Technical report, NAG Ltd, Oxford, UK, 1999.

[40] Greg Nelson. *Systems Programming with MODULA-3*. Prentice Hall, 1991.

[41] Inc. Numerical Algorithms Group. The FRISCO Project, http://www.nag.co.uk/projects/frisco.html.

[42] C. Oancea and S. M. Watt. A Framework for Using Aldor Libraries with Maple. In *Actas de los Encuentros de Algebra Computacional y Aplicaciones*, pages 219–224, 2004.

[43] C. E. Oancea, J. W. A. Selby, M. Giesbrecht, and S. M. Watt. Distributed Models of Thread-Level Speculation. In *Proceedings of the PDPTA'05*, pages 920–927, 2005.

[44] C. E. Oancea and S. M. Watt. Domains and Expressions: An Interface between Two Approaches to Computer Algebra. In *Proceedings of the ACM ISSAC 2005*, pages 261–269, 2005.

[45] C. E. Oancea and S. M. Watt. Parametric Polimorphism for Software Component Architectures. In *Proceedings of the ACM OOPSLA*, 2005.

[46] M. Odersky, P. Wadler, G. Bracha, and D. Stoutamire. Pizza into Java: Translating Theory into Practice. In *24th ACM Symposium on Principles of Programming Languages*, pages 146–159. ACM, 1997.

[47] M. Odersky, P. Wadler, G. Bracha, and D. Stoutamire. Making the Future Safe for the Past: Adding Genericity to the Java Programming Language. In Craig Chambers, editor, *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, pages 183–200, 1998.

[48] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang. The Case for a Single-Chip Multiprocessor. In *ASPLOS-VII: Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, pages 2–11. ACM Press, 1996.

[49] OMG. Common Object Request Broker Architecture — OMG IDL Syntax and Semantics. Revision2.4 (October 2000), OMG Specification, 2000.

[50] OMG. Common Object Request Broker: Architecture and Specification. Revision2.4 (October 2000), OMG Specification, 2000.

[51] OpenMath. Special Issue on OpenMath. *ACM SIGSAM Bulletin*, 34(2), June 2000.

[52] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 2002.

[53] G. D. Plotkin. A Note on Inductive Generalization. In *Machine Intelligence*, pages 153–163, 1970.

[54] M. Prvulovic, M. J. Garzar, L. Rauchwerger, and J. Torrellas. Removing Architectural Bottlenecks to the Scalability of Speculative Parallelization. In *ISCA '01: Proceedings of the 28th annual international symposium on Computer architecture*, pages 204–215. ACM Press, 2001.

[55] James Purtilo. Applications of a Software Interconnection System in Mathematical Problem Solving Environments. In *Symposium on Symbolic and Algebraic Manipulation (SYMSAC 86)*, pages 16–23. ACM, 1986.

[56] John C. Reynolds. Transformational Systems and the Algebraic Structure of Atomic Formulas. In *Machine Intelligence, 5(1)*, pages 135–151, 1970.

[57] John C Reynolds. Towards a Theory of Type Structure. In *Proc. Colloque sur la Programmation*, pages 408–425. Springer-Verlag LNCS 19, 1974.

[58] John C Reynolds. *Theories of Programming Languages*. Cambridge University Press, 1998.

[59] P. Rundberg and P. Stenstrom. An All-Software Thread-Level Data Dependence Speculation System for Multiprocessors. *The Journal of Instruction-Level Parallelism*, 1999.

[60] Y. Sazeides and J. E. Smith. The Predictability of Data Values. In *MICRO 30: Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 248–258. IEEE Computer Society, 1997.

[61] Victor Shoup. NTL: A Library for Doing Number Theory, http://www.shoup.net/ntl/doc/tour.html.

[62] Bjarne Stroustrup. A History of C++: 1979-1991. In *Paper and talk transcript from History of Programming Languages II conference*, 1993.

[63] Sun. Java Native Interface Homepage, http://java.sun.com/j2se/1.4.2/docs/guide/jni/.

[64] J. M. Tendler, J. S. Dodson, J. S. Fields Jr., H. Le, and B. Sinharoy. POWER4 System Microarchitecture. *IBM Journal of Research and Development*, 46(1):5–26, 2002.

[65] Sami Vaisanen. Microsoft Component Technology Concepts: ActiveX, COM, DCOM, COM+, 2005.

[66] Mirko Viroli and Antonio Natali. Parametric Polymorphism in Java: an Approach to Translation Based on Reflective Features. In *OOPSLA'00 Proceedings*, pages 146–165. ACM, 2000.

[67] 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.

[68] S. M. Watt. Aldor. In J. Grabmeier, E. Kaltofen, and V. Weispfenning, editors, *Handbook of Computer Algebra*, pages 154–160, 2003.

[69] S.M. Watt, P.A. Broadbery, S.S. Dooley, P. Iglio, J.M. Steinbach, and R.S. Sutor. A First Report on the $A^\sharp$ Compiler. In *ISSAC 94 Proceedings*, pages 25–31. ACM, 1994.

[70] Stephen M. Watt, Peter A. Broadbery, Samuel S. Dooley, Pietro Iglio, Scott C. Morrison, Jonathan M. Steinbach, and Robert S. Sutor. *AXIOM Library Compiler User Guide*. Numerical Algorithms Group (ISBN 1-85206-106-5), 1994.

[71] Dachuan Yu, Andrew Kennedy, and Don Syme. Formalization of Generics for the .NET Common Language Runtime. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2004.

[72] Antonia Zhai, Cristopher B. Colohan, J. Gregory Steffan, and Todd C. Mowry. Compiler Optimization of Scalar Value Communication Between Speculative Threads. In *ASPLOS X 2002 Proceedings*. ACM, 2002.

[73] Craig Zilles and Gurindar Sohi. Master/Slave Speculative Parallelization. In *Micro-35 Proceedings*. ACM, 2002.