Generic IDL: Parametric Polymorphism for Software Component Architectures

Cosmin Oancea (coancea@csd.uwo.ca)
Stephen Watt (watt@csd.uwo.ca)

Motivation:

- Multilanguage Architectures
  - components are developed independently, and then combined to construct applications
  - have lagged behind in exposing new language ideas
- Our Goals
  - extend Software Component Architectures, making them competitive for multi-language programming using modern language constructs efficiently
  - assisted interface generation for generic libraries
  - explore optimizations in multi-language environments
  - optimizations of deeply composed generics

Parametric Polymorphism

- one mechanism to support generic programming
- increases the flexibility, reusability, expressive power, avoids the need for down-casting, and ensures type inference termination in some higher order lang.
- various semantics in different prog. lang.
  (C++, Modula, GJ, Ada, ML, Aldor)
- a general mechanism should accommodate both:
  - compile and run time type instantiation
  - qualified/free type variables

Multilanguage Environments

- Extern C
- Java Native Interface
- CORBA
- DCOM
- .NET

- Parametric Polymorphism has become a common feature of mainstream programming languages, but SCAs have not as yet exposed it

Early Experiment: FRISCO project (1997)

- Objective: allow Aldor programs to make use of the PoSSo library (heavy use of C++ templates)

- Aldor: strongly typed functional language, with a higher order type system:
  - each value belongs to some unique type: its domain;
  - domains can be created at run time by user defined functions
  - domains belong to type categories (can be statically determined)
  - explicit p.p. through dependent types

$x \in D \in C$ vs. $x \in S \subset C$

Early Experiment (conclusions)

- Through clever use of virtual functions, we were able to:
  - produce proper binding time semantics by prototypic instantiation of templates
  - produce lightweight proxies to make hierarchies available on either side of the language interface

- Conclusions:
  - the C++/Aldor semantics gap can be overcome
  - (objects vs. type-categories and compile vs. run time bindings for generics)
  - a general, well defined semantics for p.p. can be constructed
  - (for which C++/Aldor mappings are particular solutions)
  - need a systematic solution that encompasses more languages ***GIDL***

Introduction to GIDL

- mappings to C++, GJ, Aldor

- type variables may be qualified:

  extend based qualification T : B

  export based qualification T :- B

```
interface Foo {  void foo(); };
interface Foo_extend : Foo {};
interface Foo_impl { void foo(); }; // not in an isA relation with Foo

interface Test<T1 : Foo, T2 :- Foo> {  void print(T a); };
interface Main {
  Test< Foo_extend, Foo_extend > op1(); //OK
  Test< Foo_extend, Foo_impl     > op2(); //OK
  Test< Foo_impl,    Foo_impl     > op3(); //Error
};
```

GIDL’ s model for generics

- allows generic type qualifications
  - generic type has a well defined meaning (context independent)
  - precise, easily extensible GIDL specifications
- natural mappings to common prog. langs. within a small overhead cost
- homogeneous implementation approach, based on a type-erasure technique
  - preserves backward compatibility
  - works on top of any CORBA vendor implementation

Type Checking

- generic types are attached to GIDL interfaces
- the visibility scope is throughout the defining interface
- sub-typing is defined to be invariant with respect to the type variables
  - List<S> $\not\subset$ List<T> , even if S $\subset$ T
  - guarantees type checking termination for mutual recursive generic type bounds
- the extend qualification is stronger than the export one:

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

Type Checking Example

```
interface Comp<A> {
  boolean compare(in A a);
};

interface Double : Comp<Float> {..};
interface Float : Comp<Double> {..};

interface Comparator<A: Comp<B>, B : Comp<A>> {
  Comparator<Comp<B>, Comp<A>> op3(); //** Error
  Comparator<Double, Float> op4();             //*   OK
}
```

// Comp<B> should extend Comp<Comp<A>>**

//   (False since then B==Comp<A>)

//* Double extends Comp<Float> by def., so true

GIDL translator output

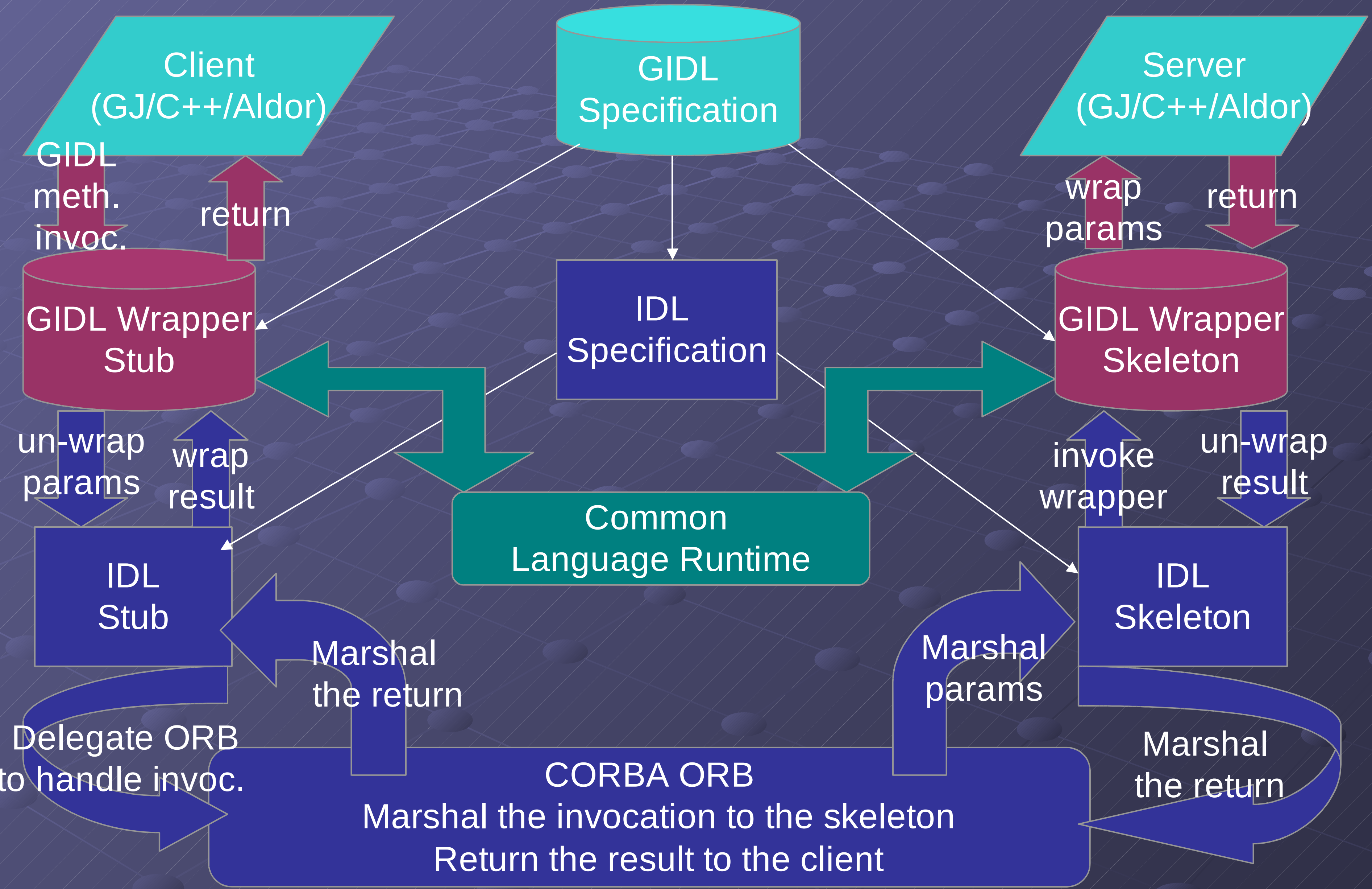
- erasure technique: GIDL => IDL
  - preserves the backward compatibility
  - translator works on top of any CORBA implementation
  - can generate proxies (extern C/JNI/..) and link them in a single process environment
    - opportunities for cross file, inter-language optimizations

- recover the lost generic type information at the mapped language skeleton/stub wrapper level

```
//GIDL
interface Test<T,
P:ExtQual, Q:-ExpQual> {
  T  op1();
  P  op2();
  Q op3(Test<T,P,Q> a);
};
```

```
//IDL
interface Test {
  any  op1();
  ExtQual  op2();
  Object op3(Test a);
};
```

GIDL Base Application Architecture



Using the Architecture

- Server side: inherits and implements the GIDL skeleton wrappers
- Most of the implementation details are hidden
- Now client/server may use generic programming as desired

```
// GIDL Specification
interface GPriorElem<A:-GPriorElem<A>>
{
  short getPriority();
  short compareTo(in A r);
  A createNewA(in short s);
};

interface PriorElem :
        GPriorElem<PriorElem>{};

interface PriorQueue<A:-GPriorElem<A>>
{
  void enqueue(in A a); A dequeue();
  boolean empty();        short size();
  A createPriorElem(in short s);
};
```

```
// code excerpt from a C++ CLIENT
1. CORBA::Object_var obj = orb->string_to_object
      (s);
2. GIDL::PriorQueue<GIDL::PriorElem> gpq
      (pq_orig);
3. GIDL::PriorElem gPEobj = gpq.createPriorElem
      (GIDL::Short_GIDL(1));
4. gpq.enqueue(gPEobj);
//Obtain a reference to a CORBA::Object – obj
5. gpq.enqueue(obj); //ERROR
6. gPEobj = gpq.dequeue();
7. GIDL::Short_GIDL sh = gPEobj.getPriority();
8. cout<<sh<<endl; //prints " 1"
```

GIDL to C++ Mapping

- follows closely CORBA-C++ mapping ideas: scopes scopes, modules namespaces, interfaces (generic) classes
- C++ wrappers (erased) CORBA reference

  + associated generic type inf. + two way casting + functionality
- export/extend base qualification mapping introduce no run-time overhead
  - their implementation relies on C++' s static binding time

GIDL to C++ Mapping Example

```
// 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); };
```

```
template<class T1, class T2, class T3> class
Test : virtual public ::GIDL::GIDL_Object {
  protected: ::Test_var* obj;
  private:
  virtual void implTestFunction() {
    if(1) return;
    T2 a_T2; T1 a_T1; Foo f = (Foo)a_T1;
    GIDL::String_GIDL t=a_T2.tostring();
  }
  public:  Test(::Test_var ob) {
    obj = new ::Test_var(ob); implTestFunction();
  }
  static ::Test_var _narrow(Test<T1, T2, T3> o) {…}
  static Test<T1, T2, T3> _lift(CORBA::Object_var o) { ..}
  static Test<T1, T2, T3> _any_lift(CORBA::Any_var a) {..}
  static CORBA::Any_var _any_narrow(Test<T1,T2,T3> w){..}
```

```
virtual GIDL::Foo op(T1 a1, T2 a2,
                 T3 a3, GIDL::Foo a4) {
  ::Foo_var a = a1._narrow(a1);
  CORBA::Object_var b=
    a2._narrow(a2);
  CORBA::Any_var c=
    a3._any_narrow(a3);
  ::Foo_var d = a4._narrow(a4);
  ::Foo_var a0=(*obj)->op(a, b, c, d)
  GIDL::Foo ret; return ret._lift(a0);
}
}
```

GIDL to GJ Mapping

- same main ideas as the C++ mapping
- user’ s help is required, as GJ does not support:
  - generic type object instantiation,
  - reflective features for the generic types
- new scopes GJ packages
- GIDL’ s implicit parametric structures generic classes

```
// GIDL specification
interface Base<C:Object, D, E> {
  typedef struct BaseStruct {
    C field_C;
    E field_E;
  };
};
```

```
package GIDL.Base; import GIDL.*;
public final class BaseStruct
<C extends GIDL_Object, E extends GIDL_Value>
implements GIDL_Value {
  private C c; private E e;
  private org.omg.CORBA.Object obj;
  public BaseStruct(C c, E e,
    org.omg.CORBA.Object ob){
      this.c=c; this.e=e; this.obj=ob;
  } /* …*/ };
```

Export Qualification Mapping
Most General Generic Unifier (MGGU)

- <A:-Type> compute the MGGU for A,
  w.r.t. all the types in the specification
- use unification algo. to minimize the # of generic types and the # of MGGUs
- preserve the inheritance hierarchy among MGGUs

GIDL

```
interface Tp1<A:-Tp1<A>>              {..};  // A   MGGU1
interface Tp2<B:-Tp2<B>>: Tp1<B> {..};  // B   MGGU2
```

GJ

GJ

```
interface Tp1<A implements MGGU1<A>> extends MGGU1<A>{..};
interface Tp2<B implements MGGU2<B>> extends Tp1<B>,
                                                   MGGU2<B>{..};
```

IFF

```
interface MGGU2<T> extends MGGU1<T> {..};
```

MGGU (continuation)

GIDL

```
interface Element        { tp0 op(in tp1 a, in tp2 b); };
interface GenEl1<T,P> { P    op(in  T   a, in tp2 b); };
interface GenEl2<T,P> { tp0 op(in  P   a, in  T   b); };
interface Test<A:-Element> { /* use A */ };
```

GJ

```
interface MGGU<T,P,Q> { T op(in P a, in Q b); }

interface Element        extends MGGU<tp0, tp1, tp2>{..}
interface GenEl1<T,P> extends MGGU<P,    T,    tp2>{..}
interface GenEl2<T,P> extends MGGU<tp0, P,    T   >{..}

interface Test<A implements   MGGU<tp0, tp1, tp2>> {..}
```

Semi-Automatic STL Translation

- Library interface   GIDL specification   stub/ skeleton + implementation
  (STL == black box TAT^{-1} scheme is applied).

- STL:
  6 components: containers, generic algorithms, iterators, function objects, adaptors, allocators
  orthogonal components   by using iterators (abstract data accessing methods)
  - each container/algorithm provides/requires certain iterator’ s categories – specified in English; we can do better with GIDL

Translation design

- enforces component orthogonality at the lang. level
- iterators/containers design is non-intrusive (do not assume any inheritance relation)

```
interface InputIterator<T, It:-Iterators::InputIterator<T, It>> {..};

interface STLvector<T, Ite:-Iterators::RandAccessIterator<T, Ite>,
   II:-Iterators::InputIterator<T,II> >{..};

interface InpIterator<T> : InputIterator<T, InpIterator<T>>  {..};
```

Difficulties in Translating STL

- STL call by value; GIDL-STL application level call by reference
  - Provide *clone()* and *destroy()* methods for GIDL-STL objects (create/destroy CORBA objects).
  - Big overhead when using iterators (since they are just supposed to be pointers)

- Optimization is needed!!!

```
//STL internal implementation
interface  FindAlg<T, It:-InputIterator<T,It>>
{ It find(in It first, in It last, in T val); }
```

```
//C++ STL implementation for find:
while(first<last) {
    //.....
    first++;
}
```

Multi-Language Environment Optimizations

- We have covered the declarative aspect:

  way of having software typed in one program

- Ultimate goal: optimization of modules with p.p. in a multi-language environment
  - Inter-procedural, inter-file optimizations between programs in different languages (inlining, .., etc.)
  - Macroscopic optimization: speculative(optimistic) / semantic driven optimizations (eg: library translation)

Conclusions:

- Exposed parametric polymorphism to software component architectures
- Qualification of type parameters can be enforced in various target languages, and come with small overhead penalty
- Semi-automatic generic library translation
- Opportunity for inter-language optimizations