Guile-GNOME: CORBA

version 2.16.0, updated 12 June 2008

Andy Wingo (wingo at pobox.com)
Martin Baulig (baulig at suse.de)

This manual is for Guile-GNOME: CORBA (version 2.16.0, updated 12 June 2008)

Short Contents

1 (gnome corba)

1.1 Overview

A CORBA wrapper for Guile.

1.2 Opening CORBA modules

`(gnome corba)` allows full integration between Scheme and remote CORBA objects. However, the problem is how to get type information about these remote objects – it's not very useful to have an opaque `<CORBA:Object>` in Scheme. It's also not very useful if you can't write your own CORBA servants in Scheme.

Basically, there are two ways to solve this problem. You can parse the type's interface description language (IDL) at runtime, or you can get the necessary information from some other source. `(gnome corba)` does the latter, via so-called "imodules".

Imodules are a feature of ORBit2, the ORB used in GNOME. ORBit2 is a CORBA 2.4-compliant Object Request Broker (ORB), which is of course interoperable with other ORB implementations. An imodule is a shared library, installed as `$(libdir)/<modulename>_imodule.la`. To create such a library for your own IDL, you need to run ORBit2's IDL compiler, `orbit-idl`, with the '`--imodule`' argument. See the `demos/corba/` directory in this distribution for an example.

As an example, the rest of this section will refer to the sample IDL which can be found in `demos/corba/Foo.idl` in this distribution.

Once you have installed the `Foo` CORBA library (including its imodule), you can load its type information into Scheme by calling:

```
(corba-primitive-open-module "Foo")
```

As a side effect, this call will define all of the GOOPS classes and methods associated with the `Foo` module.

If there is a CORBA interface `Foo::Hello`, `corba-primitive-open-module` will create a GOOPS class `<Foo:Hello>` which serves as stub class and another GOOPS class `<POA:Foo:Hello>` which serves as skeleton class.

All stub classes are derived from `<CORBA:Object>` and their CORBA class hierarchy is preserved in Scheme.

All skeleton classes are derived from `<PortableServer-ServantBase>` and their CORBA class hierarchy is preserved as well.

1.3 Calling CORBA Methods

To call a CORBA method, all you need to do is to invoke the corresponding method in the stub class. Let's assume `hello` is an instance of the `<Foo:Hello>` class. We may invoke the `Foo::Hello::doHello` method directly, in a most Schemely fashion:

```
(Foo:Hello:doHello hello)
```

So to call CORBA methods, you don't even need to know that it's CORBA.

If a CORBA exception is signalled, a Scheme error will be thrown to the key `corba-system-exception` or `corba-user-exception`, as appropriate.

1.4 Implementing CORBA servants

The interesting part is to implement CORBA servants in Scheme. Let's assume you want to write a servant for the `Foo::Hello` interface.

The first thing you need to do is to derive its POA class

```
(define-class <hello> (<POA:Foo:Hello>))
```

Then, you define methods:

```
(define-method (Foo:Hello:doHello (hello <hello>))
  (display (list "Hello World!" hello)) (newline))
```

If you call `(next-method)`, the POA class' method will be run, which by default will throw a `CORBA::NO_IMPLEMENT` system exception.

However, you can override this:

```
(define-method (Foo:Bar:Baz:haveFun (object <POA:Foo:Bar:Baz>) a b)
  (display (list "Default Foo:Bar:Baz:haveFun handler!" a b))
  (newline))
```

If you created all the methods, you can create servants and call `corba-servant->reference` to get a `CORBA::Object` reference:

```
(define servant (make <hello>))
(define hello (corba-servant->reference servant))
```

Now you have a CORBA Object `hello`, and can invoke methods on it:

```
(Foo:Hello:doHello hello)
```

Although this looks like a normal Scheme procedural application, this is a "real" CORBA call: `hello` is a "normal" CORBA Object.

Note of course that any CORBA Objects which you create in Guile are "owned" by Guile's garbage collector, so make sure to `CORBA_Object_duplicate()` in a C function before you store it somewhere.

1.5 Multiple inheritance

Like in C, you can also create servants for CORBA interfaces which are derived from other interfaces:

```
(define-class <maximum> (<hello> <POA:Foo:MaximumHello>))
(define-method (Foo:Hello:doHello (hello <maximum>))
  (display (list "Hello Maximum World!" hello))
  (newline)
  (next-method))
```

```
(define maximum-servant (make <maximum>))
(define maximum (corba-servant->reference maximum-servant))
```

This creates a new servant for the CORBA interface `Foo::MaximumHello` which is derived from `Foo::Hello` and `Foo::Bar::Baz`. This inheritance is reflected in Scheme.

```
;; Calls method 'Foo:Hello:doHello' in class <maximum> and then
;; in <hello> because of the (next-method).
(Foo:Hello:doHello maximum)

;; Calls method 'Foo:Bar:Baz:haveFun' in class <POA:Foo:Bar:Baz>,
;; the default handler.
(Foo:Bar:Baz:haveFun maximum 1 2)
```

Since we're using real CORBA calls, all of this also works for calls which are coming "from the outside", i.e. from C or from a remote process.

1.6 An important limitation

CORBA servants can be implemented either in C or in Scheme, but you cannot mix them.

For example, in the example above, you learned how to create a CORBA servant for the `Foo::MaximumHello` CORBA interface in Scheme. Now let's assume you already have an implementation for the `Foo::Hello` interface in C.

Even if `Foo::MaximumHello` is derived from `Foo::Hello`, you cannot use the `Foo::Hello` C implementation in Scheme.

This limitation may sound obvious, but it's not so obvious at all if you're a bit familiar with CORBA. In C, you would normally expect to have a `vepv` and a `epv` vector in a CORBA servant, and to be able to poke around in the vepv to override methods.

As an ORBit2 specific implementation detail, servants which you create from Scheme don't have a `vepv` at all and the `epv` is not what you'd expect – the `epv` entries are Scheme vectors and not pointers to C functions.

1.7 CORBA structs / sequences

There is also support to access CORBA structs / sequences from Scheme, including a special record type for structs. See the source code for details.

1.8 Usage

`<CORBA:Object>` [Class]

`<PortableServer-ServantBase>` [Class]

`corba-record-type-vtable` [Variable]

`bonobo-get-object` *moniker class* [Primitive]

`bonobo-object-query-interface` *object class* [Primitive] `corba-servant->reference` *servant* [Primitive]

`corba-record-accessor` *rtd field-name* [Function]

`corba-record-constructor` *rtd . opt* [Function]

`corba-record-constructor-from-struct` *rtd* [Function]

`corba-record-modifier` *rtd field-name* [Function]

`corba-record-predicate` *rtd* [Function]

`corba-record-type-descriptor` *obj* [Function]

`corba-record-type-fields` *obj* [Function]

`corba-record-type?` *obj* [Function]

`corba-record-typecode` *obj* [Function]

`corba-record?` *obj* [Function]

`corba-sequence->list` *sequence* [Function]

`corba-struct->record` *struct* [Function]

`gnome-corba-error` *format-string . args* [Function]

`make-corba-record-type` *typecode . opt* [Function]

2 (gnome corba primitives)

2.1 Overview

A CORBA wrapper for Guile.

2.2 Usage

`<CORBA:Object>` [Class]

`<PortableServer-ServantBase>` [Class]

`corba-primitive-find-poa-class` *class* [Primitive]

`corba-primitive-invoke-method` *method_name imethod class args* [Primitive]

`corba-primitive-main` [Primitive]

`corba-primitive-make-poa-instance` *class* [Primitive]

`corba-primitive-open-module` *name* [Primitive]

`corba-primitive-register-interface` *name* [Primitive]

`corba-primitive-typecode->class` *type* [Primitive]

`corba-typecode-primitive->name` *typecode* [Primitive]

`corba-typecode-primitive?` *typecode* [Primitive]

3 (gnome corba types)

3.1 Overview

A CORBA wrapper for Guile.

3.2 Usage

`%corba-sequence-vtable` [Variable]

`%corba-sequence-vtable-offset-printer` [Variable]

`%corba-sequence-vtable-offset-user` [Variable]

`%corba-struct-vtable` [Variable]

`%corba-struct-vtable-offset-printer` [Variable]

`%corba-struct-vtable-offset-user` [Variable]

`corba-object-class->typecode` *class* [Primitive]

`corba-sequence-length` *corba_sequence* [Primitive]

`corba-sequence-ref` *corba_sequence index* [Primitive]

`corba-sequence-set!` *corba_sequence index value* [Primitive]

`corba-sequence-set-length!` *corba_sequence length* [Primitive]

`corba-sequence-type` *corba_sequence* [Primitive]

`corba-struct-fields` *typecode* [Primitive]

`corba-struct-is-a?` *corba_struct typecode* [Primitive]

`corba-struct-ref` *corba_struct index* [Primitive]

`corba-struct-set!` *corba_struct index value* [Primitive]

`corba-struct-type` *corba_struct* [Primitive]

`corba-struct?` *corba_struct* [Primitive]

`corba-typecode->gtype-class` *typecode* [Primitive]

`make-corba-sequence` *typecode num_tail_elts init_smob* [Primitive]

`make-corba-struct` *typecode num_tail_elts init_struct* [Primitive]

`gnome-corba-error` *format-string . args* [Function]

Function Index