MCORBA: A CORBA Binding for Mercury

David Jeffery, Tyson Dowd, Zoltan Somogyi

{dgj,trd,zs}@cs.mu.oz.au
Department of Computer Science and Software Engineering
University of Melbourne
Parkville, Victoria 3052, Australia

Abstract. MCORBA is a binding to the CORBA distributed object framework for the purely declarative logic/functional language Mercury. The binding preserves the referential transparency of the language, and has several advantages over similar bindings for other strongly typed declarative languages. As far as we know, it is the first such binding to be bidirectional; it allows a Mercury program both to operate upon CORBA components and to provide services to other CORBA components. Whereas the Haskell binding for COM maps COM interfaces onto Haskell types, MCORBA maps CORBA interfaces onto Mercury type classes. Our approach simplifies the mapping, makes the implementation of CORBA's interface inheritance straightforward, and makes it trivial for programmers to provide several different implementations of the same interface. It uses existential types to model the operation of asking CORBA for an object that satisfies a given interface but whose representation is unknown.

1 Introduction

Purely declarative programming languages have many benefits over imperative languages. Modern declarative programming languages are expressive, efficient, have clear semantics and are amenable to aggressive optimisation, which makes them very attractive tools for large classes of problems.

For declarative languages to be used extensively in real-world programs they cannot simply offer a solution (no matter how elegant) to a particular problem in isolation, nor can they demand exclusive use of one programming language. Declarative languages need to be able to be used as a part of a larger system, or their benefits to a particular subsystem will be ignored in face of the larger concerns of the system. Declarative languages must therefore include comprehensive interfaces with existing languages and systems if they are to see widespread adoption. Put simply, interoperability needs to have high priority in the language design.

We believe the design of the external interfaces of declarative languages should be guided by the following principles:

- The purity and expressiveness of the language should be maintained.
- The natural style of the language should not be unduly restricted (but can certainly be augmented) by an external interface.
- The interface should be as seamless as possible, unless visible seams have some practical benefit.
- The interface should be bi-directional, allowing code written in the declarative language to act both as provider and as consumer of services.

Many other languages and systems have had to address similar interoperability issues in recent years. This has lead to the development and refinement of new software development architectures, including a strong movement towards the development of systems based around components. Each component has a well defined interface but the implementation is relatively unconstrained – the tools, techniques and languages used for each component can be chosen based upon their merits, rather than the demands of the rest of the system.

Component based systems are a scaled up version of programs based on abstract module interfaces, and like their smaller cousins, they come in procedural and object oriented flavours. Procedural systems are based on procedure calls from one component to another, whereas object oriented systems view each component as an object and pass messages between objects.

While at first these systems mainly put components together to form systems on a single machine, the growth of networking and particularly the Internet has lead to the widespread use of distributed component based development, with systems being comprised of components connected via local or wide area networks. This involves either remote procedure calls, or distributed object systems invoking methods over a network.

Component based systems are an excellent opportunity to use declarative languages – their strengths can be put to work in isolation from the concerns of other components. This allows declarative languages to be adopted in a piecemeal fashion, which is an important consideration when programmers must be trained and familiarized with new techniques.

One of the most popular component based systems is CORBA, the Common Object Request Broker Architecture [2]. CORBA is a distributed object framework created and maintained by the Object Management Group (OMG) which represents the input of a large consortium of companies. CORBA is based around a standard Interface Definition Language (IDL). It uses Object Request Brokers (ORBs) to act as an object "bus", which allows easy local or remote communication between objects.

This paper describes *MCORBA*, a CORBA binding for Mercury, a purely declarative logic/functional programming language that is designed for general purpose, large-scale, real-world use. This binding uses type classes and existential types, two language constructs recently added to the Mercury implementation, to provide a natural, expressive, and yet still purely declarative interface to CORBA. Although MCORBA is still only a prototype that does not yet address the full functionality of CORBA objects, we have already used it both to operate upon existing CORBA components from Mercury and to implement CORBA components in Mercury.

H/Direct [1], an interface from Haskell to COM, Microsoft's Component Object Model [9] which is a CORBA competitor, uses some similar techniques to this paper. The lack of support for existential types in Haskell requires H/Direct to model component interfaces as types, The MCORBA approach, which we introduce in this paper, is to model component interfaces as type classes, which we believe to be significantly more natural.

Section 2 gives a brief overview of Mercury. Section 3 introduces CORBA and explains how we interface between CORBA and Mercury. The low-level details of the interface are explored in section 4.

2 Overview of Mercury

Mercury is a pure logic/functional programming language designed explicitly to support teams of programmers working on large application programs. In this section, we will briefly introduce the main features of Mercury, these being its module, mode, determinism and type systems, including two important recent additions to the type system: type classes and existential types. We will use the following simple program as the example for this introduction.

```
:- module example.

:- interface.
:- import_module io.
:- pred main(io:state, io:state).
:- mode main(di, uo) is det.

:- implementation.

main(S0, S) :-
    io:read_line(Result, S0, S1),
    (
        Result = ok(CharList),
        string:from_char_list(CharList, String),
        io:write_string("The input was: ", S1, S2),
        io:write_string(String, S2, S)
    ;
        Result = error(_),
        io:write_string("Some error occurred.\n", S1, S)
    ;
        Result = eof,
        S = S1
    ).
```

Syntax. As this example shows, the syntax of Mercury is very similar to the syntax of Prolog. Identifiers starting with upper case letters represent variables; identifiers starting with lower case letters represent predicate or function symbols. Facts and rules comprise the code of the program, while terms starting with '`:-`' are declarations. Mercury has many more kinds of these than Prolog.

Modules. Some of these declarations are part of the module system. In Mercury, every program consists of one or more modules. Each module defines one or more entities (e.g. types and predicates). Some of these entities are exported from the module; others are private to the module. The declarations of the former are in the interface section of the module while the declarations of the latter are in the implementation section. The definitions, where these are separate from the declarations (as they are for predicates and abstract types) are all in the implementation section. If a module wants to refer to any entity exported from another module, it must first import that module. The name of any entity may (and in case of ambiguity, must) be module qualified, i.e. prefixed by the name of its defining module and a colon.

In this example, the interface section contains the declaration of the predicate '`main`', while the implementation section contains its code. The '`:- pred`' declaration states that main has two arguments, which are both of the type state defined in module io. Since the interface section refers to this entity from module io, it must and does import module io.

Recent versions of Mercury also support nested modules.

Types. Mercury has a strong Hindley-Milner style type system with parametric polymorphism and higher-order types. ':- pred' declarations specify the types of the arguments of predicates. Types themselves are defined like this:

```
:- type io:result(T) ---> ok(T) ; error(io:error) ; eof.
```

This defines the parametric type constructor io:result by stating that values of type io:result(T), where T is a type variable standing in for an arbitrary concrete type, can be bound to one of the three function symbols (data constructors in the terminology of functional programming) ok/1, error/1 and eof/0 (numbers after the slash denote arities). If the value is ok, the type of its argument will be the concrete type bound to T, while if the value is error, its argument will be of type io:error.

Modes. Besides its type declaration, every predicate has a mode declaration, which says which arguments are input, and which are output. (A predicate may have more than one mode, but none of the predicates we discuss in this paper do.) Further, some mode declarations encode information about uniqueness. If the mode of a given argument of a given predicate is out, this represents an assertion by the programmer that the argument will be instantiated by the predicate. If the mode is uo, which stands for *unique output*, this represents a further assertion that the value returned to the caller will not have any other references pointing to it. Similarly, the mode in, represents an assertion that the argument will instantiated by the caller. The mode di, which stands for *destructive input*, represents a further assertion that the value passed by the caller for this argument does not have any other references pointing to it. Therefore the predicate may destroy the value once it has finished using it.

The Mercury runtime system invokes the predicate main/2, which is analogous to the function main() in C, with a unique reference to an io:state, and requires main to return a unique reference to another io:state. io:state is a type representing the state of the world. It is an abstract type, whose name is exported from module io but whose definition is not, which means that only predicates in module io can manipulate values of type io:state. Predicates in other modules cannot do anything with such values except give them as arguments in calls to predicates in module io. By design, all these predicates have two arguments of type io:state, one whose mode is di and one whose mode is uo. Together, these constraints ensure that at any point in the forward execution there is exactly one live io:state. (The io:state arguments threaded through the program are usually hidden by definite clause grammar (DCG) notation.)

Determinism. To allow the predicates in module io to update the state of the world destructively without compromising referential transparency, we must make sure that versions of the io:state that have been destructively updated cannot become live after backward execution either. This requires us to reason about where backtracking may occur in the program. To this end, we associate with each mode of each predicate a determinism, which puts lower and upper bounds on the number of times the predicate may succeed when called in that mode. The determinism 'det' asserts that the predicate will succeed exactly once; 'semidet' asserts that it will succeed at most once; 'multi' asserts that it will succeed at least once; and 'nondet' means that it may succeed any number of times.

The Mercury compiler can prove that `main` above succeeds exactly once, and that backtracking can never cause obsolete `io:state`s to become live, by noting that all the predicates called by `main` are declared `det`, that the unification `S = S1` acts as an assignment and is therefore `det`, and that since the type of `Result` is `io:result(list(char))`, exactly one of the three unifications involving `Result` will succeed regardless of what value `io:read_line` gives to `Result`. The Mercury compiler is required to prove *all* the assertions contained in `:- pred` and `:- mode` declarations.

Type classes. Mercury has always supported parametric polymorphism (unconstrained genericity). Recent versions of Mercury also support constrained genericity in the form of type classes [5], a language construct we borrowed from functional languages which is similar to Java's interfaces. A '`:- typeclass`' declaration such as

```
:- typeclass printable(T) where [
    pred print(T, io:state, io:state),
    mode print(in, di, uo) is det,
].
```

introduces a new type class, in this case the type class `printable/1`, and gives the signatures of the methods that must be implemented on a type if that type is to be considered a member of that type class. Specific types can then be declared to be members of this type class by giving implementations for the methods listed in the type class declaration. For example, given the predicate

```
:- pred io:write_int(int, io:state, io:state).
:- mode io:write_int(in, di, uo) is det.
```

the `instance` declaration

```
:- instance printable(int) where [
    pred(print/3) is io:write_int
].
```

makes the `int` type a member of the `printable/1` type class. This in turn makes it possible to pass `int`s where a printable type is needed. For example, the declaration of the predicate

```
:- pred print_list(list(T), io:state, io:state) <= printable(T).
:- mode print_list(in, di, uo) is det.
print_list([], S, S).
print_list([T | Ts], S0, S) :-
    print(T, S0, S1),
    print_list(Ts, S1, S).
```

has a *type class constraint* which requires the first argument to be a list of printable items because it invokes the `print` method on each element of this list.

Existential types. The language features we have discussed so far all require the caller to know the concrete types of all arguments. However, sometimes one wants a predicate to return a variable whose type is constrained to be in a particular type class but whose concrete type is unknown to the caller. Existential types provide a solution to this problem [7]; a type variable which is existentially quantified is bound by the *callee* of a predicate. For example, the predicate

```
:- some [T] pred read_printable(T, io:state, io:state)
    => printable(T).
:- mode read_printable(out, di, uo) is det.
```

reads and returns a data item whose type the caller does not know, but on which all the operations of the type class `printable` are defined.

For a detailed explanation of Mercury's type classes, see [4]; for more information on the rest of Mercury, e.g. functional syntax, please see the language reference manual [3].

3 Mercury binding for CORBA

The Common Object Request Broker Architecture (CORBA) is a standard designed to tackle the problem of interoperability between software systems. It allows applications and components to communicate, whether or not they are executing in the same address space, whether or not they are executing on the same machine, and whether or not they are written in the same language.

Communication between software components in a system using CORBA is handled by an Object Request Broker (ORB). The ORB is responsible for establishing and maintaining communication between objects. The ORB is "middleware" through which a client can transparently invoke operations on an object, regardless of the location or implementation of the object. Different ORB implementations may communicate in different ways, so the CORBA 2.0 specification[2] includes a protocol for communication between ORBs, the Internet Inter-ORB Protocol (IIOP).

The interfaces of CORBA components are described in OMG's Interface Definition Language (IDL). An IDL specification defines a set of *interfaces*, each of which is the abstract "signature" of an object. Each interface includes a set of operations and attributes. (Attributes are equivalent to a pair of operations, one that sets the attribute to a value and one that retrieves the value, so we will not mention them further.) For each operation, the interface defines the types of the parameters, and for each parameter, whether the parameter is an input ('`in`'), an output ('`out`') or both ('`inout`'). (This level of detail in the description is necessary for strongly typed languages such as Ada, Eiffel and Mercury, and even for weakly typed languages such as C when different components reside in different address spaces or on different machines.) For example, the following IDL specification:

```
interface CallBack {
    void NewMessage(in string msg);
};
```

defines an interface which has one operation, a one-argument procedure taking a string as input.

A CORBA binding for a particular language provides a way for code written in that language to operate on objects whose interface is described in IDL, and to implement objects whose interface is described in IDL. This binding takes the form of a tool that takes an IDL specification and automatically generates *stubs* and *skeletons* in the target language. When a client written in the target language wants to invoke one of the operations of a CORBA object, it invokes the corresponding stub routine. The stub routine acts as a proxy; it asks the ORB to invoke the same operation on the real object, regardless of its location and implementation language. The ORB communicates with the real object through the skeleton code, which invokes a method of the object when it receives a corresponding request from the ORB.

Standard mappings from IDL are defined for many programming languages including Java, C, C++, and Cobol. For C++, the stub will be a class that passes messages to its real implementation via the ORB. A user who requests an object from the ORB will be given an instance of the stub class, and will be able use it as if the class were implemented locally. The skeleton is an abstract class from which the user must derive a concrete class. Instances of the derived class can then be registered with the ORB so clients can use them, or be passed as parameters to other objects.

This paper may be seen as (the beginnings of) a Mercury binding for CORBA. This binding consists of a run-time library (called `corba`), and a tool (called `mcorba`) which generates Mercury stubs and skeletons from IDL specifications.

`mcorba` generates a Mercury type class for each IDL interface. For each operation of the interface, the type class has a method. The method will have two pairs of arguments with modes `di` and `uo`, to denote the state of the object and the state of the external world before and after the invocation of the method. The types and modes of the remaining arguments will correspond to the types and modes of the arguments of the IDL operation, with the exception that for `inout` parameters, we generate two arguments, with modes `in` and `out`.

From the IDL specification above, `mcorba` would generate

```
:- typeclass callback(T0) where [
    pred newmessage(T0, T0, corba:string, io:state, io:state),
    mode newmessage(di, uo, in, di, uo) is det
].
```

This type class serves two purposes:

- The ORB can give a client an object which belongs to the type class so that the client can invoke the appropriate operations on it (and no others).
- If a Mercury value is of a type which is an instance of the type class, it may be registered with the ORB as an object that satisfies the relevant interface.

We will now examine these two purposes in greater detail.

3.1 The forwards interface: Mercury calls CORBA

The generation of type classes for IDL interfaces, outlined above, is sufficient for a Mercury program to invoke operations on a CORBA object. However, we also need a way of getting a handle on such an object from the ORB. CORBA provides several methods for doing this. Each method is implemented by a predicate in the `corba` module. The predicate

```
:- some [T] pred corba:resolve_object_name(orb, string, string, T,
        io:state, io:state) => corba:object(T).
:- mode corba:resolve_object_name(in, in, in, uo, di, uo) is det.
```

uses the COS (Common Object Services) naming service, through which a client may request an object by name. Note that the client does not request an object of a particular interface type, but merely a generic CORBA object. Since the argument is existentially typed, the implementation of this object is hidden from the client, but it is known to be connected to the ORB since the predicates of the `corba` module guarantee that only such objects satisfy the `corba:object` type class constraint. It is then the client's responsibility to ask the ORB to "narrow" this object into the required interface. It can do so by calling the `narrow` predicate that `mcorba` generates for the desired IDL interface along with the type class. As this operation has the same name for each IDL interface, the code generated for each interface is placed inside a separate Mercury module.

We will show how this is done using as an example a "chat" server with three operations: sending a message to the server for distribution to all registered clients, registering a client, and removing a client. The IDL specification is:

```
interface Chat {
    void SendMessage(in string msg);
    void RegisterClient(in CallBack obj, in string name);
    void RemoveClient(in CallBack obj, in string name);
};
```

From this, `mcorba` generates the following type class:

```
:- typeclass chat(T0) where [
    pred sendmessage(T0, T0, corba:string, io:state, io:state),
    mode sendmessage(di, uo, in, di, uo) is det,
    pred registerclient(T0, T0, T1, corba:string, io:state,
        io:state) <= (callback(T1), corba:object(T1)),
    mode registerclient(di, uo, in, in, di, uo) is det,
    pred removeclient(T0, T0, T1, corba:string, io:state,
        io:state) <= (callback(T1), corba:object(T1)),
    mode removeclient(di, uo, in, in, di, uo) is det
].
```

Once again, there is a type class method corresponding to each IDL interface operation. This time, the `RegisterClient` and `RemoveClient` methods both have arguments which are objects satisfying IDL interfaces. The corresponding arguments of the type class methods are constrained to belong to both the `callback` and `corba:object` type classes, since the method may invoke specific `CallBack` or generic CORBA operations on that argument.

The automatically generated `narrow` operation for the `Chat` interface has the signature:

```
:- type corba:narrow_result(U, N) ---> narrowed(N) ; unnarrowed(U).

:- some [C] pred chat:narrow(O, corba:narrow_result(O, C))
         => (chat(C), corba:object(C)) <= corba:object(O).
:- mode chat:narrow(di, uo) is det.
```

The `narrow` operation may or may not be successful; the CORBA object may or may not actually satisfy the required IDL interface. The ORB finds out whether it does or not. If it does, `narrow` will bind the second argument to `narrowed(Chat)`, where `Chat` is a value which belongs to the `chat` type class as well as `corba:object`. If not, it binds the second argument to `unnarrowed(Obj)`, where `Obj` is the original non-narrowed value.

The following code fragment shows how a Mercury program can use the library predicate `corba:resolve_object_name` and the automatically generated predicate `chat:narrow` to connect to a CORBA chat server. The program can then send the lines of text that it reads from the user to the chat server. (Note that this code uses standard DCG notation to hide two parameters. The curly braces denote goals which do not use the hidden parameters; all other goals have two hidden `io:state` arguments threaded through them.)

```
:- pred sender(corba:orb, io:state, io:state).
:- mode sender(in, di, uo) is det.
sender(Orb) -->
    corba:resolve_object_name(Orb, "Chat", "Server", Obj),
    { chat:narrow(Obj, Narrowed) },
    (
        { Narrowed = narrowed(Chat) },
        sender_loop(Chat)
    ;
        { Narrowed = unnarrowed(_OriginalObj) },
        io:write_string("Couldn't narrow object\n")
    ).

:- pred sender_loop(T, io:state, io:state) <= chat(T).
:- mode sender_loop(di, di, uo) is det.
sender_loop(Chat0) -->
    io:read_line(Res),
    (
        { Res = ok(CharList) },
        { string:from_char_list(CharList, String) },
        sendmessage(Chat0, Chat, String),
        sender_loop(Chat)
    ;
        { Res = error(_Error) },
        io:write_string("Some kind of error occurred\n")
    ;
        { Res = eof }
    ).
```

3.2 The backwards interface: CORBA calls Mercury

Mercury objects can be used to implement a CORBA object by providing an instance of the corresponding type class. For example, we can implement a `Chat` server as follows[1]:

[1] Mercury does not yet support existentially quantified components of data structures, so we cannot store the callbacks in a list. We have instead represented each callback as a `callback_object`, and used Mercury's C interface to implement two predicates:

- `construct_callback_object`, which takes an arbitrary value in type classes `callback` and `corba:object` and produces a value of type `callback_object`.
- `deconstruct_callback_object`, which takes a `callback_object` and produces an existentially typed value in type classes `callback` and `corba:object`.

```
:- type our_server ---> clients(list(callback_object))
:- instance chat(our_server) where [
    pred(sendmessage/5) is our_sendmessage,
    pred(registerclient/6) is our_registerclient,
    pred(removeclient/6) is our_removeclient
].

:- pred our_sendmessage(our_server, our_server, string,
    io:state, io:state).
:- mode our_sendmessage(di, uo, in, di, uo) is det.
our_sendmessage(clients([]), clients([]), _Msg) --> [].
our_sendmessage(clients([CBObj0 | Rest0], clients([CBObj | Rest]),
        Msg) -->
    { deconstruct_callback_object(Obj0, CBObj0) },
    newmessage(Obj0, Obj, Msg),
    { construct_callback_object(Obj, CBObj) },
    our_sendmessage(clients(Rest0), clients(Rest), Msg).
```

If a type has been made an instance of a type class which corresponds to an IDL interface, we need to be able to tell the ORB that a particular object of that type is ready to service requests. We achieve this by generating another predicate for each generated type class. The `is_ready` predicate notifies the ORB that a Mercury value is ready to service requests. For the `Chat` example, we would get:

```
:- some [T2] pred chat:is_ready(corba:object_adaptor, T1, T2,
                io:state, io:state)
        => (chat(T2), corba:object(T2)) <= chat(T1).
:- mode chat:is_ready(in, di, uo, di, uo) is det.
```

A CORBA construct called an *object adaptor* keeps track of which objects are ready to service requests. A given ORB may have more than one object adaptor, so the caller of `chat:is_ready` must choose which object adaptor the object connects to.

With this machinery, our chat server above can make its services available as simply as this:

```
:- pred start_server(corba:orb, corba:object_adaptor,
    io:state, io:state).
:- mode start_server(in, in, di, uo) is det.
start_server(Orb, ObjectAdaptor) -->
    { Server = clients([]) },
    chat:is_ready(ObjectAdaptor, Server, ReadyServer),
    corba:bind_object_name(Orb, "Chat", "Server", ReadyServer),
    corba:implementation_is_ready(ObjectAdaptor).
```

The call to `bind_object_name` registers the object with the name server. The call to `implementation_is_ready` hands control over to the event loop of the object adaptor, which listens for operations on any of the objects that have been registered with the adaptor, and invokes the corresponding methods. Each method invocation will generate a new version of the state of the object, which becomes the state of the object for the next method invocation. These states are not visible from outside of the event loop; the only *visible* effect of the methods is on the I/O state. The I/O state arguments of `implementation_is_ready` (here hidden using DCG notation) capture these effects, just as the I/O state arguments of `io:write_string` do. Therefore method invocations are purely declarative for the same reasons that I/O operations are.

3.3 Other features of CORBA

The CORBA binding we have described above has touched upon only a few of CORBA's features. We will now briefly describe how we handle some other CORBA features.

- One IDL interface may inherit from another. For example, with:

  ```
  interface sub : super {
      ...
  };
  ```

  all of the operations available on `super` are also available on `sub`. We map IDL interface inheritance directly onto type class inheritance. In Mercury, type class inheritance is denoted by a type class declaration itself having type class constraints. In this example:

  ```
  :- typeclass sub(T) <= super(T) where [
      ...
  ].
  ```

  any member of type class `sub` must also be a member of `super`. (Neither IDL nor Mercury allows any kind of implementation inheritance.)
- The CORBA IDL includes several basic types. Many of these (e.g. `double`, `int`, `char`, `string` and `boolean`) can have their values mapped one-to-one to the values of a Mercury type, which allows us to map the IDL type directly to the Mercury type. Some other IDL types (e.g. `float` and `short`), whose values are a subset of the values of a Mercury type, can be handled by mapping them to the Mercury type and having code automatically generated by `mcorba` check their values before giving them to CORBA. The remaining types are harder to handle. We can map them to new, abstract and non-native Mercury types that can then have the appropriate operations defined on them, but whether users will find such an arrangement convenient enough remains a matter for further experimentation.
- The CORBA IDL also includes several ways to build more complex types. Some of these (e.g. enumerations, structures, unbounded length sequences and discriminated unions in which the discriminant is an enumeration) can be mapped to Mercury in a natural manner, though some of these mappings may imply significant conversion costs. The rest (e.g. bounded length sequences and discriminated unions with integer discriminants) do not have obvious direct mappings to Mercury (or to Haskell either [1]), and finding a convenient way to represent them in Mercury is also future work.
- An IDL specification may declare sub-modules which introduce a new (nested) namespace. For each such module we create a Mercury sub-module.
- Constants can also be defined in a specification. For each such constant, we define a function with no input arguments but with a result corresponding to the constant.

- IDL also includes exceptions. We could handle these by mapping CORBA methods onto Mercury predicates that return either an exception indication or their actual result, (the way we handled the `narrow` operation in section 3.1), but we expect that most users would find these too cumbersome. We cannot use the obvious alternative solution, mapping CORBA exceptions to Mercury exceptions, until exceptions formally become part of the Mercury language.
- CORBA allows dynamic lookup of available operations through its Dynamic Invocation Interface (DII), and dynamic creation of operations through the Dynamic Skeleton Interface (DSI). Incorporating DII and DSI into MCORBA is a topic for future work.

4 Implementation

The primary goals of the initial implementation were simplicity and expediency. We wanted to allow easy experimentation with the mapping from IDL to Mercury, and permit handwritten code to be substituted for machine generated code when necessary. We therefore decided that instead of writing a binding which talks to an ORB directly (using IIOP for example), we would simply build on top of an existing binding. The binding we chose to build upon is omniORB 2.0 [8], whose `omniidl2` tool generates stubs and skeletons in C++. Besides generating Mercury stubs and skeletons, `mcorba` also generates the glue that joins the Mercury stub to the C++ stub, and the Mercury skeleton to the C++ skeleton. Since omniORB 2.0 conforms to the CORBA 2.0 standard [2], which specifies a standard C++ binding, our implementation can easily be ported to other ORBs with C++ bindings.

Building a Mercury layer on top of a C++ binding is somewhat complicated by the fact that the current Mercury implementation has an interface only to C, not to C++. Therefore `mcorba` generates C functions that provide a simple interface to C++ code. Since these C functions complicate the explanation of the implementation, in the following discussion we will assume that Mercury can directly call C++. (Building a Mercury layer on top of a C binding would avoid these complications, so when a satisfactory C binding becomes available we will examine it.) We will also simplify the example code by skipping the required error checking and type casts.

4.1 The forwards interface: Mercury uses CORBA components

The stub predicate generated by `mcorba` for a given IDL operation will invoke the corresponding method on the C++ stub object. This requires Mercury code to remember pointers to these objects. When the program calls a predicate such as `is_ready`, which returns a handle on the underlying C++ object, the actual return value will be a pointer to the C++ object. This pointer is a value in the type `corba:cppobject`, whose implementation is known only to the MCORBA system. Since values of type `corba:cppobject` are only ever returned as existentially quantified type arguments belonging to the `corba:object` type class, user code can never break the abstraction.

For each operation in an IDL specification, `mcorba` generates a predicate implemented in C++. This predicate casts the `cppobject` to a pointer to the right C++ class, invokes the requested method, and then creates the new versions of the object and the I/O state.

```
:- pred sendmessage_impl(corba:cppobject, corba:cppobject,
    corba:string, io:state, io: state).
:- mode sendmessage_impl(di, uo, in, di, uo) is det.
:- pragma c_code(sendmessage_impl(Obj0::di, Obj::uo,
        Msg::in, IO0::di, IO::uo), may_call_mercury, "
    Chat_ptr chat = cast_to_chat(Obj0);
    chat->SendMessage(Msg);
    *Obj = cast_from_chat(chat);
    update_io(IO0, IO);
").
```

Given one of these predicates for each method, `mcorba` creates an instance of the chat type class:

```
:- instance chat(corba:cppobject) where [
    pred(sendmessage/5) is sendmessage_impl,
    pred(registerclient/6) is registerclient_impl,
    pred(removeclient/6) is removeclient_impl
].
```

4.2 The backwards interface: Mercury as a CORBA component

Any Mercury type that is an instance of the generated type class for an interface can be used as an implementation of that interface. For the ORB to use the Mercury implementation there needs to be a C++ implementation, and a layer that allows the C++ implementation to call Mercury. To do this, the C++ implementation needs to know how to invoke a given method on a given Mercury object. In C++ this would be done using a virtual function table. Mercury has a similar construct called a *type class dictionary*. A Mercury predicate that manipulates an object of a given type class will have access to a type class dictionary that specifies which predicates implement the methods of the type class on that object. (A type class declaration defines the layout of dictionaries for that type class; an instance declaration specifies its contents.)

`omniidl2` generates an abstract base class for every IDL interface. `mcorba` derives a subclass from each such base class. The derived class simply stores the Mercury object and its type class dictionary as attributes. For our chat example, `omniidl2` generates the base class `_sk_Chat`, while `mcorba` generates the derived class `Chat_i`:

```
class Chat_i : public virtual _sk_Chat {
public:
    Chat_i(void *mcorba_tcdict, void *mcorba_mobj) {
        mcorba_typeclass_dict = mcorba_tcdict;
        mcorba_mercury_object = mcorba_mobj;
    }
    virtual ~Chat_i() {}
    void SendMessage(const char *);
    void RegisterClient(class CallBack *, const char *);
    void RemoveClient(class CallBack *, const char *);
private:
    void *mcorba_typeclass_dict;
```

```
    void *mcorba_mercury_object;
};
```

The methods of this C++ class call the methods of the corresponding Mercury type class. This involves looking up the relevant method in the type class dictionary, and calling it with the appropriate arguments. The Mercury type class implementation already creates code to do the lookup and parameter passing, so we simply export this code using the `pragma export` directive to allow it to be called from C (or, in our case, C++):

```
:- pragma export(sendmessage(di, uo, in, di, uo),
    "chat__sendmessage_method").
```

The implementation of the C++ method can simply call the exported C function with the stored type class dictionary, the original value of the object, and the input arguments of the method. The function will return the updated value of the object and the output arguments of the method through reference parameters.

```
void Chat_i::SendMessage(const char *msg)
{
    chat__sendmessage_method(mcorba_typeclass_dict,
        mcorba_mercury_object, &mcorba_mercury_object, msg);
}
```

Note that the C++ code does not know anything about I/O states, and that the two arguments of type `io:state` are missing. This is possible because Mercury uses values of this type only to enforce referential transparency; they do not actually have any representation.

Each invocation of the `mcorba`-generated `chat:is_ready` predicate creates an instance of the `Chat_i` class and calls the C++ `_sk_Chat::_obj_is_ready` method to register it with the ORB.

5 Conclusions and further work

The main contribution of this paper is showing how IDL interfaces can be mapped onto type classes. Since existential types are an integral part of this approach but are not (yet) part of the standard language specification for Haskell, the Haskell binding for COM [6] could not take this approach, and mapped IDL interfaces onto types instead. The advantages of our approach are that:

- Type classes provide a significantly more natural means of modeling CORBA interfaces. In particular, it is very easy for a program to provide more than one implementation of an interface by providing more than one `instance` declaration for it.
- Inheritance of one interface by another comes “for free”. There is no need to explicitly convert from an inherited-by to an inherited-to interface; the type class constraint mechanism handles this automatically.

Our approach should be easily adaptable to other logic and/or functional languages that have both type classes and existential types, as we expect Haskell 2 will when it arrives.

Our approach also has an advantage over C++ bindings. When an argument of a CORBA operation is another CORBA object, C++ bindings check that the actual parameter satisfies the right interface, but do not check that it has been registered with an ORB. The type class method generated by `mcorba` for the operation requires the corresponding argument to be a member of the `corba:object` type class, which guarantees that any attempt to pass an unregistered object as the actual parameter will be caught at compile time.

We have found our approach to the implementation, which is the use of a thin layer on top of an existing C++ binding, to be very effective. It can be implemented relatively quickly using a simple foreign language interface, and the resulting binding should be easily portable among different ORB implementations for C++.

MCORBA can already generate all the stubs and skeletons required by the chat server. Nevertheless, at the moment it is only a prototype. It doesn't yet handle CORBA exceptions or several CORBA IDL types, and we have not yet started to work on fitting CORBA's Dynamic Skeleton/Invocation Interfaces into our framework. We are working on lifting these limitations.

MCORBA is available from `http://www.cs.mu.oz.au/mercury`. We would like to thank the Australian Research Council for their support.

References

1. Sigbjorn Finne, Daan Leijen, Erik Meijer, and Simon Peyton Jones. H/Direct: a binary foreign language interface for Haskell. In *Proceedings of the 1998 International Conference on Functional Programming*, September 1998.
2. Object Management Group. *The Common Object Request Broker: Architecture and Specification, Revision 2.0.* Object Management Group, 492 Old Connecticut Path, Framingham, MA 01701, U.S.A., July 1996.
3. Fergus Henderson, Thomas Conway, Zoltan Somogyi, and David Jeffery. The Mercury language reference manual. Technical Report 96/10, Department of Computer Science, University of Melbourne, Melbourne, Australia, 1996.
4. David Jeffery, Fergus Henderson, and Zoltan Somogyi. Type classes in Mercury. Technical Report 98/13, Department of Computer Science, University of Melbourne, Melbourne, Australia, 1998.
5. Simon Peyton Jones, Mark Jones, and Erik Meijer. Type classes: an exploration of the design space. In *Proceedings of the Haskell Workshop*, volume 788 of *Lecture Notes in Computer Science*. Springer Verlag, June 1997.
6. Simon Peyton Jones, Erik Meijer, and Daan Leijen. Scripting COM components from Haskell. In *Proceedings of the Fifth International Conference on Software Reuse*, June 1998.
7. Konstantin Läufer. Type classes with existential types. *Journal of Functional Programming*, 6(3):485–517, May 1996.
8. Sai-Lai Lo. *The omniORB2 User's Guide.* Olivetti and Oracle Research Laboratory, March 1997.
9. Dale Rogerson. *Inside COM.* Microsoft Press, 1998.

This article was processed using the LaTeX macro package with LLNCS style