Electronic Notes in Theoretical Computer Science 59 No. 1 (2001)
URL: http://www.elsevier.nl/locate/entcs/volume59.html 17 pages

A framework for interoperability

Kathleen Fisher [1]

AT&T Labs, Research
180 Park Ave., E244
Florham Park, NJ 08932

Riccardo Pucella [2]

Department of Computer Science
Cornell University
Ithaca, NY 14853

John Reppy [3]

Bell Labs, Lucent Technologies
600 Mountain Ave.
Murray Hill, NJ 07974

Abstract

Practical implementations of high-level languages must provide access to libraries and system services that have APIs specified in a low-level language (usually C). An important characteristic of such mechanisms is the *foreign-interface policy* that defines how to bridge the semantic gap between the high-level language and C. For example, IDL-based tools generate code to marshal data into and out of the high-level representation according to user annotations. The design space of foreign-interface policies is large and there are pros and cons to each approach. Rather than commit to a particular policy, we choose to focus on the problem of supporting a gamut of interoperability policies. In this paper, we describe a framework for language interoperability that is expressive enough to support very efficient implementations of a wide range of different foreign-interface policies. We describe two tools that implement substantially different policies on top of our framework and present benchmarks that demonstrate their efficiency.

[1] Email: kfisher@research.att.com
[2] Email: riccardo@cs.cornell.edu
[3] Email: jhr@research.bell-labs.com

1 Introduction

High-level languages, such as most functional and object-oriented languages, present the programmer with an abstract model of data representations. While such an abstraction hides the details of the special run-time representations needed to support high-level features, it comes at the cost of making interoperability with low-level languages like C non-trivial. This incompatibility poses serious challenges for both implementors and users of high-level languages, since there are numerous important libraries that have C APIs (application program interfaces). For the purposes of this paper, we view C as the prototypical low-level language. [4]

All widely used high-level language implementations provide some means for calling *foreign functions* written in C. Such a mechanism is called a *foreign-function interface* (FFI). The requirements of an FFI mechanism are to convert the arguments of a call from their high-level to their low-level representations (called *marshaling*), to handle the transfer of control from the high-level language to C and back, and then to convert the low-level representation of the results into their corresponding high-level representation (called *unmarshaling*). In addition, the FFI mechanism may map errors to high-level exceptions. The details of how data marshaling and unmarshaling are performed define a *policy* that determines how high-level code can interact with foreign functions.

One important policy question is how to treat complicated foreign data structures, such as C `structs`, arrays, and pointer data structures. While most existing FFI mechanisms handle C scalars well, they usually treat large C data structures as abstract values in the high-level language. Although this approach is often sufficient, there are situations in which the high-level language needs to have direct access to large C data structures and marshaling is infeasible. Examples include situations in which an outside authority predetermines a data format not expressible in the high-level language, such as network packet headers and RPC stub generation, or where the volume of data makes data marshaling prohibitively expensive, such as real-time graphics applications that must pass large amounts of vertex and texture data to the rendering engine. For these situations, we need a *foreign-data interface* (FDI), which is a mechanism for allowing the high-level language to manipulate C representations directly. The combination of an FFI and an FDI provides a complete solution to the interoperability problem.

This paper describes the architecture of the Moby compiler and how that architecture efficiently supports a wide range of interoperability policies, including both FFIs and FDIs. Moby is a high-level, statically-typed programming language with an ML-like module system [FR99]. Moby's support for interoperability is based on two features of its compiler. First, the compiler's intermediate representation, called BOL, is expressive enough to implement

[4] We do not consider the problems associated with interoperability between different high-level languages in this paper.

C. Hence, it is easy for interoperability tools to generate BOL code to manipulate C data structures, access C global variables, and invoke C functions. Second, the compiler can import BOL code into its compilation environment; including BOL code produced by interoperability tools.

This framework produces very efficient foreign interfaces because the compiler infrastructure supports cross-module inlining of BOL code, allowing it to tightly integrate BOL code for manipulating C structures with high-level client code. Because of inlining, the assembly code sequences generated by the MOBY compiler for accessing C data structures have very little overhead; indeed, the instruction sequences mimic those produced by a C compiler. We use the same mechanism to define MOBY's primitive types, such as integers, and the associated operations, such as addition. Hence the efficiency of this mechanism is crucial to the performance of the MOBY compiler.

Our architecture serves as the basis for implementing a number of different foreign interface policies. Each such policy must determine how C types should be packaged in MOBY. We have built tools that implement two different policies: one that marshals data to cross the boundary between C and MOBY and one that simply embeds C into MOBY. The first of these belongs to the family of interoperability tools based on an *interface description language* or IDL [FLMP98,Ler99,PR00]. An IDL provides annotations to specify function argument and result-passing conventions, as well as the semantics of C types (*e.g.*, marking a "`char *`" value as a string). The *moby-idl* tool generates a MOBY interface to C functions from an IDL specification. The second tool, called *Charon*, implements the minimal (or identity) policy. By embedding C directly into MOBY, this tool provides both a foreign function interface and a foreign data interface, in that it allows high-level code to manipulate low-level data structures in-place. It is worth noting that in our framework, a given MOBY program may use foreign APIs from both of these sources, or others like them, concomitantly.

The paper is organized as follows. In Section 2, we describe the architecture of the MOBY compiler and explain how this architecture supports data-level interoperability. We describe *moby-idl* and *Charon* in Section 3 as examples of interoperability tools built using our framework. In Section 4, we give experimental evidence showing that these tools yield very efficient foreign interfaces between C and MOBY. We discuss related work in Section 5 and conclude in Section 6.

2 The Moby compiler infrastructure

Our framework for interoperability is based on our existing compiler infrastructure. There are several aspects of this infrastructure that are key to supporting interoperability:

- The compiler's intermediate representation, called BOL, is expressive enough to describe low-level data representations and manipulations that are not

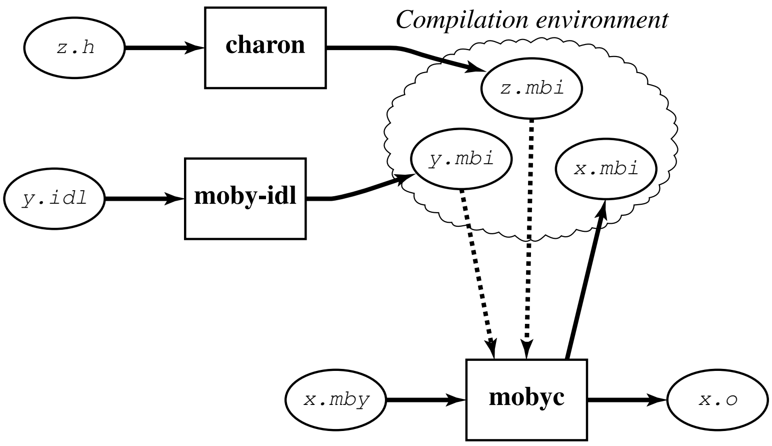

Fig. 1. The Moby compiler infrastructure

expressible in Moby itself.

- Primitive Moby types and operations are defined in terms of BOL types and functions. These definitions are given in external Moby interface files, called MBI files, and play a rôle similar to that of *native methods* in Java [Lia99] in that they allow Moby interfaces to be implemented by low-level code that cannot be written in Moby.
- The compiler can import and inline code from MBI files.
- There is a tool for generating MBI files from textual descriptions, called MBX files.

Figure 1 illustrates our compiler infrastructure when compiling a Moby source file (***x.mby***) that imports interfaces from both an IDL file (***y.idl***) and a C header file (***z.h***). We use *moby-idl* to generate ***y.mbi*** from the IDL file and *Charon* to generate ***z.mbi*** from the C header file. In this section, we discuss this infrastructure and the small number of additional features we added to support interoperability.

2.1 MBI files

In addition to an object file, the Moby compiler (**mobyc**) generates an MBI file when compiling a source file. Such a file contains information to support cross-module typechecking, analysis, and inlining. Collectively, the MBI files of an application are called its *compilation environment*. An MBI file contains the information found in a Moby signature, but it also can contain information about the implementation that is not visible in the signature, such as the representation of abstract types (as BOL types) and the implementation of functions (as BOL terms).

While most MBI files are generated by the compiler from Moby source files, the compiler can use MBI files from other sources. For example, the primitive types and operations (*e.g.*, `Int` and `+`) are specified in hand-written

MBI files,[5] which the compiler imports. We use the same mechanism to import information about foreign functions and data representations into the compiler.

2.2 BOL

The MOBY compiler uses an extended λ-calculus, called BOL, as its intermediate representation for optimization. We first describe BOL's type system and how to connect MOBY types to their underlying BOL representation. We then give an overview of BOL terms and how they support efficient interoperability.

BOL has a weak type system that does not guarantee type safety (one can think of BOL's type system as roughly equivalent to C's without the recursive types). The MOBY compiler uses BOL types to guide the mapping of BOL variables to machine registers, to provide representation information for the garbage collector (we are using the Smith-Morrisett *mostly-copying collector* [SM97]), and to provide some sanity checking for the optimizer's transformations. For example, using BOL type constructors `enum`, `ptr`, `vector`, and `struct`, we can define names for BOL types:

```
typedef char = enum(0,255)
typedef string_data = ptr(vector(1, char))
typedef string = ptr(struct 8:4 (0: int, 4: string_data))
```

A `char` is a value between 0 and 255, a `string_data` is a pointer to a vector of unknown length storing `char`s (each of which has a size of one byte), and a `string` is a pointer to a heap object consisting of an integer (the length) and a `string_data` pointer. To promote interoperability with C, the runtime representation of string data is null-terminated. We use BOL `struct` types to describe heap objects. In the example, the "`8:4`" notation on the BOL `struct` type gives its size and alignment, respectively; the "`0:`" and "`4:`" are the offsets of its fields. These definitions assume a 32-bit architecture.

The MBI file format allows an abstract MOBY type to be defined in terms of a BOL type definition. For example, the MBI file that implements the primitive MOBY types defines the MOBY `String` type in terms of the primitive BOL type `string` given above.

```
type String = prim string
```

BOL types are stratified into three levels (or kinds): types with kind **word** describe those values that can be represented by a general-purpose register on the target machine (*e.g.*, scalars and pointers); types with kind **var** describe those values that can be bound to a BOL variable, which includes the **word** types; and types with kind **memory** describe values that can be held in memory, which includes the **word** and **var** types. Target-specific aspects of the type system are captured by the kinding judgements. For example, on a 32-bit architecture the 64-bit integer type has **var** kind, whereas on a 64-bit

[5] Strictly speaking, we write an MBX file that is translated into a binary MBI file.

architecture, the 64-bit integer type has **word** kind.

Syntactically, BOL is a direct-sytle extended λ calculus.[6] It includes a primitive notion of heap object and primitive operations for manipulating raw machine data types such as addresses and 32-bit integers. The binding form

```
let obj : ty = alloc (x1, ..., xn)
  ...
```

allocates a heap object of n fields and binds *obj* to it. The memory layout of *obj* is inferred from the types of the x_i. We use the notation #*i* to select the *i*th field of such objects (zero-based indexing). BOL primitive operations include `I32Add` for adding 32-bit integers, `AdrAdd` for pointer arithmetic, `AdrEq` for comparing addresses, and `AdrLoadI32` for fetching 32-bit integers from memory.

MBI files can include definitions of Moby functions in terms of BOL terms; these definitions are used to support cross-module inlining and inlining of primitive operations. For example, the `length` function on strings is defined as follows:

```
val length : String -> Int =
      fun len (s : string, _ : exn_handler) {
        let n : int = s#0
        return n
      }
```

The first line gives the Moby name and type of the function; the remaining code is its BOL definition. The BOL function `len` (the name is to allow recursion) has two parameters. The first parameter (`s`) has BOL `string` type (which is the representation of the Moby `String` type); the second is the implicit exception-handler continuation parameter required by the Moby calling convention. Since the implementation of the `length` function does not use this parameter, we use a wildcard for it. The body of the function is trivial: we select the first component of `s` (recall that the BOL type string is a pointer to a heap object), bind it to `n`, and then return `n`.

The Moby compiler inlines such BOL implementations to reduce the overhead of calling the associated Moby functions and to enable other optimizations. For example, when compiling the Moby expression `length s + 1`, the definition of `length` will be inlined, resulting in the following intermediate BOL term:

```
let n = s#0
let t = I32Add(n, 1)
return t
```

In the case that `s` is bound to a known literal, this expression can be further reduced to a constant.

[6] Strictly speaking, BOL is a normalized representation that requires all intermediate results (including literals) to be bound to fresh variables, but our tools for compiling MBI files from their textual description accepts more complicated subexpressions and performs the normalization by introducing new temporary variables. In this paper, we use the unnormalized representation, since it is easier to read.

2.3 Interoperability extensions

To support interoperability, we have added C calls, function types, and declarations to BOL. To improve the efficiency of foreign function calls, we added a mechanism to stack allocate temporary storage. We describe these pieces in turn.

BOL has a binding form for calling C functions:

```
let result = ccall id (arg₁, ..., argₙ)
  ...
```

The identifier ***id*** is a BOL variable that may be bound to a known C function (or to a function pointer). The type of `id` controls the calling convention for the C call; we added a C-function type constructor to the BOL type system for this purpose. The types of C function parameters are mapped to BOL types in a way that preserves the specified calling convention (*e.g.*, small integer types are promoted to the BOL equivalent of `int`). The one technical complication arises from `struct` parameters. BOL's kinding system does not allow BOL variables to be bound to values whose type has **memory** kind, so we have to represent C `struct` values by their addresses. The problem then is how to distinguish between a pointer to a `struct` and a `struct` parameter? We added an additional type constructor for the latter case to solve this problem. We also added a `void` type to represent `void` return types from C functions. Our mechanism does not support *varargs* yet, but it should be possible to do so once the underlying code generator has such support. [7]

If an MBI file has a reference to a C function, it must contain an external declaration, which has the following form in the textual (MBX) representation:

```
external result-ty C-ident (param-ty₁, ..., param-tyₙ)
```

where ***result-ty***, ***param-ty***$_1$, ..., ***param-ty***$_n$ are BOL types of **var** kind.

Connecting a high-level language and C often requires temporary space for marshaling `struct` arguments and results. Efficiency considerations require a lightweight mechanism for allocating this space. Since the lifetime of such storage is the call to the C function, the most efficient place for such storage is in the stack frame of the wrapper function. To support such allocation, we added a binding form to BOL for allocating stack space. The binding form

```
stackalloc x[sz, align]
  ...
```

binds ***x*** to *sz* bytes of stack storage aligned on an *align*-byte boundary (*sz* and *align* are both integer constants). The lifetime of the storage is the scope of the binding (*i.e.*, the "`...`"), so BOL code that uses `stackalloc` must not allow references to the storage to escape. In practice, this restriction has not been a problem; furthermore, the MOBY optimizer does not perform any transformations that would expand the extent of a variable beyond its scope.

[7] Our code generator is based on the MLRisc framework [GGR94].

3 Applications

Our framework is designed to support a wide range of interoperability policies. In this section, we describe two foreign-interface generator tools for Moby that implement significantly different policies on top of our framework. We also describe some other potential uses of our framework.

3.1 Moby IDL

One of the most popular ways of automating the connection between high-level languages and C is to use an *interface description language* (IDL) to specify the foreign interface [FLMP98,Ler99,PR00]. An IDL specification is essentially a C header file with annotations. The annotations are used to specify the *direction* of parameters and the interpretation of pointer types. IDL-based generators compile these specifications into high-level interfaces and glue code. We have retargeted the SML/NJ IDL-based foreign-interface generator [PR00] to produce MBI files from an IDL specification. [8]

The *moby-idl* tool generates an MBI file that defines a Moby type for each non-trivial type in the IDL specification and a stub function for each function prototype. The stub function embodies a *copy-in/copy-out* policy for calling C functions: every input parameter is translated from its Moby representation to C and then passed to the C function, and the result and every output parameter is translated from C to its Moby representation. The stub functions have Moby types, but are implemented using BOL.

To make this discussion concrete, we give two examples, each highlighting different aspects of the *moby-idl* tool. The first example is the `getenv` function from the C library. This function takes a string argument that names an environment variable and returns the value of the variable as a string. If the named variable has no value, `NULL` is returned. We use the following IDL specification to capture this behavior:

```
typedef [unique,string] char *StringOpt;

StringOpt getenv ([in,string] char *name);
```

The `StringOpt` type is annotated as being `unique`, which means that it can be `NULL`, and it is annotated as being a `string`. The `name` parameter to the `getenv` function is marked as being an input parameter and also as a string. The *moby-idl* tool is guided by these annotations to produce the following Moby specification:

```
val getenv : String -> Option(String)
```

In the MBI file, this stub function has the following definition:

[8] There are a number of IDL variants; the SML/NJ tool (and *moby-idl*) accepts an extension of the OSF DCE dialect, which is essentially the version that Microsoft uses for COM [Gud01].

```
val getenv : String -> Option(String) =
      fun getenv (str : string, _ : exn_handler) {
        let c_str = str#1
        let c_res = ccall getenv (c_str)
        if AdrEq (c_res, nil) then return 0
        else {
          let res_str = ccall MOBY_AllocCString(c_res)
          let res = alloc (res_str)
          return (res)
        }
      }
```

This BOL code converts the input parameter `str` to a C string by selecting the second component of the heap object (recall that Moby strings are represented as a pair of a length and data-pointer) and passing it to the `getenv` function. If the result (`c_res`) is `NULL`, then `0` is returned, which is the representation of the Moby constant `None`. If the result is not `NULL`, then a Moby string is allocated and initialized from `c_res` by calling the Moby runtime system function `MOBY_AllocCString`. The Moby string is then wrapped in a one-word heap object (the representation of the `Some` data constructor) and returned.

A slightly more involved example is the following IDL specification of the interface to the Unix `gettimeofday` system call:

```
typedef struct {
    long  tv_sec;
    long  tv_usec;
} timeval;

typedef struct {
    int  tz_minuteswest;
    int  tz_dsttime;
} timezone;

int gettimeofday (
    [ref, out] timeval *t,
    [ref, out] timezone *tz);
```

This specification includes `out` annotations on the two parameters of `gettimeofday`. These `out` annotations identify the parameters as pointers to storage for returning the results of the function call. The *moby-idl* tool produces the following Moby interface from this specification:

```
datatype Timeval { TIMEVAL of (Int, Int) }
datatype Timezone { TIMEZONE of (Int, Int) }
val gettimeofday : () -> (Int, Timeval, Timezone)
```

The tool has translated the C `struct` types to Moby singleton datatypes.[9] The tool has mapped the `out` parameters to results in generating the Moby type for the function `gettimeofday`. Again, we implement the stub code that connects C and Moby using BOL code in the generated MBI file. This code

[9] A labeled record type would be preferable, but Moby does not have records yet.

has the following form:

```
val gettimeofday : () -> (Int, Timeval, Timezone) =
      fun gettimeofday (_ : exn_handler) {
        stackalloc tm[8:4], tz[8:4]
        let res = ccall gettimeofday (tm, tz)
        let tm2 = alloc(AdrLoadI32(tm), AdrLoadI32(AdrAdd(tm, 4)))
        let tz2 = alloc(AdrLoadI32(tz), AdrLoadI32(AdrAdd(tz, 4)))
        return (res, tm2, tz2)
      }
```

The code uses the BOL `stackalloc` construct to allocate temporary storage for the results in the stack. The variables `tm` and `tz` are each bound to the address of 8 bytes of stack storage (with 4-byte alignment). The extent of this storage is the scope of the variables (the rest of the function in this case). After calling the C `gettimeofday` function, the code extracts the contents of the results from the temporary storage and allocates a pair of heap objects for the results.

In our framework, uses of the `getenv` or `gettimeofday` functions can be inlined at the call site, which avoids the extra level of function call found in most IDL-based FFI mechanisms and also provides further opportunities for optimization such as avoiding unnecessary marshaling. Section 4 provides experimental results that demonstrate the efficiency of this approach.

3.2 Charon

Using the Moby compiler infrastructure, we have built a second interoperability tool, called *Charon*. This tool implements the minimal interoperability policy, in that it simply embeds C into MOBY. It maps C types into abstract MOBY types and provides MOBY functions, implemented in BOL, for manipulating C values. It provides MOBY functions, again implemented in BOL, for calling C functions.

Charon takes as input a C header file and produces two output files. The first file contains a MOBY signature describing the types and operations defined by the header file. The second file is an MBI file containing BOL code that implements the signature in the first file. Note that the signature cannot be implemented in MOBY directly because unlike BOL, MOBY does not include the low-level operations necessary to manipulate C data structures. The MOBY compiler converts MBI files into assembly code that can be linked with the compiled C code that implements the header file.

Charon factors its embedding into two parts: one generic to C and one specific to the input header file. The generic part is the *C-interface library*, which is implemented by a hand-written MBI file. The C-interface library provides several type constructors for encoding C types:

```
type LValue(t)
type CPtr(t)
type SizeOf(t)
```

`LValue(ty)` is the type of an assignable C value of type `ty`. The underlying representation of `LValue(ty)` is an address of a memory-location containing a value of type `ty` (*e.g.*, the address of a C global variable or a C heap location). `CPtr(ty)` is the type of a pointer to a value (or array of values) of type `ty`. `SizeOf(ty)` is used to type an abstract representation of the size of `ty`; we explain further below. We define a collection of *phantom* types [10] corresponding to the primitive C types (*e.g.*, `SChar` for signed characters and `SInt` for signed integers). These phantom types are used to constrain the types of generic operations on C values. For example, the C-interface library defines the following operations on C pointers:

```
val isNull : [t] CPtr(t) -> Bool
val deref  : [t] CPtr(t) -> LValue(t)
val getPtr : [t] LValue(CPtr(t)) -> CPtr(t)
val setPtr : [t] (LValue(CPtr(t)), CPtr(t)) -> ()
val malloc : [t] SizeOf(t) -> CPtr(t)
```

(the notation "`[t]`" is the binding of a type variable `t`). The C-interface library also defines specific operations for the primitive C types, such as the following operations for signed integers:

```
val getSInt : LValue(SInt) -> Int
val setSInt : (LValue(SInt), Int) -> ()
val sizeOfSInt : () -> SizeOf(SInt)
```

In addition to the generic support provided by the C-interface library, *Charon* generates an MBI file that contains types and functions specific to the input header file. For example, consider the following C declarations that describe a binary tree type and a function for creating such trees:

```
typedef struct tree {
    int         label;
    tree_ptr    left;
    tree_ptr    right;
} tree_node, *tree_ptr;
extern tree_ptr MakeTree (int depth);
```

From these declarations, *Charon* generates an MBI file that implements the following Moby interface:

```
type Struct_tree
type Def_tree_node = Struct_tree
type Def_tree_ptr = CPtr(Struct_tree)

module Stree {
  val label  : LValue(Struct_tree) -> LValue(SInt)
  val left   : LValue(Struct_tree) -> LValue(Def_tree_ptr)
  val right  : LValue(Struct_tree) -> LValue(Def_tree_ptr)
  val sizeOf : () -> SizeOf(Struct_tree)
}

val makeTree : Int -> LValue(Def_tree_ptr)
```

[10] Phantom types are types whose only purpose is to serve as arguments to type constructors [Bur90,Rep96,LM99]. The idea of using phantom types to encode C types was suggested to us by Matthias Blume [Blu01].

The type Struct_tree is a phantom type corresponding to the C struct tree type. The two C type definitions (tree_node and tree_ptr) are translated to MOBY type definitions. The struct tree type is mapped to a module of operations for accessing its fields. Note that the access functions return LValues, which can be used to assign to the fields. In addition, a sizeOf function is provided, which can be used to allocate objects of type Struct_tree. Lastly, the MakeTree function is mapped to the MOBY function makeTree.

Using the C-interface library and the generated interface, we can write MOBY code that manipulates the tree data structures and calls the MakeTree function. For example, the following MOBY function walks a tree, adding one to each label:

```
fun incLabels (t : CPtr(Struct_tree)) -> ()
{
  if isNull t
    then ()
    else {
      val t = deref t;
      setSInt(Stree.label t, getSInt(Stree.label t) + 1);
      incLabels (getPtr(Stree.left t));
      incLabels (getPtr(Stree.right t))
    }
}
```

The generated MOBY interface is implemented using BOL types and functions in the generated MBI file. For this interface, the MBI file contains the following definitions:

```
type Struct_tree = prim void
type Def_tree_node = Struct_tree
typedef def_tree_ptr = addr(data)
type Def_tree_ptr = prim def_tree_ptr
```

Notice that the phantom type Struct_tree is defined to be the BOL void type. Because BOL does not have recursive types, we map the C tree_ptr type to addr(data) (the BOL equivalent of void*). The access functions for the struct tree type are implemented as follows:

```
module Stree {
  val label : LValue(Struct_tree) -> LValue(SInt) =
        fun fld(p : lvalue, _ : exn_handler) { return p }

  val left : LValue(Struct_tree) -> LValue(Def_tree_ptr) =
        fun fld(p : lvalue, _ : exn_handler)
        { let q = AdrAdd(p, 4) return q }

  val right : LValue(Struct_tree) -> LValue(Def_tree_ptr) =
        fun fld(p : lvalue, _ : exn_handler)
        { let q = AdrAdd(p, 8) return q }

  val sizeOf : () -> SizeOf(Struct_tree) =
        fun sz(_ : exn_handler) { let n = 12 return n }
}
```

Finally, the makeTree function is implemented as a trivial wrapper around the MakeTree C function.

```
extern addr(struct_tree) MakeTree (int)
val makeTree : Int -> CPtr(Struct_tree) =
      fun makeTree (arg : int, _ : exn_handler) {
        let result : addr(struct_tree) = ccall MakeTree(arg)
        return result
      }
```

3.3 Other applications

Other approaches to generating foreign interfaces are also compatible with our framework. For example, the *C→Haskell* tool uses an interface specification file coupled with a C header file to generate a Haskell interface to a C library [Cha99]. The Haskell interface is implemented using GHC's foreign interface support. Our framework can express the same level of interoperability, so building a *C→Moby* tool should be straightforward.

A more interesting application of our framework is to support domain-specific data objects. For example, a tool for generating evaluators from atttribute-grammar specifications might use a highly-tuned tree representation that cannot be expressed directly in MOBY. Such a tool could instead generate an MBI file that defines its tree representation, along with operations on the trees, which can then be used by other code written in MOBY. The compiler's cross-module inlining mechanism can then eliminate any performance penalty for using an abstract type.

4 Experimental results

In this section, we present the results of some synthetic benchmarks that demonstrate the efficiency of interoperability in our framework. Our measurements were performed on a dual-733MHz PIII workstation running Linux, kernel version 2.2.14.

Our first benchmark tests the overhead of marshaling when using the *moby-idl* tool. For this benchmark, we measured the time taken to perform 10^7 calls of the gettimeofday system call. The interface to gettimeofday was generated from the IDL specification given as an example in Section 3.1. We compare the performance of the *moby-idl* tool with that of the Haskell *H/Direct* tool (using GHC 4.08.2) and *camlidl* (using OCAML 3.00), as well as with the direct C version (using gcc -O2, version 2.91.66). The measured user and system times for this benchmark are given in Table 1. The first three data columns give the execution time in seconds (user, system, and total). The last two columns give the ratio of the user and total times compared to the direct C version. As expected, each version of the test uses essentially the same amount of system time (about 5 seconds), but they have very different user times. Discounting the system time, the *moby-idl* version has a marshaling overhead of about 15% over the direct C version. This result compares favorably with the *camlidl* overhead of 250% and the *H/Direct* overhead of

Table 1
Measured execution times for `gettimeofday` benchmark

Language or tool	Execution time (seconds) sys	usr	tot	Relative to C usr	tot
C	5.02	1.96	6.98	1.00	1.00
moby-idl	5.18	2.25	7.43	1.15	1.06
camlidl	5.48	4.90	10.38	2.50	1.48
H/Direct	6.40	68.68	75.08	35.0	10.76

Table 2
Measured execution times for `tree` benchmark

Language or tool	Execution time (seconds) sys	usr	tot	Relative to C usr	tot
C	0.37	4.61	4.98	1.00	1.00
charon	0.29	4.36	4.65	0.94	0.93
GHC	0.41	15.10	15.51	3.28	3.11

3500%.[11]

Our second benchmark is designed to test the efficiency of direct access to C data structures from MOBY code using our framework. Each iteration of the benchmark constructs a complete binary tree of depth 16 (65535 nodes), where each node is labeled with an integer generated by calling the `rand` function from the C-library. After constructing the tree, we perform a depth-first traversal to find the largest label, and then explicitly free the tree. The benchmark iterates these three steps 100 times. We wrote the MOBY version of the benchmark entirely in MOBY using the *Charon*-generated interface to the C representation. The tree data structure was managed using the MOBY C-interface library's interface to `malloc` and `free`. We compare the performance of the MOBY program with the native C version of the same algorithm; the results are given in Table 2. From these results, we can see that the data-level interoperability supported by *Charon* has no overhead over native C code.[12] We also measured a version of the program compiled under GHC. In this version, we used GHC's `Addr` type and strictness annotations to implement an interface similar to the one provided by *Charon*. These

[11] We believe that the high execution time of the *H/Direct* version is caused by its use of `malloc` and `free` to manage the temporary storage needed for the results of the `gettimeofday` call.

[12] The slight performance advantage that MOBY has in this benchmark is most likely a result of a slightly more efficient argument passing mechanism.

measurements demonstrate that while GHC's foreign-data support has much of the expressiveness of our framework, it lags in efficiency.

5 Related work

Our approach to interoperability is based on the MOBY compiler infrastructure. This infrastructure serves as the foundation for a wide range of interoperability policies, each with its own user-level mechanism. This approach contrasts with most of the prior work on language interoperability, which fixes a particular interoperability policy and user-level mechanism. [13]

Most high-level language implementations provide some mechanism for connecting with C code. Often, this mechanism requires hand-written stub functions to translate between the high-level and C representations. Examples of languages with such mechanisms include JAVA (the JAVA Native Interface) [Lia99], SML/NJ, and OCAML [Ler00]. Our framework also supports hand-written stubs. Such stubs can either be written in C, as we have done with some low-level, run-time system functions, or in an MBX file. In the latter case, the MOBY compiler's cross-module inlining mechanism allows the stub code to be inlined at its call sites.

Some systems make it possible to write interoperability code in the high-level language. For example, both the SML'97 Basis Library [GR01] and the Glasgow HASKELL compiler (GHC) [GHC01] provide operations for reading and writing scalar values in a *bytearray*. With this mechanism, one can manipulate a C data structure by importing it into a program as a bytearray. However, the user is responsible for understanding the layout of the data structure. The GHC mechanism is lower-level than that of the SML Basis; specifically, GHC does no bounds checking and it is possible to read and write pointer values.

In many respects, GHC's foreign interface support is the closest to providing the flexibility of our framework. Our mechanism is slightly more expressive in that we support C functions with `struct` arguments. As demonstrated in Section 4, we also have a significant performance advantage. For more complicated policies, such as those required by IDL, the BOL `stackalloc` construct can greatly reduce the overhead of data marshaling. It would difficult to put such a mechanism into a high-level language without using a sophisticated type system to control the extent of stack-allocated variables.

6 Conclusion

In this paper, we described MOBY's interoperability infrastructure, which consists of an expressive intermediate representation called BOL and the ability to

[13] Of course, some systems have multiple mechanisms, each of which supports a different policy.

import externally-defined BOL code into the compilation environment. This infrastructure supports a wide-range of interoperability tools, from IDL-based tools that marshal their data to tools such as *Charon* that provide data-level interoperability by embedding C into MOBY. This infrastructure is highly efficient because it supports cross-module inlining of BOL code. Experimental evaluation showed that the overhead associated with invoking foreign functions and manipulating C data structures was very low. While supporting data-level interoperability requires compiler support, our approach is not specific to the MOBY language *per se*. Specifically, we expect that our approach is compatible with compilers that have more strongly-typed IRs.

References

[Blu01] Blume, M. No-longer-foreign: Teaching an ML compiler to speak C "natively". In *BABEL'01*, September 2001. *To appear.*

[Bur90] Burton, F. Type extension through polymorphism. *TOPLAS*, **12**(1), January 1990, pp. 135–138.

[Cha99] Chakravarty, M. M. T. C → Haskell, or yet another interfacing tool. In *IFL'99*, vol. 1868 of *LNCS*. Springer-Verlag, September 1999.

[FLMP98] Finne, S., D. Leijen, E. Meijer, and S. Peyton Jones. H/Direct: A binary foreign language interface for Haskell. In *ICFP'98*, September 1998, pp. 153–162.

[FR99] Fisher, K. and J. Reppy. The design of a class mechanism for Moby. In *PLDI'99*, May 1999, pp. 37–49.

[GGR94] George, L., F. Guillame, and J. H. Reppy. A portable and optimizing back end for the SML/NJ compiler. In *CC'94*, April 1994, pp. 83–97.

[GHC01] *The Glasgow Haskell Compiler User's Guide*, version 5.00 edition, January 2001.

[GR01] Gansner, E. R. and J. H. Reppy (eds.). *The Standard ML Basis Library.* Cambridge University Press, Cambridge, England, 2001. *To appear.*

[Gud01] Gudgin, M. *Essential IDL.* Addison-Wesley, Reading, MA, 2001.

[Ler99] Leroy, X. *CamlIDL user's manual*, March 1999. Available from `http://caml.inria.fr/camlidl/`.

[Ler00] Leroy, X. *The Objective Caml System (release 3.00)*, April 2000. Available from `http://caml.inria.fr`.

[Lia99] Liang, S. *The Java Native Interface.* Addison-Wesley, Reading, MA, 1999.

[LM99] Leijen, D. and E. Meijer. Domain specific embedded compilers. In *Proceedings of the Second Conference on Domain-Specific Languages (DSL'99)*, 1999, pp. 109–122.

[PR00] Pucella, R. and J. H. Reppy. An abstract IDL mapping for Standard ML. 2000.

[Rep96] Reppy, J. H. A safe interface to sockets. *Technical memorandum*, AT&T Bell Laboratories, 1996.

[SM97] Smith, F. and G. Morrisett. Mostly-copying collection: A viable alternative to conservative mark-sweep. *Technical Report TR97-1644*, Department of Computer Science, Cornell University, August 1997.