Data-level interoperability

Kathleen Fisher
AT&T Labs, Research
kfisher@research.att.com

Riccardo Pucella
Cornell University
riccardo@cs.cornell.edu

John Reppy
Bell Labs, Lucent Technologies
jhr@research.bell-labs.com

Draft of April 14, 2000 — 16 : 36

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). Our approach to supporting foreign interfaces in the MOBY compiler is based on a mechanism for *data-level interoperability*, which allows MOBY code to manipulate C data representations directly. Data-level interoperability is important when dealing with large external data sets or data that is in a fixed format. It also serves as the foundation for a wide range of different foreign-interface policies. We describe tools that implement three such policies: *Charon*, which embeds C types directly into MOBY, *moby-idl*, which provides an IDL-based embedding, and an API miner, which enables application-specific embeddings. The benefits of our approach stem from the design of our compiler and do not rely on properties of the MOBY language.

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 special run-time representations needed to support high-level features, it comes at the cost of making interoperability with low-level languages, such as 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.

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 a FFI mechanism are to convert the arguments of the call from their high-level to their low-level representations (called *marshalling*), 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 *unmarshalling*). In addition, the FFI mechanism may map errors to high-level exceptions. The details of how data marshalling and unmarshalling are performed define a *policy* that determines how foreign functions are presented to high-level client code.

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 because marshalling is infeasible. An important case is where the sheer volume of data that must be communicated across the foreign interface makes data marshalling prohibitively expensive. For example, a real-time graphics application must pass large amounts of vertex and texture data to the rendering engine. A second example involves processing the gigabytes of call-record data that a telecommunications system collects each day [BFRS99], where the data-format is a stream of C **`struct`**s. Another situation is where an outside authority predetermines a data format, as in the case of network packet headers and RPC stub generation. Such data formats are often specified as C types and usually cannot be expressed using the high-level language's type system. 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 a FFI and FDI provides a complete solution to the interoperability problem.

This paper describes the foreign interface mechanism that we have designed and implemented for the MOBY programming language. 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 the compiler. First, the compiler's intermediate representation, called BOL, supports direct access to C data representations, global variables, and functions. And second, the compiler is able to import externally defined BOL functions. Because BOL's data representations include those of C, it is easy to generate BOL code that manipulates C data structures from a description of those structures. Using this low-level mechanism, we can take a C header file and generate a MOBY interface to the C API specified in the header file. We implement this interface in BOL because the implementation is not expressible in MOBY.

This mechanism for mapping C representations to BOL code that can manipulate them is the foundation for implementing a number of different foreign interface policies. Each policy determines how the underlying C types should be packaged in MOBY. The minimal (or identity) policy simply embeds C directly into MOBY; we have implemented this policy in a tool called *Charon*. We built another tool, called *moby-idl*, that generates a foreign interface from a specification written in an *interface description language* (IDL). The 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). We are also developing a refinement of *Charon* that supports application-specific policies. This tool allows both direct access to C representations and data marshalling in the same API. It is worth noting that in our framework a given MOBY program may use foreign APIs from all of these sources.

In summary, our approach relies on a low-level mechanism for data interoperability, upon which we build tools that support various policies for mapping from low-level APIs to MOBY. While supporting data-level interoperability requires compiler support, our approach is not specific to the MOBY language *per se*.

The paper is organized as follows. In Section 2, we describe the aspects of the MOBY implementation that provide data-level interoperability. In Section 3, we formalize the translation from C declarations to BOL and describe *Charon*, which implements this translation. We describe tools that implement higher-level foreign interface policies in Section 4. These tools target the data-interoperability mechanism provided by BOL. We discuss related work in Section 5 and conclude in Section 6.

2 Compiler support for data-level interoperability

The MOBY compiler provides support for data-level interoperability by the combination of an intermediate representation that can express C-style data manipulations and a mechanism for importing functions defined in terms of its IR. These functions play a rôle similar to that of *native methods* in JAVA [Lia99] in that they allow MOBY interfaces to be implemented by code that is not possible to write in MOBY. In this section, we survey the features of the MOBY compiler that support interoperability. The reader should note that while our mechanism is specific to our particular compiler, it is applicable to any high-level language.

2.1 MBI files

The MOBY compiler (**mobyc**) compiles MOBY source files into object files. The compiler also generates a MOBY interface file (called an MBI file), which 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 and the implementation of functions.

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,[1] which the compiler imports. We also use this mechanism to import information about foreign functions and data representations into the compiler. Figure 1 illustrates this process in the compilation of a program, which is using both the *moby-idl* and *Charon* tools to define foreign interfaces.

2.2 BOL

The MOBY compiler uses an extended λ-calculus, called BOL, as its intermediate representation for optimization. One of the more important optimizations is *inlining*, which is supported across module boundaries by including the BOL representation of functions that are good candidates for inlining in the MBI files. We rely on the cross-module inlining mechanism to inline primitive operations (*e.g.*, integer addition). The same mechanism works on the BOL code generated by our foreign interface tools.

[1] Strictly speaking, we write a textual description that is translated into a binary MBI file.

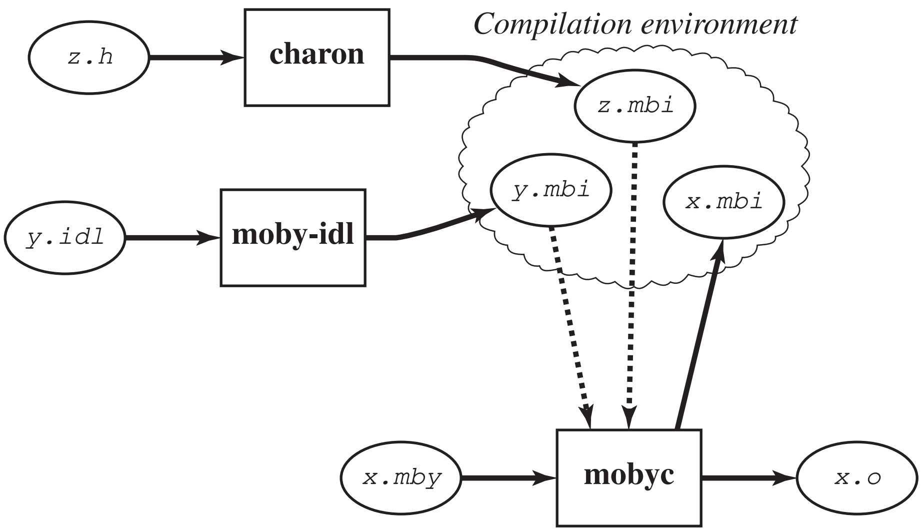

Figure 1: The MOBY compiler structure

In this paper, we use an ML-like syntax for BOL code that is designed for readability. Table 1 describes the BOL operations that we use to support C data representations in the examples. The BOL form also supports calls to external C functions.

2.3 BOL types

The BOL IR has an associated type system. Unlike many recent typed IRs, this type system does not attempt to soundly type any BOL term.[2] Rather, it is a *representation* description that the compiler uses 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 basic sanity checking of the optimizer's transformations. The MBI file format allows an abstract MOBY type to be defined in terms of a BOL type definition; we use this feature to define the primitive types (*e.g.*, `Int`) and to define the abstract MOBY types that denote C data representations.

For the purposes of this paper, we are interested in the subset of BOL types that covers the data-layout issues presented by C. BOL types are stratified into two layers: *simple types* (denoted τ) and *heap types* (denoted θ). Simple types describe those values that can be named by a BOL variable (*i.e.*, scalar values and pointers), while heap types describe the layout of data in memory

[2] Although BOL is not strongly typed, we believe that our mechanism for direct access to C representations is largely compatible with typed IRs, such as TAL [MWCG98, MCG^{+}99].

Table 1: Some BOL primitive operations

Operation	**Description**
`AdrAdd` (a, n)	add the integer n to the address a
`AdrSub` (a, n)	subtract the integer n from the address a
`AdrLoadU8` (a)	load an unsigned 8-bit integer from the address a
`AdrLoadI32` (a)	load a signed 32-bit integer from the address a
`AdrLoadP` (a)	load a pointer from the address a
`AdrStoreI8` (a, n)	store the 8-bit integer n at the address a
`AdrStoreI32` (a, n)	store the 32-bit integer n at the address a
`AdrStoreF32` (a, f)	store the 32-bit floating-point value f at the address a
`AdrStoreP` (a, p)	store the pointer p at the address a
`I32Mul` (n, m)	multiply the integers n and m
`U32Lt` (n, m)	return **`True`** if the unsigned integer n is less than m

and include the simple types. The syntax of BOL types is as follows:

$$\begin{array}{rcll} \tau & ::= & \texttt{unit} \mid \texttt{int8} \mid \texttt{int16} \mid \texttt{int32} \mid \cdots & \textit{scalar types} \\ & \mid & \texttt{ptr}(\theta) & \textit{pointer type} \\ \\ \theta & ::= & \tau & \\ & \mid & \langle p_1 \rhd \theta_1^{k_1}, \ldots, p_n \rhd \theta_n^{k_n} \rangle_{sz} & \textit{struct type} \end{array}$$

Objects are a restricted form of C **`struct`** type; the type

$$\langle p_1 \rhd \theta_1^{k_1}, \ldots, p_n \rhd \theta_n^{k_n} \rangle_{sz}$$

is a structure of size sz with n fields. The ith field is located at offset p_i and has k_i elements of type θ_i. For example, on an Intel IA32 processor the C type

```
struct { char c[3]; int i; };
```

maps to the BOL heap type $\langle 0 \rhd \texttt{int8}^3, 4 \rhd \texttt{int32}^1 \rangle_8$. Note that there is an implicit byte of padding between the first and second fields in this representation, which is required to satisfy the ABI (*Application Binary Interface*) alignment constraints.

3 Embedding C representations in MOBY

In this section, we describe the interoperability mechanism that allows MOBY code to create, access, and modify C data representations and to call C functions. We define our mechanism via a collection of type-directed translation functions from C declarations to special MOBY modules. These modules have normal MOBY signatures, but BOL-level implementations because the operations necessary to manipulate C data structures cannot be expressed directly in MOBY. We represent such modules as MBI files in the MOBY compilation environment. In more detail, the target of the

Table 2: Summary of the translations

$\mathcal{N}_{\mathcal{T}}[\![Ct]\!]$	translates the C type Ct to a MOBY type name.
$\mathcal{N}_{\mathcal{M}}[\![Ct]\!]$	translates the C type Ct to a MOBY module name.
$\mathcal{D}_{\mathcal{T}}[\![Ct]\!]$	translates the C type Ct to a MOBY type specification.
$\mathcal{D}_{\mathcal{M}}[\![Ct]\!]$	translates the C type Ct to a MOBY module specification.
$\mathcal{D}_{\mathcal{V}}[\![Cd]\!]$	translates the C declaration Cd to a MOBY value specification.
$\mathcal{I}[\![Ct]\!]$	translates the C type Ct to specifications of its data-manipulation operations.
$\mathcal{I}_{memb}[\![Cm]\!]\ T$	translates the C **`struct`** member Cm to specifications of its data-manipulation operations, where T is the MOBY type name of the containing C **`struct`**.
$\mathcal{A}[\![Ct]\!]$	translates the C type Ct to a BOL simple type.
$\mathcal{T}[\![Ct]\!]$	translates the C type Ct to a BOL heap type.
$\mathcal{S}[\![\vec{Cm}]\!]$	translates the C **`struct`** member list $\vec{Cm}$ to a BOL heap type and member offset map.
$\mathcal{S}_{memb}[\![Ct]\!]\ p$	translates the C **`struct`** member type Ct at offset p to an offset, BOL heap type, and alignment.
$\mathcal{E}[\![Ct]\!]$	translates the C type Ct to the BOL implementation of its data-manipulation operations.
$\mathcal{E}_{memb}[\![Cm]\!]\ \phi$	translates the C **`struct`** member Cm to the BOL implementation of its data-manipulation operations, where ϕ is the member offset map of the containing C **`struct`**.

translation of a C header file is three-fold: a MOBY module name, a MOBY signature for that module, and a BOL implementation for that signature. The MOBY signature declares abstract MOBY types that correspond to the types in the header file, submodules that contain the data-manipulation operations necessary to support these types, and MOBY values that correspond to the external variable and function declarations. The implementation contains BOL bindings for the abstract types, BOL implementations for the necessary data-manipulation operations, and BOL terms to access external variables and functions.

We summarize the various translation functions that collectively map C header files to such MOBY modules in Table 2. Figure 2 illustrates these functions and their relationships. Functions $\mathcal{N}_{\mathcal{T}}$ and $\mathcal{N}_{\mathcal{M}}$ map C types to MOBY type and module names, respectively; we omit their definitions as they use standard name-mangling techniques. We have built a tool, called *Charon*, that implements these translations. In the remainder of this section, we describe the various translation functions and discuss *Charon* in more detail.

3.1 C types and declarations

Before defining the translation functions, we must specify a grammar for the C types and declarations that we will translate. To simplify the presentation, we omit from this grammar C enumerations, unions, bit-fields, function pointers, and all but the base types `int` and `void`. We also emit functions that return **`struct`** results. The tool *Charon* supports enumerations, unions, and the full

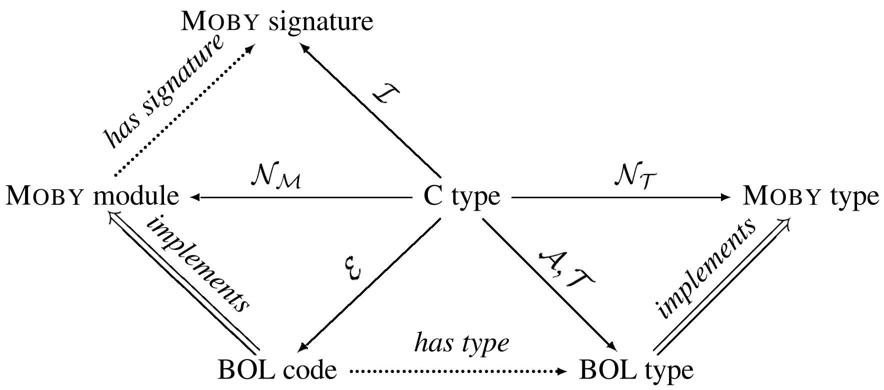

Figure 2: Translations of C types

range of C scalar types. It does not support bitfields, function pointers, **struct** return types, nor functions with variable-length argument lists.

We assume that a pre-processor has normalized the types, removing nested **struct** types, naming anonymous **struct** types, and expanding type definitions. The normalized C declarations and types have the following syntax:

$$
\begin{array}{rcll}
Cd & ::= & \mathbf{extern}\ Ct\ x; & \\
 & | & \mathbf{extern}\ Ct\ f(\vec{Ct}); & \text{where } Ct \neq \mathbf{struct}\ t \\
 & | & \mathbf{struct}\ t\ \{\vec{Cm}\}; & \\
Cm & ::= & Ct\ m; & \\
Ct & ::= & \texttt{int} \mid \texttt{void} & \\
 & | & Ct\,* & \\
 & | & Ct\,[\,n\,] & \\
 & | & \mathbf{struct}\ t &
\end{array}
$$

Declarations include global variables, function prototypes, and C **struct** definitions. One subtlety about the normalization process is the treatment of static-sized multi-dimensional arrays. Because C uses a row-major order, it is necessary to reverse the order of subscripts. For example, the type declaration

```
struct t { int a[3][4]; };
```

is normalized to

```
struct t { int[4][3] a; };
```

In the remainder of this section, we use SCALAR to refer to the set of scalar types (*i.e.*, `int` and `void`), and ATOMIC to refer to the set of *atomic types*, which we define to be SCALAR unioned with the pointer types (*i.e.*, $Ct\,*$).

3.2 Translating to MOBY signatures

In this section, we describe the translation functions that generate the MOBY signature for the target module. A C header file consists of a sequence of declarations ($\vec{Cd}$); let $\mathcal{C}$ be the set of all non-scalar types mentioned in $\vec{Cd}$.[3] The first step of our translation is to compute a global environment Γ that maps the names of C structs to their vector of fields. Formally,

$$\Gamma = \{t \mapsto \vec{Cm} \mid \texttt{struct}\ t\ \{\vec{Cm}\}; \in \vec{Cd}\}$$

We then precompute the *closure* of $\mathcal{C}$, which is the set of types that the generated MOBY interface will support. We define the closure as follows:

$$\begin{aligned}\hat{\mathcal{C}} = \mathcal{C} & \cup \{ Ct * \mid Ct\,[\,n\,] \in \mathcal{C}\} \\ & \cup \{\texttt{struct}\ t * \mid \texttt{struct}\ t \in \mathcal{C}\} \\ & \cup \{ Ct * \mid Ct\ m \in \text{rng}(\Gamma)\}\end{aligned}$$

For example, consider a header file consisting of the declaration of **struct** t given above. The non-scalar types mentioned in that declaration are

$$\mathcal{C} = \{\texttt{struct}\ t, \texttt{int}\,[4], \texttt{int}\,[4]\,[3]\}$$

and the closure is

$$\hat{\mathcal{C}} = \mathcal{C} \cup \{\texttt{struct}\ t*, \texttt{int}*, \texttt{int}\,[4]*\}$$

Given these definitions, we can define the interface translation of $\vec{Cd}$. We construct the target interface by first applying two translation functions ($\mathcal{D}_\mathcal{T}$ and $\mathcal{D}_\mathcal{M}$) to each $Ct \in \hat{\mathcal{C}}$, then applying a third translation function ($\mathcal{D}_\mathcal{V}$) to each external declaration in the header file, and finally collecting the resulting MOBY declarations into a signature. The translation function $\mathcal{D}_\mathcal{T}$ maps a C type to the corresponding MOBY abstract type declaration. Translation $\mathcal{D}_\mathcal{M}$ constructs the submodule signature that specifies the operations to manipulate values of type Ct. And translation $\mathcal{D}_\mathcal{V}$ generates the specifications of the operations to access external variable and function definitions. These translations are detailed in Figure 3.

We apply the function $\mathcal{D}_\mathcal{T}$ to each type $Ct \in \hat{\mathcal{C}}$. Because $\hat{\mathcal{C}}$ does not contain scalar types, this definition is exhaustive. We embed non-pointer types into MOBY as abstract types, while we make pointer types partially abstract, revealing that they are subtypes of the MOBY type for `void*`.

We also apply $\mathcal{D}_\mathcal{M}$ to each type $Ct \in \hat{\mathcal{C}}$, which gives the sub-module that defines the operations on the abstract MOBY type that corresponds to Ct. The definition of $\mathcal{D}_\mathcal{M}$ involves the translation $\mathcal{I}$, which is also given in Figure 3. This translation maps a type Ct to the specifications that comprise the interface of the submodule $\mathcal{N}_\mathcal{M}[\![Ct]\!]$. There are three kinds of operations in these interfaces: *get operations* that extract a value, *set operations* that store a value, and *address operations* that return a value's address. One should think of the get operations as corresponding to the C dereferencing operator (`*`), the set operations as corresponding to assignment (`=`), and the address operations as corresponding to the address-of operator (`&`). The semantics of the `adr` operation depends on the

[3] We exclude the scalar types because they have direct counterparts in MOBY; *e.g.*, the C type `int` is equivalent to the MOBY type `Int`.

$$
\begin{array}{rcll}
\mathcal{D}_{\mathcal{T}}[\![Ct]\!] & = & \texttt{type}\ \mathcal{N}_{\mathcal{T}}[\![Ct]\!] & Ct \notin \textsc{Atomic} \\
\mathcal{D}_{\mathcal{T}}[\![Ct\,*]\!] & = & \texttt{type}\ \mathcal{N}_{\mathcal{T}}[\![Ct\,*]\!]\ \texttt{<:}\ \mathcal{N}_{\mathcal{T}}[\![\texttt{void}*]\!] &
\end{array}
$$

$$
\mathcal{D}_{\mathcal{M}}[\![Ct]\!] \;=\; \texttt{module}\ \mathcal{N}_{\mathcal{M}}[\![Ct]\!]\ \texttt{:}\ \{\ \mathcal{I}[\![Ct]\!]\ \}
$$

$$
\begin{array}{rcl}
\mathcal{D}_{\mathcal{V}}[\![\texttt{extern}\ Ct\ x;]\!] & = & \texttt{val}\ \texttt{global\_x}\ \texttt{:}\ \texttt{Unit} \rightarrow \mathcal{N}_{\mathcal{T}}[\![Ct\,*]\!] \\
\mathcal{D}_{\mathcal{V}}[\![\texttt{extern}\ Ct\ f(Ct_1,\ \ldots\ ,\ Ct_n);]\!] & = & \texttt{val}\ \texttt{fun\_f}\ \texttt{:}\ (\mathcal{N}_{\mathcal{T}}[\![Ct_1]\!],\ \ldots\ ,\ \mathcal{N}_{\mathcal{T}}[\![Ct_n]\!]) \rightarrow \mathcal{N}_{\mathcal{T}}[\![Ct]\!] \\
\mathcal{D}_{\mathcal{V}}[\![\texttt{struct}\ t\ \{\vec{Cm}\};]\!] & = & \emptyset
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{I}[\![Ct\,*]\!] & = & \left\{ \begin{array}{lcl}
\texttt{val get} & : & \mathcal{N}_{\mathcal{T}}[\![Ct\,*]\!] \rightarrow \mathcal{N}_{\mathcal{T}}[\![Ct]\!] \\
\texttt{val set} & : & (\mathcal{N}_{\mathcal{T}}[\![Ct\,*]\!],\ \mathcal{N}_{\mathcal{T}}[\![Ct]\!]) \rightarrow \texttt{Unit} \\
\texttt{val adr} & : & (\mathcal{N}_{\mathcal{T}}[\![Ct\,*]\!],\ \texttt{Int}) \rightarrow \mathcal{N}_{\mathcal{T}}[\![Ct\,*]\!]
\end{array} \right\} \\[2ex]
\mathcal{I}[\![Ct\,[\,n\,]]\!] & = & \left\{ \begin{array}{lcl}
\texttt{val get} & : & (\mathcal{N}_{\mathcal{T}}[\![Ct\,[\,n\,]]\!],\ \texttt{Int}) \rightarrow \mathcal{N}_{\mathcal{T}}[\![Ct]\!] \\
\texttt{val set} & : & (\mathcal{N}_{\mathcal{T}}[\![Ct\,[\,n\,]]\!],\ \texttt{Int},\ \mathcal{N}_{\mathcal{T}}[\![Ct]\!]) \rightarrow \texttt{Unit} \\
\texttt{val adr} & : & (\mathcal{N}_{\mathcal{T}}[\![Ct\,[\,n\,]]\!],\ \texttt{Int}) \rightarrow \mathcal{N}_{\mathcal{T}}[\![Ct\,*]\!]
\end{array} \right\} \\[2ex]
\mathcal{I}[\![\texttt{struct}\ t]\!] & = & \left\{\ \mathcal{I}_{memb}[\![Cm]\!]\ (\mathcal{N}_{\mathcal{T}}[\![\texttt{struct}\ t]\!])\ \mid\ Cm \in \Gamma(t)\ \right\} \\[2ex]
\mathcal{I}_{memb}[\![Ct\ m;]\!]\ T & = & \left\{ \begin{array}{lcl}
\texttt{val get\_m} & : & T \rightarrow \mathcal{N}_{\mathcal{T}}[\![Ct]\!] \\
\texttt{val set\_m} & : & (T,\ \mathcal{N}_{\mathcal{T}}[\![Ct]\!]) \rightarrow \texttt{Unit} \\
\texttt{val adr\_m} & : & T \rightarrow \mathcal{N}_{\mathcal{T}}[\![Ct\,*]\!]
\end{array} \right\} \text{when } Ct \in \textsc{Atomic} \\[2ex]
\mathcal{I}_{memb}[\![Ct\,[\,n\,]\ m;]\!]\ T & = & \left\{ \begin{array}{lcl}
\texttt{val get\_m} & : & (T,\ \texttt{Int}) \rightarrow \mathcal{N}_{\mathcal{T}}[\![Ct]\!] \\
\texttt{val set\_m} & : & (T,\ \texttt{Int},\ \mathcal{N}_{\mathcal{T}}[\![Ct]\!]) \rightarrow \texttt{Unit} \\
\texttt{val adr\_m} & : & (T,\ \texttt{Int}) \rightarrow \mathcal{N}_{\mathcal{T}}[\![Ct\,*]\!]
\end{array} \right\} \\[2ex]
\mathcal{I}_{memb}[\![\texttt{struct}\ t\ m;]\!]\ T & = & \left\{ \begin{array}{lcl}
\texttt{val get\_m} & : & T \rightarrow \mathcal{N}_{\mathcal{T}}[\![\texttt{struct}\ t]\!] \\
\texttt{val set\_m} & : & (T,\ \mathcal{N}_{\mathcal{T}}[\![\texttt{struct}\ t]\!]) \rightarrow \texttt{Unit} \\
\texttt{val adr\_m} & : & T \rightarrow \mathcal{N}_{\mathcal{T}}[\![\texttt{struct}\ t\,*]\!]
\end{array} \right\}
\end{array}
$$

Figure 3: The interface translations

form of Ct: for pointer types, it supports C-style pointer arithmetic; for array types, it returns the address of an element; and for **struct** types, it returns the address of a component. In addition to these operations, we synthesize conversions from the universal address type (`void*`) to the pointer types and include size information for **struct** and array types. We omit the formalization of these pieces to save space, but include them in an example below.

The last interface translation is $\mathcal{D}_{\mathcal{V}}$, which is applied to the declarations in $\vec{Cd}$ to generate MOBY specifications. Its definition is given in Figure 3. This translation maps a global variable x to a function that returns x's address. C external variables correspond to pointers in MOBY because C variables are inherently imperative. Function $\mathcal{D}_{\mathcal{V}}$ maps each function prototype to the corresponding MOBY function type. It ignores **struct** declarations because the $\mathcal{D}_{\mathcal{T}}$ and $\mathcal{D}_{\mathcal{M}}$ functions have introduced all the necessary specifications already.

3.3 Translating to an implementation

In this section, we describe the translations that generate the implementation of the target module. As noted before, we use BOL types and terms to implement the module because the operations to manipulate C data structures are not expressible directly in MOBY. The implementation translations are presented in the same order as the interface translations: first the types, then the data manipulation operations, and finally the external variables and functions.

The translation to the implementation is necessarily dependent on the target ABI. We isolate that dependency in two auxiliary functions. The Sz function returns the size in bytes of a C type and the Align function returns the type's alignment requirement in bytes. We define the alignment of an offset p by alignment a to be

$$\lceil p \rceil_a = \min\{p' \mid p \leq p' \text{ and } p' \bmod a = 0\}$$

The translations $\mathcal{T}$ and $\mathcal{S}$, which we give in Figure 4, define the in-memory representation of C types. The $\mathcal{T}$ translation maps a C type Ct to the BOL heap type that describes Ct's memory representation. An important piece of this translation is the heap layout of C **struct**s, which the $\mathcal{S}$ translation defines. The $\mathcal{S}$ translation also defines a *member map* ϕ that associates byte offsets with the field names of the **struct**. We use this map in the definition of the $\mathcal{E}$ translation function for defining **struct** member manipulation operations.

For each type $Ct \in \hat{\mathcal{C}}$, we implement the corresponding abstract MOBY type using the BOL type $\mathcal{A}[\![Ct]\!]$. We write this binding in an MBI file as:

$$\textbf{type}\ \mathcal{N}_{\mathcal{T}}[\![Ct]\!]\ \textbf{=}\ \textbf{prim}\ \mathcal{A}[\![Ct]\!]$$

The definition of the $\mathcal{A}$ translation is similar to that of the $\mathcal{T}$ translation and is also given in Figure 4. Note that the $\mathcal{A}$ translation ranges over simple BOL types, whereas $\mathcal{T}$ ranges over heap types.

The $\mathcal{E}$ translation defines the implementations for the data manipulation operations specified by the $\mathcal{I}$ translation (see Figure 3). Because its complete definition is rather tedious, we give only a subset of the cases in Figure 5. The result of the translation is a sanitised form of BOL code (the actual BOL IR is a normalized representation in which all intermediate results are bound to variables). In this code, the functions like `AdrAdd` and `I32Mul` are the BOL primitive operations

$$
\begin{array}{rcll}
\mathcal{T}[\![\texttt{int}]\!] & = & \texttt{int32} & \\
\mathcal{T}[\![Ct\,*]\!] & = & \texttt{ptr}(\theta) & \text{where } \mathcal{T}[\![Ct]\!] = \theta \\
\mathcal{T}[\![Ct\,[\,n\,]]\!] & = & \langle 0 \rhd \theta^n \rangle_{sz} & \text{where } \mathcal{T}[\![Ct]\!] = \theta \text{ and } sz = n \times \mathrm{Sz}(Ct) \\
\mathcal{T}[\![\textbf{struct}\ t\,*]\!] & = & \texttt{ptr(any)} & \text{where } m \notin \mathrm{dom}(\Gamma) \\
\mathcal{T}[\![\textbf{struct}\ t]\!] & = & \theta & \text{where } \mathcal{S}[\![\Gamma(m)]\!] = (\theta, \phi)
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{S}[\![Ct_1\ m_1;\ \ldots\ Ct_n\ m_n;]\!] & = & (\langle p_1 \rhd {\theta_1}^1,\ \ldots\ ,\ p_n \rhd {\theta_n}^1 \rangle_{sz}, \phi) \\
& & \text{where} \\
& & \begin{array}{rcl}
(p_1, \theta_1, a_1) & = & \mathcal{S}_{memb}[\![Ct_1]\!]\ 0 \\
(p_2, \theta_2, a_2) & = & \mathcal{S}_{memb}[\![Ct_2]\!]\ (p_1 + \mathrm{Sz}(Ct_1)) \\
& \vdots & \\
(p_n, \theta_n, a_n) & = & \mathcal{S}_{memb}[\![Ct_n]\!]\ (p_{n-1} + \mathrm{Sz}(Ct_{n-1})) \\
a & = & \max(a_1,\ \ldots\ ,\ a_n) \\
sz & = & \lceil p_n + \mathrm{Sz}(Ct_n) \rceil_a \\
\phi & = & \{ m_1 \mapsto p_1,\ \ldots\ ,\ m_n \mapsto p_n \}
\end{array}
\end{array}
$$

$$
\mathcal{S}_{memb}[\![Ct]\!]\ p \quad = \quad (\lceil p \rceil_a, \theta, a) \qquad \text{where } \mathcal{T}[\![Ct]\!] = \theta \text{ and } \mathrm{Align}(\theta) = a
$$

$$
\begin{array}{rcll}
\mathcal{A}[\![Ct\,*]\!] & = & \texttt{ptr}(\theta) & \text{where } \mathcal{T}[\![Ct]\!] = \theta \\
\mathcal{A}[\![Ct\,[\,n\,]]\!] & = & \texttt{ptr}(\langle 0 \rhd \theta^n \rangle_{sz}) & \text{where } \mathcal{T}[\![Ct]\!] = \theta \text{ and } sz = n \times \mathrm{Sz}(Ct) \\
\mathcal{A}[\![\textbf{struct}\ t]\!] & = & \texttt{ptr}(\theta) & \text{where } \Gamma(m) = (\theta, \phi)
\end{array}
$$

Figure 4: Translation of C types to BOL types

(see Table 1). Notice that each function has an additional argument named h; this argument is the exception handler continuation and is part of the MOBY calling convention. The implementation of the `set` operation for non-atomic values requires data copying, which we implement using the C `memcpy` function (see Figure 5 for an example).

External variable declarations are translated into a function that returns the address of the variable. The translation of function prototype declarations produces two entries in the generated MBI file: a foreign-function declaration that gives the argument and return types of the function, and a simple function that contains a call to the foreign function. Our machine code generator, which is based on MLRISC [GGR94], handles the machine-specific mechanics of the C-calling conventions, including **`struct`** parameters.

3.4 Charon

We have built a tool, called *Charon*, that embodies the translations we have described in this section. We implemented *Charon* using SML/NJ's *CKit* library to parse and typecheck C header files.[4] The

[4]The *CKit* is available from `http://cm.bell-labs.com/cm/cs/what/smlnj/doc/ckit/`.

$$\mathcal{E}[\![\texttt{int}\,*]\!] = \left\{\begin{array}{l} \textbf{fun}\ \texttt{get}\ (p,h)\ \{\,\texttt{AdrLoadI32}(p)\,\} \\ \textbf{fun}\ \texttt{set}\ (p,i,h)\ \{\,\texttt{AdrStoreI32}(p,i)\,\} \\ \textbf{fun}\ \texttt{adr}\ (p,i,h)\ \{\,\texttt{AdrAdd}(p,\texttt{I32Mul}(i,\mathrm{Sz}(\texttt{int})))\,\} \end{array}\right\}$$

$$\mathcal{E}[\![Ct**]\!] = \left\{\begin{array}{l} \textbf{fun}\ \texttt{get}\ (p,h)\ \{\,\texttt{AdrLoadP}(p)\,\} \\ \textbf{fun}\ \texttt{set}\ (p,q,h)\ \{\,\texttt{AdrStoreP}(p,q)\,\} \\ \textbf{fun}\ \texttt{adr}\ (p,i,h)\ \{\,\texttt{AdrAdd}(p,\texttt{I32Mul}(i,\mathrm{Sz}(Ct*)))\,\} \end{array}\right\}$$

$$\mathcal{E}[\![Ct*]\!] = \left\{\begin{array}{l} \textbf{fun}\ \texttt{get}\ (p,h)\ \{\,p\,\} \\ \textbf{fun}\ \texttt{set}\ (p,q,h)\ \{\,\textbf{ccall}\ \texttt{memcpy}\,(p,q,\mathrm{Sz}(Ct))\,\} \\ \textbf{fun}\ \texttt{adr}\ (p,i,h)\ \{\,\texttt{AdrAdd}(p,\texttt{I32Mul}(i,\mathrm{Sz}(Ct)))\,\} \end{array}\right\} \text{where } Ct \notin \textsc{Atomic}$$

$$\mathcal{E}[\![\texttt{int}\,[\,n\,]]\!] = \left\{\begin{array}{l} \textbf{fun}\ \texttt{get}\ (p,i,h)\ \{\,\texttt{AdrLoadI32}(\texttt{AdrAdd}(p,\texttt{I32Mul}(i,\mathrm{Sz}(\texttt{int}))))\,\} \\ \textbf{fun}\ \texttt{set}\ (p,i,j,h)\ \{\,\texttt{AdrStoreI32}(\texttt{AdrAdd}(p,\texttt{I32Mul}(i,\mathrm{Sz}(\texttt{int}))),j)\,\} \\ \textbf{fun}\ \texttt{adr}\ (p,i,h)\ \{\,\texttt{AdrAdd}(p,\texttt{I32Mul}(i,\mathrm{Sz}(\texttt{int})))\,\} \end{array}\right\}$$

$$\vdots$$

$$\mathcal{E}[\![\textbf{struct}\ t]\!] = \{\ \mathcal{E}_{memb}[\![Cm]\!]\,\phi \mid Cm \in \Gamma(t) \text{ and } \mathcal{S}[\![\Gamma(t)]\!] = (\theta,\phi)\ \}$$

$$\mathcal{E}_{memb}[\![\texttt{int}\ m]\!]\,\phi = \left\{\begin{array}{l} \textbf{fun}\ \texttt{get\_m}\ (p,h)\ \{\,\texttt{AdrLoadI32}(\texttt{AdrAdd}(p,\phi(m)))\,\} \\ \textbf{fun}\ \texttt{set\_m}\ (p,i,h)\ \{\,\texttt{AdrStoreI32}(\texttt{AdrAdd}(p,\phi(m)),i)\,\} \\ \textbf{fun}\ \texttt{adr\_m}\ (p,h)\ \{\,\texttt{AdrAdd}(p,\phi(m))\,\} \end{array}\right\}$$

$$\vdots$$

Figure 5: The $\mathcal{E}$ translation

CKit type-checker produces a sequence of typed abstract syntax trees (ASTs) corresponding to the declarations in the header file. Generating an MBI file by applying our translations to these ASTs is straightforward.

In the remainder of this section, we illustrate *Charon* and our translations by showing the MOBY declarations that result when we invoke *Charon* on an example C header file. This file contains C declarations that specify the format of a vertex buffer in a graphics application:

```
struct rgba {
    unsigned char       r, g, b, a;
};
struct vb {
    struct rgba         c;
    float               v[3];
};
typedef struct vb *vbuf;
extern vbuf VB;
```

From these declarations, *Charon* generates the MOBY interface given in Figure 6. For each of the

```
type S_rgba
type PS_rgba <: CTypes.P_void
type S_vb
type T_vbuf <: CTypes.P_void
type A3_float

module S_rgba : {
  val get_r : S_rgba -> Int
  val set_r : (S_rgba, Int) -> Unit
  val adr_r : S_rgba -> CTypes.P_unsigned_char
  ...
  val get_a : S_rgba -> Int
  val set_a : (S_rgba, Int) -> Unit
  val adr_a : S_rgba -> CTypes.P_unsigned_char
  val size : Int
}

module PS_rgba : {
  val get : PS_rgba -> S_rgba
  val set : (PS_rgba, S_rgba) -> Unit
  val adr : (PS_rgba, Int) -> PS_rgba
  val cast : CTypes.P_void -> PS_rgba
}

module S_vb : {
  val get_c : S_vb -> S_rgba
  val set_c : (S_vb, S_rgba) -> Unit
  val adr_c : S_vb -> PS_rgba
  val get_v : (S_vb, Int) -> Float
  val set_v : (S_vb, Int, Float) -> Unit
  val adr_v : (S_vb, Int) -> CTypes.P_float
  val size : Int
}

module T_vbuf : {
  val get : T_vbuf -> S_vb
  val set : (T_vbuf, S_vb) -> Unit
  val adr : (T_vbuf, Int) -> T_vbuf
  val cast : CTypes.P_void -> T_vbuf
}

module A3_float : {
  val get : (A3_float, Int) -> Float
  val set : (A3_float, Int, Float) -> Unit
  val adr : (A3_float, Int) -> CTypes.P_float
  val size : Int
}

val global_VB : Unit -> T_vbuf;
```

Figure 6: A MOBY interface to the vertex buffer types

```
type S_rgba = prim ptr(⟨0 ▷ int8⁴⟩₄)
type PS_rgba = prim ptr(⟨0 ▷ int8⁴⟩₄)
type S_vb = prim ptr(⟨0 ▷ int8⁴, 4 ▷ float³⟩₁₆)
type T_vbuf = prim ptr(⟨0 ▷ int8⁴, 4 ▷ float³⟩₁₆)
type A3_float = prim ptr(⟨0 ▷ float³⟩₁₂)
...
fun S_rgba.get_g (p, exh) {
  AdrLoadU8(AdrAdd(p, 1))
}
...
fun S_vb.set_v (p, i, f, exh) {
  AdrStoreF32(AdrAdd(p, I32Add(4, Int32Mul(4, i))), f)
}
...
fun T_vbuf.adr (p, i, exh) {
  AdrAdd(p, I32Mul(i, 16))
}
...
```

Figure 7: The BOL implementation of the vertex buffer types

types in the closure of the header file, *Charon* defines an abstract MOBY type and a module that implements operations on the type. For common C types, such as "`void*`" and "`unsigned char`," we provide definitions in the `CTypes` utility module. The `CTypes` structure also provides access to `malloc`, which enables object allocation in the C heap. Since `malloc` returns a `void*` result, we provide `cast` functions for each C pointer type. As an example of how to use this API, if we wanted to set the red component of the second vertex in `vb` to `255`, we would write

```
S_rgba.set_r (
  S_vb.get_c (T_vbuf.get(T_vbuf.adr(vb, 1))),
  255)
```

which is equivalent to the C assignment "`vb[1].c.r = 255`." While not as concise as the C notation, the MOBY code permits the same fine-grained data manipulation that C does.

The implementation of this API in BOL is a straightforward application of the translation from Section 3.3; we give some excerpts in Figure 7. With inlining of these definitions and a standard *contraction* pass [App92], the red-component assignment example reduces to the BOL expression

```
AdrStoreI8 (AdrAdd (vb, 16), 255)
```

On the Intel IA32, this code can be implemented with a single move instruction. This example illustrates how our cross-module inlining facility gives direct access to C representations with C-like levels of efficiency.

4 Foreign-interface policies

While *Charon* provides a low-overhead way to import a C API into MOBY, the resulting translation does not take advantage of MOBY's stronger typechecking. In practice, we often want our foreign interfaces to fit the MOBY programming model. A typical C API requires that client programs respect various conventions that are not explicit in the types of the API. Often such conventions can be reflected in the type structure of a high-level interface to the API, but establishing the mapping requires some form of programmer intervention. For example, consider the following C function:

```
void f (char *s, char *c) { *c = s[1]; }
```

The two arguments to this function have exactly the same C type, even though the first is being used to refer to an array of characters and the second is being used to return a result. When defining a MOBY interface to such a function, we would like to reflect this distinction in the type given to the function. Such distinctions are embodied in a *policy* that guides the generation of a foreign interface from a specification. In this section, we describe several policy layers that are implemented on top of the mechanism of the previous section.

4.1 `moby-idl`

Many high-level language implementations provide a FFI mechanism based on specifications written in an *Interface Description Language* (IDL) [FLMP98, Ler99, PR00]. We have retargeted the SML/NJ IDL-based foreign-interface generator [PR00] to produce MBI files from an IDL specification.[5] 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. The *moby-idl* tool generates an abstract MOBY type for each non-trivial type and a stub function for each function prototype in the IDL specification. The generated stub function uses our foreign-interface mechanism to communicate values to and from the C code and also implements the IDL-specified policy by converting between the C and MOBY representations. Figure 8 illustrates this rôle of the generated stub function.

To make this discussion concrete, consider the following simple IDL specification of the interface to the UNIX `gettimeofday` system call:

```
typedef struct {
    long        tv_sec;
    long        tv_usec;
} timeval;
void gettimeofday ([out] timeval *tp);
```

This specification includes an **`out`** annotation on the argument of `gettimeofday`. The **`out`** annotation identifies the argument as a pointer to storage for returning a result of the function call. The *moby-idl* tool produces the following MOBY interface from this specification:

[5] There are a number of IDL variants; the SML/NJ tool (and *moby-idl*) accepts an extension of the OSF DCE dialect.

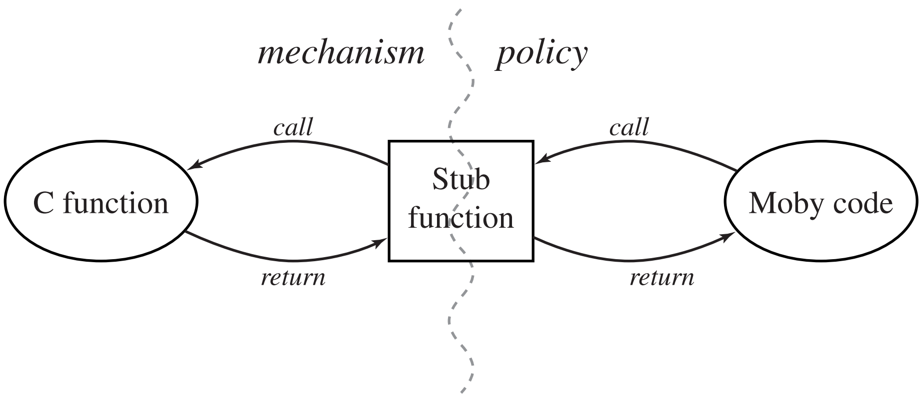

Figure 8: IDL function-call protocol

```
type Timeval
module Timeval : {
  val mk : (Int, Int) -> Timeval
  val get_tv_sec  : Timeval -> Int
  val get_tv_usec : Timeval -> Int
}
val gettimeofday : Unit -> Timeval
```

As in the translation of Section 3, the C `timeval` type has been translated to an abstract MOBY type with a supporting submodule of operations. Unlike the previous translation, however, the resulting type is a MOBY type (*i.e.*, nonmutable).

From the user's point of view, the *moby-idl* tool is similar to other IDL-based tools, such as *Camlidl* [Ler99], *H/Direct* [Fin99a], and *ml-idl* [PR00]. The main difference lies in the implementation of the stub functions that handle the marshalling and unmarshalling of data. These aforementioned tools all use a combination of low-level (*i.e.*, C) and high-level code to implement the foreign function-calling protocol. For instance, the *Camlidl* tool generates both an OCAML function and a C stub function for each IDL function specification. The generated OCAML function uses OCAML's standard foreign-function calling protocol to call the stub function, which is responsible for the actual marshalling and unmarshalling of arguments and results.

In contrast, the *moby-idl* tool generates a single MBI file that includes both the MOBY signature for the API as well as a BOL implementation of the stub functions. This reduces the overhead of calling foreign functions and also allows the stub function to be inlined at the call site. For example, the BOL implementation of the `gettimeofday` stub code is

```
fun timeofday (h) {
  stackalloc arg(8,4)
  let _ = ccall timeofday (arg);
  let s = AdrLoadI32(arg)
  let us = AdrLoadI32(AdrAdd(arg, 4))
  let res = alloc(s, us)
  return res
}
```

This code uses *stack allocation* to allocate temporary space for the result. The **`stackalloc`** constructs binds the address of 8 bytes of stack storage (with 4-byte alignment) to the variable `arg`. The extent of this storage is the scope of `arg` (the rest of the function in this case). After calling the C `gettimeofday` function, the code extracts the contents of the result from the stack and allocates a MOBY pair object on the heap, which is returned as the result of the MOBY `gettimeofday` function. This code illustrates layering a policy (i.e., the interpretation of output parameters) over our basic mechanism for interoperability.

4.2 Application-specific translation

While *Charon* provides a low-overhead way for a programmer to import a C API as a foreign interface in a MOBY program, we believe that users will be willing to spend extra effort to customize the translation for their particular application. We are working on a refinement of *Charon* that we call an *API miner*. The idea is that the user writes an application-specific translation script that describes how to translate the C header file (or files) to a MOBY API. At its simplest, the script performs *filtering* by defining which declarations in the header file should be extracted to the MOBY API. More sophisticated scripts control the policies used in the translation (the default policy is the identity policy used by *Charon*) and can inject additional definitions in the output that have no direct C correspondent. A script consists of rules that specify header-file components; each rule can be annotated with a policy that governs the translation to MOBY.

The simplest example of a rule is a translation of C names to MOBY names, such as the name mangling translations $\mathcal{N}_{\mathcal{T}}$ and $\mathcal{N}_{\mathcal{M}}$. Another example are IDL-style policies for controlling the interpretation of function arguments and C pointer types. We can also use policies to define a higher-level interface to pieces of an API. For example, the OpenGL API defines a large number of symbolic constants (over 70) that are logically grouped into 18 "*types*." In the C API, these constants all have type `int`, but we would like to define more type structure in a MOBY version of the API. We do this by defining an abstract type in the output for each of the logical types in the ABI (these definitions are an example of injecting additional definitions into the output). For the symbolic constants, we define policies that map them to an *abstract value constructor* [AR92] of the appropriate type. The advantage of this approach is that we get a high-level API while maintaining binary compatibility with the C implementation.

The API miner works by first constructing a database of C definitions (including preprocessor symbols) from the C header file. Then each rule in the script is executed as a query on the database; the policy associated with the rule is applied to the result of the query and the result is added to the generated MBI file. We are currently implementing this mechanism as an enhancement to *Charon* and plan to report on it in more detail in the final paper.

While both the API miner and the *moby-idl* tool provide a policy layer on top of the same underlying mechanism, there are a number of differences in what they support. The IDL tool is geared toward function-centric APIs, where the marshalling and unmarshalling of function parameters is not a serious performance issue.[6] In contrast, the API mining tool supports mixing raw C and marshalled representations in the same API. Another advantage of the API miner is that it uses the host

[6]While it is possible to pass pointers to C data values across an IDL specified API, the representation is abstract on the MOBY side.

system's header files (including preprocessor symbols) as the specification of the C API; this property makes the generation of the corresponding MOBY API more portable across different ABIs and more robust to changes in the C API.

5 Related work

We base our approach to foreign interfaces on a low-level data-interoperability mechanism that serves as the foundation for a wide range of different interoperability policies with different user-level mechanisms. Our approach contrasts with most of the prior work on language interoperability, which fixes a particular interoperability policy and user-level mechanism.[7]

We organize the relevant prior work on foreign interfaces into three groups based on the user's view of the mechanism. These groups also happen to correspond to various aspects of our approach. The first consists of low-level mechanisms for programming the interface between the high-level and C representations; we compare this group to our BOL-based data-interoperability mechanism. The second consists of high-level libraries for manipulating C types; we compare these to *Charon*. The last group consists of foreign-interface generators, including IDL-based tools; we compare these to our *moby-idl* and API miner tools. Note that the tools of the third group often target the mechanisms of the first group.

The first group of prior work consists of mechanisms where the user writes wrapper code that handles the interface between the high-level and C representations. Typically, the wrapper code is written in C, as is the case in SML/NJ, OCAML [Ler98], and the JAVA Native Interface (JNI) [Lia99]. Some implementations make it possible to write such code in the high-level language. For example, the SML'97 Basis Library [GR00] and the Glasgow HASKELL compiler (GHC) [GHC00] both provide operations for reading and writing scalar values in a bytearray. Thus one can manipulate a C data structure by importing it into a program as a bytearray, but 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, no bounds checking is done and it is possible to read and write pointer values. The combination of this mechanism and GHC's new foreign function interface [Fin99b] provides most of the expressiveness of our BOL-based mechanism, but it does not support the full range of C function prototypes (*e.g.*, **`struct`** parameters are not supported). BOL also supports stack allocation of temporary storage, which is useful for implementing data marshalling policies.

The second group of prior work consists of mechanisms where a high-level language API is provided for describing C types. For example, Huelsbergen's C Interface library for SML/NJ [Hue96] defines datatypes that represent the abstract syntax of C types. The user uses these datatypes to construct a value that represents the type of her foreign function. The run-time system interprets this value to drive the marshalling of arguments and results. The MLWorks foreign interface [Har98] uses a similar mechanism. The main difference between this work and our embedding of C types into MOBY is that our mechanism supports direct manipulation of the C representations, whereas these other mechanisms use data marshalling. Also, our *Charon* tool uses the existing C header file as its specification, instead of requiring the user to supply a redundant specification in a different

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

notation.

The third group of prior work consists of mechanisms where the user writes an interface description in a specification language and uses a tool to generate the glue between the high-level language and C. Most recent examples of this approach are based on IDL variants, including H/Direct [FLMP98], *Camlidl* [Ler99], and *ml-idl* [PR00]. These tools differ from *moby-idl* in that they generate C code to handle data-marshalling, whereas we use our data-interoperability facility to handle data marshalling on the MOBY side of the interface.

There are other examples of foreign-interface generators that are not based on IDL. *Green Card* is a tool for generating the stub code of a foreign interface for HASKELL [PNR97]. It supports a rich language for specifying data marshalling, but it does not support direct access to C data representations. *SWIG* [Bea96] is a tool for generating scripting interfaces to C, C++, or Objective-C programs. Its specification language is essentially C header files with policy annotations; the policy specification language is richer than IDL and includes a mechanism for specifying mappings between C and high-level types (*e.g.*, mapping between arrays and lists). Like *Green Card*, *SWIG* does not support data-level interoperability. *FFIGEN* [Han96] is a *front-end* that takes a C header file as input and produces a, so-called, *rational translation* of the interface, which is essentially a collection of data structures describing the declarations of the header file. An application-specific back-end is then run on this representation to generate the actual foreign interface. *FFIGEN* can implement any foreign-interface policy that the target SCHEME system can support. The API miner tool that we are developing (see Section 4.2) is partially inspired by *FFIGEN*. Our use of *CKit* to process header files is similar to what the *FFIGEN* front-end does, but we are using a specification language (more like *Green Card* and *SWIG*) to define the translation.

This paper addresses interoperability between code written in a high-level language and C running in the same address space. There has also been work on interoperability between different high-level languages in the same address space (*e.g.*, SML and JAVA) [BK99, Rei98, TS99]. Another related area is *distributed* interoperability, where the interoperating components live in different address spaces, or even on different machines [Obj93]. Since, in distributed interoperability all data must be marshalled by definition, data-level interoperability is not possible.

6 Conclusion

We have described how the MOBY compiler supports foreign interfaces. The basis of our approach is a mechanism that provides direct manipulation of C data representations from MOBY. This data-level interoperability is important for applications that need high-bandwidth communication between components written in different languages. It also serves as the foundation for a wide range of policies for translating C APIs into MOBY. We have described two such translation policies and the tools that implement them. The first is a direct embedding of C representations into MOBY and is implemented by the *Charon* tool. The second is a translation based on IDL specifications and is implemented by *moby-idl*. We have also described our API miner, which is tool that we are developing. The API miner does not implement a fixed translation policy, but rather provides a scripting language for defining application-specific policies. In addition to supporting a range of different policies, our low-level mechanism also has the advantage of providing tight integration of the foreign-interface glue code and the client MOBY code. Because the glue code is specified in

BOL, the compiler can inline it at the foreign-function call site.

We have been able to implement our foreign interface support with relatively little programming effort. This is because our compiler already had support for low-level data manipulations in BOL and a mechanism for importing externally defined BOL functions. We were also able to take advantage of existing infrastructure in the form of the *CKit* library and the existing *ml-idl* tool.

In the future, we intend to extend this work to support policies that use MOBY's object-oriented features in the foreign interface to low-level object-oriented APIs. These policies would be useful for object-oriented C APIs (*e.g.*, *GTk* [Pen99]), COM interfaces, and C++ APIs.

References

[App92] Appel, A. W. *Compiling with Continuations*. Cambridge University Press, Cambridge, England, 1992.

[AR92] Aitken, W. E. and J. H. Reppy. Abstract value constructors: Symbolic constants for Standard ML. *Technical Report TR 92-1290*, Department of Computer Science, Cornell University, June 1992. A shorter version appears in the proceedings of the "ACM SIGPLAN Workshop on ML and its Applications," 1992.

[Bea96] Beazley, D. M. SWIG: An easy to use tool for integrating scripting languages with C and C++. In *USENIX Tcl/Tk Workshop*, July 1996.

[BFRS99] Bonachea, D., K. Fisher, A. Rogers, and F. Smith. Hancock: A language for processing very large-scale data. In *DSL'99*, October 1999, pp. 163–176.

[BK99] Benton, N. and A. Kennedy. Interlanguage working without tears: Blending SML with Java. In *ICFP'99*, September 1999, pp. 126–137.

[Fin99a] Finne, S. *HaskellDirect User's Manual*, November 1999. Available from `http://www.dcs.gla.ac.uk/fp/software/hdirect/`.

[Fin99b] Finne, S. *A primitive foreign function interface*, October 1999. Available from `http://www.dcs.gla.ac.uk/fp/software/hdirect/`.

[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.

[GHC00] *The Glasgow Haskell Compiler User's Guide*, version 4.06 edition, January 2000.

[GR00] Gansner, E. R. and J. H. Reppy (eds.). *The Standard ML Basis Library*. Cambridge University Press, Cambridge, England, 2000. *To appear.*

[Han96] Hansen, L. T. FFIGEN manifesto and overview. Available from `http://www.ccs.neu.edu/home/lth/ffigen/`, February 1996.

[Har98] Harlequin Ltd. *MLWorks Reference Manual*, version 2.0 edition, July 1998.

[Hue96] Huelsbergen, L. A portable C interface for Standard ML of New Jersey. *Technical report*, AT&T Bell Laboratories, January 1996.

[Ler98] Leroy, X. *The Objective Caml System (release 2.00)*, August 1998. Available from `http://pauillac.inria.fr/caml`.

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

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

[MCG+99] Morrisett, G., K. Crary, N. Glew, D. Grossman, R. Samuels, F. Smith, D. Walker, S. Weirich, and S. Zdancewic. Talx86: A realistic typed assembly language. In *1999 ACM SIGPLAN Workshop on Compiler Support for Systems Software (WCSSS '99)*, 1999, pp. 25–35.

[MWCG98] Morrisett, G., D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. In *POPL'98*, January 1998, pp. 85–97.

[Obj93] Object Management Group. The Common Object Request Broker: Architecture and Specifications (revision 1.2). *OMG document 93.12.43*, Object Management Group, 1993.

[Pen99] Pennington, H. *GTK+/Gnome Application Development*. New Riders, Indianapolis, IN, 1999.

[PNR97] Peyton Jones, S., T. Nordin, and A. Reid. Green Card: a foreign-language interface for Haskell. In *Haskell'97*, June 1997.

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

[Rei98] Reinke, C. Towards a Haskell/Java connection. In *IFL'98*, LNCS, New York, NY, September 1998. Springer-Verlag, pp. 200–215.

[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.

[TS99] Trifonov, V. and Z. Shao. Safe and principled language interoperation. In *ESOP'99*, LNCS, New York, NY, March 1999. Springer-Verlag, pp. 128–146.