A Portable C Interface for Standard ML of New Jersey

Lorenz Huelsbergen
AT&T Bell Laboratories*
`lorenz@research.att.com`

January 15, 1996

Abstract

This paper describes the design and implementation of an interface to C for the SML/NJ ML compiler. The interface supplies ML datatypes with which programmers specify C types and C data. An ML program uses these datatypes to register a foreign C function with the interface and to build specifications of structured C data. The interface automatically instantiates C function arguments from C data specifications upon foreign function application. Most C types, including aggregate and function-pointer types, are supported. A runtime code generation technique converts ML closures to C-callable function pointers. Function pointers allow C programs to call ML programs. We solve the problems due to differences in data representation, function calling conventions, and storage management by copying data between the ML and C heaps, converting representations and changing calling conventions in the process. We find that this copying strategy provides adequate performance in practice. The interface is portable in the sense that its implementation does not require changes to the SML/NJ compiler proper; it is isolated in a pair of libraries (ML and C). The interface has already found use in several non-trivial applications.

1 Introduction

The *de facto* method for distributing reusable software modules is through C language function libraries and C++ language class libraries. This poses a severe problem for mostly-functional languages since the efforts in supporting their implementation are usually small relative to the industrial development of C-based tools and libraries. The problem is not that such languages cannot be used to create competitive applications—it has been persuasively demonstrated that mostly-functional languages are well-suited for system-level programming (*e.g.*, [5])—rather, it is difficult, expensive, and arguably unproductive, for modern languages to individually reproduce the functionality of extant C software libraries. Mostly-functional languages must therefore exploit existing C libraries by providing C or C++ interfaces [6, 13]. To this end, we have designed and implemented a general interface to C for the Standard ML of New Jersey (SML/NJ) ML compiler [3] that ameliorates the issue of language interoperability.

Our interface allows SML/NJ programs[1] to call arbitrary C functions, to create and manipulate C data from within SML/NJ, and generally to exchange data with C-based applications. The interface supports most[2] C data types: `int`, `short`, `long`, `char`, `union`, `struct`, array, function, and pointer. Conversion between ML functions (closures) and C functions (function pointers) is possible; this allows programmers

*Address: Room 2C-307, 600 Mountain Ave. Murray Hill, NJ 07974. Phone: (908) 582 4628

[1] Our interface design is also applicable to other language implementations with similar compilation and runtime strategies.
[2] Bit-fields and enumeration types are currently absent.

to write callback functions for user interfaces completely in ML, for example. We devised a runtime code generation technique—similar to the one used in the *esh* Scheme/C interface [13]—that converts ML closures to C function pointers without changes to SML/NJ's code generator. Since ML functions can be converted to C function pointers, it follows that C programs can use this interface to call SML/NJ programs. The interface is parameterized by the characteristics of the target C compiler including data-type sizes, aggregate alignment, and byte order (endianness); it is therefore possible to instantiate interfaces for multiple C compilers (*i.e.*, data formats) in a portable fashion. This is useful for binary C data-file transport between applications, for example.

Our interface supports multiple *type views* for a function by admitting multiple registrations of the function. This is useful because C often overloads the types of function arguments. *Dynamic typing* checks that the arguments supplied by an ML program to a foreign function match the registered type; a type mismatch in an application of a foreign function raises an ML exception at runtime.

Portability of the interface's implementation is also a concern. The design of SML/NJ enables rapid and robust evaluation of ML programs through novel compilation techniques (*e.g.* [4, 2]) and efficient run-time storage management [12, 1]. A tradeoff however is that this design does not readily admit an interface to C. SML/NJ's function calling convention and data representations differ radically from C's. Reengineering SML/NJ to directly support linkage with C would disrupt the extensively-tuned optimizations already in place. This forces the implementation of the interface to consist of stand-alone libraries: a C library that is part of the SML/NJ runtime system and an SML/NJ library that is loaded prior to foreign function registration and their subsequent application in ML. Both libraries share a protocol that communicates types and data between the two languages.

Efficiency of the interface is only a secondary concern. We assume that a call to a foreign function f usually triggers considerable computation in f (*e.g.*, an expensive system call); for otherwise f could simply be coded in the native language. In particular, a call to f with parameter data of size n and return result of size m requires work $O(n+m)$ in our interface—the interface makes copies of parameters (with which the foreign function f operates) and return values. In practical applications of this interface, we have not found this inefficiency limiting.

Several applications have been constructed using our C interface for SML/NJ. The most extensive application to date is an ML interface to the Microsoft Win32 programmer interface [10]; over 1300 interfaced Windows C function provide SML/NJ with graphics and window management functions, *etc.* Other applications include ML interfaces to Tcl/Tk [11] and to the Unix DBM database management functions.

The next section describes the major problems in interfacing a mostly-functional language to C. Sections 3 and 4 respectively describe the design and implementation of our C interface for SML/NJ; sections 5 and 6 contain directions for future work and a discussion of previous interface designs.

2 Issues and Architecture

The architecture of the SML/NJ compiler [3] and runtime system [12, 1] pose three major obstacles to a foreign function interface: incompatible data representation, incompatible function calling conventions, and automatic storage management (copying garbage collection). In this section we discuss these obstacles and describe the interface architecture we designed to accommodate them.

2.1 Data Representation

SML/NJ differs from C in its representation of both base and aggregate data.

In SML/NJ, integers are represented as 31-bit numbers within a 32-bit machine word[3]; the extra (low-order) bit is set to distinguish an integer from a pointer which is always word aligned (*i.e.*, low-order bit clear). This distinction is required by the SML/NJ garbage collector. In C, the integer type `int` usually corresponds to an entire word, whereas the integer types `short` and `long` may respectively be a portion of a word or multiples of words. Similarly, SML/NJ real values are always 64 bits whereas C compilers may use different sizes to represent the values of their `float` and `double` types. Characters are stored in bytes

[3] A 32-bit word size is assumed in this paper and is referred to as *word*.

in SML/NJ and in most C implementations. SML/NJ strings are always null-terminated; C's "strings" are pointers to characters (`char *`) which are sometimes—but not always—null-terminated.

Since aggregates are constructed from base values, the representation of the former is affected by the representation of the latter. The physical layout of ML tuples, records, and arrays—as well as the representation of datatypes—is not specified by the language. C aggregates (`struct`, `union` and array data) follow conventional C layout rules (*cf.* [8]). This complicates the interface because an invertible mapping for aggregates is not always possible without explicit programmer directives. For example, an ML record does not directly map to a C `struct` since the order of record members is dependent on the implementation whereas lexical member order is vital for C programs.

2.2 Calling Convention

ML treats functions as having a single argument; tuples and currying provide for multiple arguments. The SML/NJ compiler implements control-flow via the continuation-passing style (CPS) [3, 14]. SML/NJ then further performs many optimizations (*e.g.* [2]), some of which are calling convention specific [4]. In contrast, a given C compiler's function calling convention is partly governed by the instruction-set architecture of the target machine.[4]

A direct but unwieldy solution to this problem is to replace SML/NJ's backend with one that emulates the conventions of C compilers. Such an approach however has severe disadvantages: (1) the compiler would have to support (efficient) code generation for many platforms, most with multiple C compilers and hence slightly different conventions; and, (2) the implementation would lose the benefits of CPS compilation, *e.g.*, inexpensive continuations.

2.3 Copying Garbage Collection

A further interface issue concerns storage management. ML requires automatic storage management in the form of garbage collection. Since ML abstracts from the notion of a physical machine address, implementations are free to use copying garbage collectors that may move a datum during its lifetime. (SML/NJ uses generational copying collection [12, 1]). C programmers, conversely, have unrestricted access to a datum's address and can, through encoding, effectively "hide" addresses from their storage manager, *i.e.* from `malloc` and `free`. C programmers maintain the invariant that—unless explicitly moved—a datum at address p remains at that address throughout its lifetime. Therefore, in a copy-collected ML implementation, data must be "pinned down" before it can be safely passed to C; otherwise the GC may move it while C still has active pointers to the old, now invalid, address. It does not suffice to suspend ML GC during a call to C since a C function can retain pointers for use by subsequent C calls. In the presence of function pointers, a C function may also reenter ML which in general requires ML allocation and hence garbage collection.

2.4 Interface Architecture

Our interface design addresses the issues above as follows. Suppose ML calls the C function f with arguments $a_0, \ldots, a_n$ and return value r. Using a supplied datatype (described below, §3.4), the ML program creates ML data that corresponds to f's n arguments. The interface understands the internal representation of this datatype and, on initiation of the call, first copies f's arguments to the C heap. In making this copy, the interface changes data representation when necessary (*e.g.*, removes SML/NJ's tag bit on integers). The function f is then called with the converted arguments in the C heap. When f returns, its return value is similarly copied and transformed back to the ML datatype. Copying ML data to and from the C heap in this manner enables the interface, when necessary, to:

1. convert between data representations,
2. change function calling conventions via a dynamically-generated *code wrapper*, and to
3. generate a fixed address for data passed to C.

[4] On some processors (*e.g.*, MIPS R4000) a small number of function arguments are passed in registers if possible; remaining arguments are passed through the C stack.

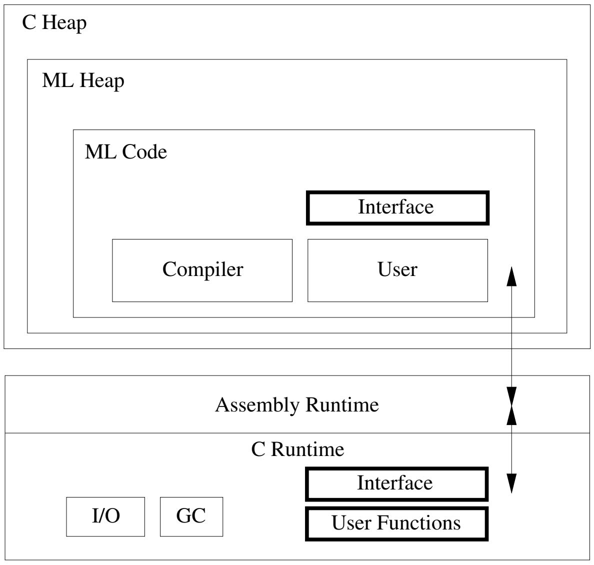

Figure 1: SML/NJ memory model. The ML heap resides in the runtime system's C heap and contains both ML data and code. The C interface consists of an ML library and a C library along with a collection of user C functions. Control can flow from the ML interface library to the C interface library, and *vice versa*, via the runtime system.

Figure 1 depicts SML/NJ's memory model (*cf.* [12, 1]). When SML/NJ starts, its runtime system—a C and assembly language program—creates the ML heap within the conventional C heap. All ML code and data is placed in this ML heap. Subsequent ML allocation occurs in the ML heap. Control can pass between an ML program and the runtime system via set of assembly-language hooks that save and restore ML register state. Among other services, the runtime system also performs I/O and manages storage.

The C interface consists of an ML library in the ML heap and a C library in the runtime system. Interfaced C functions reside in the runtime system. The ML interface library provides functions for registering foreign C functions and for specifying and manipulating C data. A call to an interfaced C function first enters the C interface in the runtime (control flows along the arrows in Figure 1). When the C call completes, control returns from the runtime to the ML caller. The next section describes the programmer's view of the ML interface library and its interaction with the C interface library.

3 The Interface

The section's description of the ML interface to C models the order of the steps that a programmer uses to interface a C function: interface instantiation for a target C compiler, C function linkage, C type and data specification, C function registration, and C function invocation. In this section we also describe auxiliary functions for converting data between ML and C formats and a function for computing the sizes of C data.

3.1 Instantiation

The ML part of the interface is a functor parameterized by a structure matching the `CC_INFO` signature of Figure 2. Structures with `CC_INFO` signatures describe the C compiler characteristics that are relevant to the interface. An interface for a target C compiler is instantiated by applying the functor to a `CC_INFO` structure.

A `CC_INFO` structure describes the sizes, in bytes, of the C compiler's base types: `int`, `short`, `long`, `char`, `float`, `double`, and pointer types.

The interface permits C `struct` and `union` types to use nonconventional alignment; that is, a non-`NONE` `structAlign` or `unionAlign` field will use the supplied alignment to align `struct` and `union` types instead

```
signature CC_INFO =
   sig
      (* all sizes in bytes *)
      val intSzB : int
      val shortSzB : int
      val longSzB : int
      val charSzB : int
      val floatSzB : int
      val doubleSzB : int
      val ptrSzB : int
      val unionAlign :  int option^a   (* alternate alignment *)
      val structAlign :  int option   (* alternate alignment *)
      val bigEndian :  bool
      val nConventions :  int         (* number of calling conventions *)
   end (* signature CC_INFO *)
```

[a] `datatype 'a option = SOME of 'a | NONE`

Figure 2: Signature for a structure describing characteristics of a target C compiler: data sizes, aggregate alignment, endianness, and the number of function calling conventions.

of the default alignment conventionally dictated by the member field's types (*cf.* [8]). Alternate alignments are sometimes necessary for system-level programming, *e.g.*, [10].

The `bigEndian` field is *true* (*false*) if the compiler stores 4-byte words with the low-order byte at the lowest (highest) byte address in the word. A C compiler typically inherits this attribute from the host computer.

The `nConventions` field indicates the number of function calling conventions that the target C compiler supports.[5]

In practice we use a simple C program to automatically generate a `CC_INFO` structure for a C compiler; this program computes data sizes and probes for alignments and endianness.

3.2 Linkage with C Functions

We now describe the mechanism for adding a C function to the set of interfaced functions available to SML/NJ programs. Suppose the C function f has been separately compiled and now resides in an object file or library. To make this function visible from SML/NJ, one includes an entry of the form

`CFUNCTION("`f`", `f`);`

in a runtime header file designated for such entries. The macro `CFUNCTION` associates the string `"`f`"` with the function's C symbol (address) f. This association allows SML/NJ to dynamically find the address f when registering it from ML. It is now only necessary to recompile the affected portions of the runtime system and link with the file or library containing f's object code.[6] As a running example, consider the hypothetical function `create_window` as described by the ANSI C prototype:

```
int create_window(void callback(int, int), struct point *pp);        (1)
```

The arguments to `create_window` are a function pointer and a pointer to a `struct`. The function pointer must point to a function of two integer parameters and no return value. The `point struct` is declared as:

```
struct point { int x, y; };        (2)
```

[5] This is necessary, for example, when interfacing to functions in Microsoft Win32 libraries [10] since they are compiled using a Pascal calling convention.

[6] It is straightforward to replace this static compile-time linkage mechanism with a dynamic one based on shared or dynamically-linked libraries, for example.

```
signature C_CALLS =
   sig
       ⋮
       (* function calling convention *)
       type convention = int

       (* ctype:  constructors for specifying C types *)
       datatype ctype =
           CcharT
         | CintT | ClongT | CshortT
         | CfloatT | CdoubleT
         | CfunctionT of (convention * ctype list * ctype)
         | CptrT of ctype | CaddrT
         | CstringT
         | CarrayT of (int * ctype) | CstructT of ctype list | CunionT of ctype list
         | CvoidT
       ⋮
   end
```

Figure 3: The `ctype` datatype in the `C_CALLS` signature.

Parameter `pp` is perhaps an initialization structure (containing only a screen position, in this simple example). To further this example, suppose that the function `create_window` returns an integer handle h for the new window and that the `callback` function receives h as its first parameter and an integer message code as its second parameter for every event affecting window h. The entry in the SML/NJ runtime system

```
CFUNCTION("C_create_window", create_window);
```

enables the C interface to find `create_window` via the name `"C_create_window"`. Note that the `CFUNCTION` macro conveys no information about the type of the function that is being interfaced. We chose to specify the type of an external function from within ML because C functions often overload parameters and return results with multiple types. For example, in `create_window` a `NULL` `pp` parameter might be used to select default window placement. Specification of a function's type in the ML interface library—instead of in the C interface library—sidesteps this problem.

The signature of an instantiated C interface (`C_CALLS`) is distributed among Figures 3–5; we now describe this signature's datatypes and values.

3.3 C Type Specification

In the interface, programmers describe the types of C data—such as the types of an interfaced function's parameters and return values—with an ML datatype called `ctype` (Figure 3). C's base types map directly to the nullary `ctype` constructors `CcharT`, `CdoubleT`, `CfloatT`, `CintT`, `ClongT`, and `CshortT`.

Aggregate types are specified using base types (and existing aggregate types). The type of a C array (`CarrayT`) carries a pair containing the number of array elements and their type (a `ctype`). C's `struct` and `union` types are specified as a list of the `ctype`s of their member fields.

For C pointer types, the interface provides two constructors: `CptrT` and `CaddrT`. The `CptrT` constructor enables construction of linked C data from within ML. The `CaddrT` constructor denotes an arbitrary C address (*e.g.*, a `(void *) pointer`); it serves as a handle to C addresses that can point to C data in the C heap with unknown types (*cf.* [6]). Manipulation, in ML, of values of type `CaddrT` is described below (§3.4).

The `CstringT` constructor is shorthand for a null-terminated array of characters, *i.e.* a C `(char *)` "string." Its inclusion in the set of `ctypes` was motivated by its frequent use in C programs.

```
signature C_CALLS =
   sig
       ⋮
       (* caddr:  an abstract pointer type *)
       eqtype caddr
       val NULL : caddr
       val free :  caddr -> unit
       val index :  (caddr * int) -> caddr
       val difference :  (caddr * caddr) -> Word32.word

       (* cdata:  constructors for specifying instances of C data *)
       datatype cdata =
           Cchar of char
         | Cint of Word32.word | Clong of Word32.word | Cshort of Word32.word
         | Cfloat of real | Cdouble of float
         | Cfunction of (cdata list -> cdata)
         | Cptr of cdata | Caddr of caddr
         | Cstring of string
         | Carray of cdata Vector.vector | Cstruct of cdata list | Cunion of cdata
         | Cvoid
       ⋮
   end
```

Figure 4: The cdata datatype and the abstract caddr type in the C_CALLS signature.

The CvoidT constructor is primarily used to indicate that a C function does not return a value. The other common use of C's void declaration is to declare pointers to arbitrary types; CaddrT serves this purpose in our interface.

The CfunctionT constructor describes the type of a C function pointer. The three components of the constructor's argument tuple specify the function's calling convention[7], the function's argument types (a ctype list) and the type of its return value (a ctype).

In the create_window example, struct point (2) has the ctype:

```
val CpointT = CstructT [CintT, CintT]                (3)
```

Function create_window's return value has CintT as its ctype. Its callback parameter has the ctype

```
val CcallbackT = CfunctionT (0, [CintT, CintT], CvoidT)                (4)
```

and its pp parameter has, using the CpointT binding (3), the ctype: CptrT CpointT .

3.4 C Data Specification

Figure 4 contains the portions of the C_CALLS signature that comprise the cdata datatype. The cdata datatype is used to specify—using ML data within an ML program—a particular instance of a C type. That is, with cdata one describes the structure and content of C data. From a cdata value, the interface will automatically construct, when necessary (before a C function call, for example), a C instance of this data in the C heap. Constructors in the cdata datatype correspond to constructors in the previously described ctype datatype (§3.3). Additionally, the cdata datatype requires the abstract caddr type to represent arbitrary C addresses (see Figure 4).

[7] A calling convention is specified with an integer $0, \ldots,$ nConventions $- 1$, where nConventions > 0. See §3.1 and §4.

The constructors `Cchar`, `Cdouble`, `Cfloat`, `Cint`, `Clong`, and `Cshort` specify base C values. Where possible, a `cdata` constructor carries the naturally-corresponding ML value.[8] In the case of integer data, we use the SML/NJ `Word32.word` type[9] instead of `int` in order to minimize data-representations conflicts. ML's `real` type is used to specify instances of C's `float` and `double` types.

Aggregate `cdata` types are specified from base C data or recursively from existing aggregate `cdata`. The array constructor `Carray` carries an immutable SML/NJ vector of `cdata`. We use SML/NJ vectors here instead of arrays because the vector type reflects the interface's copying semantics. `Cstruct` carries a list of `cdata` that specifies the content of the `struct`'s member fields. `Cunion` specifies an instance of a C `union` type and hence carries only a single member.

Application of the `Cptr` constructor to a `cdata` value specifies a C pointer (one level of indirection) to an instance of this data. The `Caddr` constructor carries values of type `caddr`. The `caddr` type is an abstract type that represents a C address p without regard for the type of the data at p. Specifically, the interface will never copy the data at p into the SML/NJ heap. `Caddr` is useful when, for example, a C function returns a pointer to ML that ML will subsequently pass back to C. The interface supplies C's `NULL` pointer value as a `caddr` value, as well as functions on `caddrs` that perform byte-based pointer arithmetic (`index` and `difference`). The function `free` deallocates the storage pointed to by its `caddr` argument.[10] This function is included in the interface to provide explicit control over the lifetimes of `cdata` values transferred to the C heap; see §3.5 and §3.8 below. The `caddr` type is similar to a *handle* to a C++ object in the Talk (Lisp) interface [6].

`Cstring` specifies a null-terminated C string constant. `Cvoid` specifies the absence of a data value.

`Cfunction` carries an ML function. The carried function must map a list of `cdata` parameter values to a `cdata` return result.

One can now specify data for the `create_window` example (§3.2). The function

```
fun simpleWindowFn [Cint h, Cint msg] =
    case msg of
         0 => closeWindow h
       | 1 => printWindow (h, "hello")
```

is a valid `callback` parameter to `create_window` when tagged as a `Cfunction`:

```
val callbackParam = Cfunction simpleWindowFn        (5)
```

A sample initialization structure for `create_window` that places the window at the origin is:

```
val initParam = Cptr (Cstruct [Cint 0, Cint 0])        (6)
```

The `create_window` function must now be registered in ML as a foreign C function before it can be applied to `callbackParam` and `initParam`.

3.5 C Function Registration

Figure 5 completes the `C_CALLS` signature with ML functions that *register* previously (§3.2) linked C functions.

The first argument to the `registerCFn` function is an integer that specifies a calling convention for the C function f being registered (see `CfunctionT`, §3.3). The next argument is a tuple giving the name of the function (a `string`), the types of f's parameters (a `ctype list`) and the type of f's return value (a `ctype`). The name is used to bind C's address for the function; the types are used to convert parameters and return values. Application of a second argument to `registerCFn` produces an ML function f' that embodies the actual call to f.

The value returned by the registered C function f' is a tuple (d, l). The value d is of type `cdata` and contains f's return value. The second tuple component, l, is a list of values of type `caddr` (§3.4). The addresses in l point to the C storage dynamically allocated by the interface in calling f. In particular, l

[8] Interface instantiation (§3.1) raises an exception if SML/NJ's base types cannot, due to size, hold C's base types; *i.e.* an extension to the `cdata` datatype is required to support 64-bit C integers.

[9] The `Word32` structure provides signed and unsigned conversions to and from integers.

[10] The `free` function is a direct binding to the C library's `free` function, and thus inherits its semantics.

```
signature C_CALLS =
   sig
       ⋮
       exception DynamicTypeMismatch

       (* function calling convention *)
       type convention = int

       (* registration functions *)
       val registerCFn :  convention ->
                          (string * ctype list * ctype) ->
                          (cdata list -> (cdata * caddr list))
       val registerAutoFreeCFn :  convention ->
                                  (string * ctype list * ctype) ->
                                  (cdata list -> cdata)

       (* data conversion functions *)
       val dataMLtoC : ctype -> cdata -> (caddr * caddr list)
       val dataCtoML : ctype -> caddr -> cdata

       (* sizeOf:  number of bytes a ctype instance occupies in the C heap *)
       val sizeOf :  ctype -> int

       ⋮
   end
```

Figure 5: Functions in the C_CALLS signature to register C functions, to export and import data, and to compute C data sizes. DynamicTypeMismatch is raised when the type of an argument to a C function does not match its registered type.

contains pointers to C-heap copies of parameters and return values.[11] The programmer must free (using, *e.g.* `app free` l) the interface's copies when they are no longer needed; this mechanism is necessary since C functions may, in general, retain pointers to their arguments. The function `registerAutoFreeCFn` is similar to `registerCFn` except that it automatically frees all storage allocated by the interface—but not the storage allocated during f's execution, if any—before returning to ML.

Assume that the example function `create_window` has been added to the SML/NJ runtime (§3.2) and that the bindings `CcallbackT` (4) and `CpointT` (3) exist. The expression

```
val createWindow = registerAutoFreeCFn 0 ("C_create_window",CcallbackT,CptrT CpointT)   (7)
```

registers `create_window` with the ML interface library. Registration with `registerAutoFreeCFn` implies that the function `create_window` does not retain pointers to its (non function) parameters beyond the extent of its execution; *i.e.*, data in the `struct` pointed to by `pp` is accessed only during a call to `create_window`.

3.6 Calling C from ML

The ML function f'—derived from the C function f via registration—can now be applied to a `cdata list` l that supplies f's actual parameters. In addition to converting the parameter $p \in l$ to C format, our interface checks that the type registered for p matches the type implied by p's `cdata` representation. If a type mismatch occurs, the interface raises the `DynamicTypeMismatch` exception (*cf.* dynamic typing). In practice, it is convenient to further shroud f' in a wrapper function whose type does not contain `cdata`.

In the example, application of `createWindow` (7) to `callbackParam` (4) and `initParam` (3)

```
val (Cint h) = createWindow callbackParam initParam          (8)
```

invokes C's `create_window` and produces`(Cint h)` where `h` is the new window's handle.

3.7 Calling ML from C

Calling ML from C is accomplished through function pointers. Suppose that the SML/NJ runtime provides C's `main` entry point. Furthermore, suppose that the entry point to the C program is (re)named `entry` and is extended to accept—in addition to `argc`, `argv`, *etc.*—a set of function pointers. An ML program need then only register `entry` as a C function. It then calls `entry` with the set of ML functions that are available to the C program.

As an example, consider the renamed `main` function of a C program:

```
int (*ML_entry)(int);   /* storage for an ML function */

int entry(int (*f)(int), int argc, char *argv[])
{
    ML_entry = f;        /* save ML function for later use */
    ⋮                    /* continue with C program's main */
}
```

In addition to the conventional command-line arguments, `entry`'s first parameter is an ML function that it stores in `ML_entry` for future application.

A model where the C program retains C's `main` is also possible, but is not described here.

3.8 Data Conversion Functions

The `C_CALLS` interface structure (Figure 5) also provides two functions (`dataMLtoC` and `dataCtoML`) that directly convert `cdata` in the SML/NJ heap to C data in the C heap and *vice versa.* The function `dataMLtoC` ensures that its `cdata` specification matches its `ctype` parameter before converting the former to its C form. As with `registerCFn` (§3.5 above), `dataMLtoC` returns, in addition to the address of the converted datum,

[11] In our implementation (§4), function pointers created via `Cfunction` do not create deallocatable storage in the C heap and hence are not included in the list l of storage pointers.

a list of pointers to the C data allocated in building its C representation. This storage, though allocated in the C heap, must be deallocated (with `free`) by the programmer when it is known to no longer be in use. The function `dataCtoML` (the inverse to `dataMLtoC`) converts the C data structure at the address of its `caddr` parameter to a `cdata` value; the `ctype` parameter again guides the conversion.

The ML `sizeOf` function computes the size (in bytes) that an instance of its `ctype` parameter occupies in C (*cf.* C's `sizeof` operator).

The conversion and sizeof functions are useful for constructing and examining C data that must persist in a fixed location over a period of multiple calls to C. For example, an alternate registration of `create_window` (§3.2) registers its second parameter as having type `CaddrT` instead of the type `CptrT CpointT`. Then, the second parameter to `createWindow` could for example be the `NULL caddr` value or the `caddr` value produced by an application of `(dataMLtoC CpointT)` to a `cdata` instance of a `CpointT`. The `caddr` would then be a handle for the initialization structure which will now persist until explicitly deallocated.

4 Implementation

This section describes two key mechanisms at the core of our interface's implementation in version 108.5 of SML/NJ: *Byte-code type descriptors* guide the necessary data conversions and *runtime code generation* effectively augments an ML function's closure with an additional entry point that conforms to the calling convention of the target C compiler.

4.1 Interface Type Descriptors

The ML portion of the interface compiles `ctype` values into *type descriptors.* Type descriptors are byte-code strings that contain the necessary type, size, and alignment information for a given C type. Type descriptors and the `cdata` they describe flow across the interface. The C portion of the interface interprets type descriptors in conjunction with `cdata` values to create instances of C data in the C heap. Conversely, type descriptors guide the conversion of C data to ML `cdata` values.

The byte-code syntax consists of characters that denote a particular C type (*e.g.* `I` for integer) followed by zero or more bytes of type-dependent size information. For example, the type descriptor for the `ctype`

```
CstructT [CcharT, CintT]
```
(9)

in byte-code string format[12] is:

```
"<8C1P3I4>"
```
(10)

This describes an eight byte structure (`<8···>`) that contains a one-byte character (`C1`), three bytes of padding (`P3`), and a four-byte integer (`I4`). The size of the structure is made explicit in the type descriptor to permit the interface's interpreter to allocate space for the structure without having to inspect the byte-codes for the structure's member fields. Type-descriptor compilation inserts the padding bytes to align the structure's integer field on a word boundary. The values supplied in the `CC_INFO` structure for interface instantiation (§3.1) govern type-descriptor compilation. Other types have similar byte-code representations and compilation transformations; we do not further describe them here.

4.2 Function Pointers

We devised a dynamic mechanism to convert ML functions (closures) to C function pointers. It is similar to the dynamic generation, in the C heap, of C functions in the *esh* Scheme/C interface [13]. Our mechanism however supports copying garbage collection (as well as *esh*'s conservative collection). Such a mechanism is needed to support C function pointers without reengineering the SML/NJ compiler's backend; calling convention conversion occurs completely in the C portion of the interface. Given an ML closure f, this mechanism creates an additional entry point for f that adheres to the C compiler's calling convention. More specifically, when handed a closure, the interface creates a three-component *bundle* that contains the closure address, a type descriptor for the closure, and a code block that can find and invoke the closure. Figure 6

[12] Our implementation uses byte values instead of the (readable) ASCII codes used in this example.

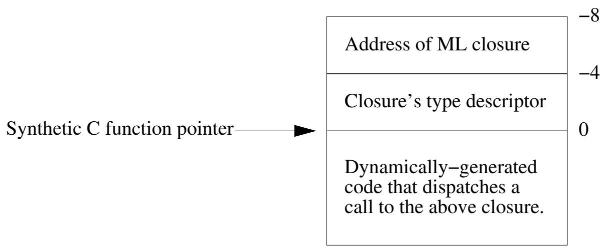

Figure 6: Conversion of ML closures to C function pointers. The entry code extracts the closure address from its header, converts parameters, calls the closure, converts the closure's return value, and returns this result to its C caller. Parameter and return value conversions are described by the pointer in the slot for the closure's type descriptor.

depicts such a bundle. A C function pointer to the ML closure is simply the address of the new bundle's code block.

A bundled type descriptor guides the conversion of C parameters to `cdata` values for the closure's argument as well as the conversion of the closure's eventual `cdata` value back to C.

The code block is constructed in the C heap from a closure address and a fresh copy of a relocatable assembly-language routine. This routine is C compiler and machine architecture specific; it must be supplied by our interface for every target C compiler. Relocation is achieved through simple byte copying. The routine must be relocatable in order to provide a unique function pointer for an ML closure—a generic dispatch routine does not work here because, by virtue of being generic, it cannot be coupled to a specific ML closure. More specifically, when handed the address p of an ML closure, the interface must produce a p' that "behaves like" a C function pointer; when applied from within C, p' must setup and perform a call to p. If p' were to point to a generic dispatch routine, this routine cannot know which ML closure (p in this case) to dispatch. The address p must therefore be accessible given only p'. We achieve this by placing p in a fresh memory block along with code for extracting p. This code block at p'—upon extraction of p—immediately passes p to a C function that sets up the call to p (*i.e.* builds p's `cdata` argument), calls p, and upon p's return finally converts p's result to C format.

A final implementation concern ensures that collection of garbage SML/NJ code objects [12] updates the closure addresses in bundles. Upon bundle construction, the location of the closure address in a bundle is registered as a root with the SML/NJ garbage collector. This suffices to correctly track movement of the closure; that is, when the garbage collector moves the closure, it also updates the closure's address in the bundle.

5 Future Work

Further work falls into one of two categories: design issues and implementation issues. We first discuss the design issues.

The copying approach of our design poses several problems. It is inefficient when large amounts of data must traverse the interface or when ML and C must share mutable data. Although the `caddr` type (§3.4) can be used to circumvent this problem, a design that avoids copies is desirable (*e.g.*, [13], §6). Such a design would require ML data representations to match those of the target C compilers as well as ML code generation that uses C calling conventions for ML functions (*cf.* §2.4). An approach that uses explicit programmer declaration of the ML data (and functions) that will be passed to C may prove to be a practical middle ground.

A further disadvantage of our design is its inability to specify cyclic C data. To support cycles, the `ctype` datatype (§3.3) must be modified to permit specification of such types and the C part of the interface would have to detect cycles when converting data between representations.

The remaining issues are due to the current implementation of the design; we believe that these issues can be resolved independent of the design. The implementation places restrictions on ML closures passed to C as function pointers (§4.2). In particular, the current interface implementation only properly handles functions with functional control flow; functions that raise escaping exceptions or throw to an SML/NJ continuation captured prior to the function's application may leave unreachable C data and state on the C stack. Analyses to determine the set of expressions that may escape from an ML expression [15, 7] (and hence from ML functions called by C) may provide some solution to this problem. Furthermore, an ML function—once passed to C—will persist until the program terminates. This conservative behavior is necessary since C may retain a pointer to an ML closure f after ML has abandoned all of its pointers to f. One must therefore assume that C may access f in the remainder of the program's lifetime. In practice, persistent closures are not problematic since one typically passes only a small set of functions to C; however, this problem requires solution in order to guarantee the absence of space leaks.

Our implementation currently requires C function arguments and return values to fit in a machine word. This is not a severe restriction since pointers fit in a machine word and one can introduce a level of indirection. Removing this restriction requires rewriting the mechanism that calls an interfaced C function in architecture-specific assembly language (word size arguments can be passed, using C, in a machine independent manner).

6 Related Work

ML runtime systems [9, 12], as well as systems for other mostly-functional languages (*e.g.* Lisp, Scheme, Haskell), provide low-level access to C functions. Such raw interfaces suffice to implement a language's run-time support (I/O, *etc.*). In a raw interface, a called C function is passed a set of untranslated arguments; that is, arguments are not automatically converted from the native system's representation to C's representation. This requires programmers to understand a compiler's internal representations and to explicitly write C code to convert representations. Garbage collection also inhibits programming with a raw interface since native and foreign storage must be explicitly allocated and freed. Direct linkage with C function libraries is also not possible since explicit representation conversions are required. The automatic interface described in this paper requires programmers to only specify the type of a C function from within ML; no additional C code need be written.

Rose and Muller's Scheme environment [13] called (*esh*) provides a high-level interface to C that is closely related to our interface for SML/NJ. The *esh* system provides basic Scheme objects that map directly to C types. Objects carry tags that identify their (C) type. The `cdata` constructors (§3.4) in essence supply the "tags" in our system. Rose and Muller's system never relocates storage; it is not clear how well they handle long running programs that allocate much data. Non-copying GC strategies greatly simplify the design of foreign function interfaces. *esh* furthermore inherits C's data representations for all basic types. Scheme functions are used to lay out C data "in place" in *esh*'s shared Scheme/C heap. Our interface adopts a copying strategy since it must convert data representations and handle copying garbage collection. To pass Scheme closures to C, *esh* dynamically generates C functions on the C heap; this is akin to the approach we use (§4.2).

The *Talk* variant of Lisp provides a C++ class interface [6]. The interface is automatic—when given a C++ class declaration it generates Talk stub functions. As described, our interface requires explicit registration of the types of interfaced C functions, but it is straightforward to have our interface automatically register the functions named in a C header file, for example. Since Talk interacts at the C++ class level, it can only create complex data via class constructors. One cannot, in general call C functions with arbitrary C data using the Talk/C++ interface. For passing Lisp functions to C++ as function pointers, Talk supplies a new defining form; functions defined with this form are compiled with the C++ calling convention instead of the default Talk convention. However, functions defined with this form may only be called from C++ and not from Talk. In contrast, our interface supports function pointers in general.

7 Conclusion

We have designed an interface to C for ML and have implemented it in the SML/NJ compiler. The interface makes explicit copies of a foreign function's parameters and return results in the C heap. This copying strategy provides solution to problems caused by incompatible data representations, calling conventions, and storage managers. Our ML interface to C is practical; many non-trivial C functions have been interfaced to date.

Acknowledgements

Thanks to David MacQueen for discussions on inter-language interfaces. John Reppy explained subtleties of the SML/NJ runtime system. Sheng Liang built the interface to Win32 for SML/NJ using the C interface; his suggestions improved the interface. Chris Fraser and Sheng Liang provided valuable comments on an early version of this paper.

References

[1] A. W. Appel. A runtime system. *Lisp and Symbolic Computation*, 3:343–380, 1990.

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

[3] A. W. Appel and D. B. MacQueen. A Standard ML compiler. *Functional Programming Languages and Computer Architecture*, 274:301–324, 1987.

[4] A. W. Appel and Z. Shao. Callee-save registers in continuation-passing style. *Lisp and Symbolic Computation*, 5:191–221, 1992.

[5] E. Biagioni, R. Harper, P. Lee, and B. G. Milnes. Signatures for a protocol stack: A systems application of Standard ML. In *Lisp and Functional Programming*, pages 55–64. Association for Computing Machinery, June 1994.

[6] H. Davis, P. Parquier, and N. Séniak. Sweet harmony: The Talk/C++ connection. In *Lisp and Functional Programming*, pages 121–127. Association for Computing Machinery, June 1994.

[7] J. C. Guzmán and A. Suárez. An extended type system for exceptions. In *Workshop on ML and it Applications*, pages 127–135. ACM SIGPLAN, INRIA, June 1994.

[8] S. P. Harbison and G. L. Steele Jr. *C: A Reference Manual*. Prentice Hall, 3rd edition, 1991.

[9] X. Leroy. The Caml Light system, documentation, and user's guide. . `http://pauillac.inria.fr/caml/man-caml/`, July 1995.

[10] Microsoft Press. *Win32 Programmer's Reference*, 1993.

[11] J. K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, 1994.

[12] J. H. Reppy. A high-performance garbage collector for Standard ML. Technical memorandum, AT&T Bell Laboratories, January 1994.

[13] J. R. Rose and H. Muller. Integrating the Scheme and C languages. In *Lisp and Functional Programming*, pages 247–259. Association for Computing Machinery, June 1992.

[14] G. L. Steele Jr. Rabbit: A compiler for Scheme. Technical Report AI-TR-474, MIT, Cambridge, MA, 1978.

[15] K. Yi. Compile-time detection of uncaught exceptions for Standard ML programs. In *Lecture Notes in Computer Science, Proceedings of the First International Static Analysis Symposium*, volume 864. Springer-Verlag, 1994.