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

No-Longer-Foreign: Teaching an ML compiler to speak C "natively"

Matthias Blume [1]

Lucent Technologies, Bell Laboratories

Abstract

We present a new foreign-function interface for SML/NJ. It is based on the idea of *data-level interoperability*—the ability of ML programs to inspect as well as manipulate C data structures directly.

The core component of this work is an encoding of the almost [2] complete C type system in ML types. The encoding makes extensive use of a "folklore" typing trick, taking advantage of ML's polymorphism, its type constructors, its abstraction mechanisms, and even functors. A small low-level component which deals with C `struct` and `union` declarations as well as program linkage is hidden from the programmer's eye by a simple program-generator tool that translates C declarations to corresponding ML glue code.

1 An example

Suppose you are an ML programmer who wants to link a program with some C routines. The following example (designed to demonstrate data-level interoperability rather than motivate the need for FFIs in the first place) there are two C functions: `input` reads a list of records from a file and `findmin` returns the record with the smallest `i` in a given list. The C library comes with a header file `ixdb.h` that describes this interface:

```
typedef struct record *list;
struct record { int i; double x; list next; };
extern list input (char *);
extern list findmin (list);
```

Our `ml-nlffigen` tool translates `ixdb.h` into an ML interface that corresponds nearly perfectly to the original C interface. Moreover, we hooked `ml-nlffigen` into the compilation manager CM [2] of SML/NJ [1] in such a

[2] Variable-argument functions are the only feature of the C type system that we do not handle very well yet.

[1] e-mail: blume@research.bell-labs.com

way that C header files like `ixdb.h` can be used as conveniently as any other source code.

We now show some ML code[3] that reads a list of records from some file and finds the `x` corresponding to the smallest `i`. The code produced by `ml-nlffigen` has the form of an ML functor (a module parameterized by the handle to the shared library object) which we must first instantiate:

```
structure IXDB = IxdbFn (val library = DynLink.openlib "ixdb.so")
```

Client code interacts with the C library by referring to structure `IXDB` as defined above. In addition to that, a client can use the predefined structure `C` which provides the basic encoding of C types and many common operations over them:

```
fun minx () = let
    val l = IXDB.fn_input (C.dupML "ixdb1")
    val m = IXDB.fn_findmin l
 in C.ml_double (C.get_double (IXDB.S_record.f_x (C.|*| m)))
end
```

In this code, `C.dupML` allocates a C character array of sufficient size and copies the given ML string into it, `IXDB.fn_input` represents function `input` that was declared in `ixdb.h`, `IXDB.fn_findmin` represents `findmin`, `C.|*|` dereferences a pointer, `IXDB.S_record.f_x` selects field `x` from a `struct record` object, `C.get_double` fetches the contents of a `double` object, and a concrete ML equivalent of type `real` is obtained from its abstract version by applying `C.ml_double`.

To add a final twist, suppose we also want to find the record with the largest `x`. The C interface does not provide such a function, but we shall not despair because we can write its equivalent directly in ML:

```
fun maxx l = let
  fun x l =
      C.ml_double (C.get_double (IXDB.S_record.f_x (C.|*| l)))
  fun loop (m, l) =
      if (C.isNull l) then m
      else loop (if x l > x m then l else m,
                 C.get_ptr (IXDB.S_record.f_next (C.|*| l)))
 in loop (l, l)
end
```

2 Introduction

Modern type-safe, higher-order languages such as ML have many advantages, but their usability often depends on how well existing code can be integrated into a project. Since existing code is usually not written in ML but rather in C or perhaps some other low-level language that has a C-like appearance, any serious implementation of a high-level languages must provide a foreign-function interface (FFI).

[3] To avoid too much detail we show ML code that corresponds to our actual implementation only in spirit but not in letter.

An FFI between ML and C must bridge not only the semantic gap between the languages but also mediate between different data formats and calling conventions. One fairly popular approach is to use *stub functions*: For the ML side to call a C function `f` of arbitrary type, it really invokes a helper routine `f_stub` that

- has also been coded in C but whose type is limited to a small set of types (possible a single type) which the ML compiler or its runtime system has been taught to handle
- translates ML data to C data to pass arguments from the ML side to `f`
- invokes `f` on behalf of its caller
- translates C data to ML data to pass results from `f` back to the ML side

This takes care of some of the data conversion issues, but the main problem of calling C from ML still remains to be addressed. Moreover, `f_stub` will not be able to deal with high-level abstract data, so in all likelihood there will be more stub code on the ML side to translate between layers of abstraction.

Much off-the-shelf C code is unaware of the existence of an ML client and, thus, has no a-priori need to understand or handle ML data structures. In this situation, the only reason for exposing the state of the ML world to the "C side" is to be able to perform *marshaling*. (Marshaling often involves heap allocation and can lead to invocations of the garbage collector.) Thus, if one can avoid C-side stubs, one can also avoid this exposure and make it much easier for an ML compiler's backend to generate instructions for calling C functions of any type directly.

2.1 Data-level interoperability

The stub-routine approach (like any marshaling scheme) also suffers from the problem that translating large data structures can be very expensive and might dominate the savings from calling a fast C routine. Since marshaling of data usually involves copying, useful properties such as sharing might also be lost in the process.

A solution to these problems is to rely on *data-level interoperability* [5] which avoids all marshaling operations as well as the need for C-side stubs. The high-level language is augmented with abstract types to represent low-level C data. ML code can then call C functions without help from any intermediary, passing arguments and receiving results directly.

One lesson from our work is that it is possible to encode the entire C type system using ML types, and that this can be done entirely within ML itself, requiring (almost) no support from the compiler. Based on this encoding, ML programs can traverse, inspect, modify, and create any C data structure *without marshaling*. Our FFI provides a glue code generator `ml-nlffigen` that automatically deals with "new" C types (`struct`- and `union`-declarations). It furthermore enables the ML compiler to generate correct calling sequences for any fixed-arity C function and puts the mechanisms for dynamic linkage in place.

The encoding of the C type system is provided as a CM library, and `ml-nlffigen` can be hooked up with CM in such a way that it gets invoked automatically whenever necessary. The resulting system has the following characteristics:

- The human programmer sees ML code only. (Some of this ML code will act much like C code, and it is remarkable that ML can be made to do that.)
- No knowledge of ML's or C's low-level representations is required.
- The types in an ML interface faithfully reflect the types in the original C interface represented by it.
- If the C interface changes, then these changes will be propagated automatically to the ML side. Any resulting inconsistencies show up as ordinary ML type errors and can be fixed as such.

ML aspires to being a "safe" language where, for example, no integer can be mistaken for a pointer. C is not a safe language, so using our interface (which embeds C into ML) makes ML unsafe as well. But since the very act of linking with C code already compromises ML's safety guarantees, this should not be seen as a problem. In practice, we believe that using our system will be more robust than many of the alternatives because it at least respects and enforces the types of the original C code within its ML clients.

3 Encoding the C type system

The encoding of C types in the ML type system is mostly based on the following "folklore" trick: Invariants for values of a given type T are enforced by making this type the representation type of a new abstract n-ary type constructor C (where $n > 0$) and adding cleverly chosen type constraints on its instantiations. The type arguments of C are called *phantom types.*

3.1 Array dimensions

As an example, let us consider static array dimensions. This is not a toy example but part of the larger picture because we will later need precise size information for *all* C types to be able to perform pointer arithmetic or array subscript correctly.

ML's built-in `array` type constructor does not express array size. In contrast, a C compiler will statically distinguish between types `(int[3])` and `(int[4])`. Still, it is not impossible to get the same effect in ML. We define a new type constructor `(`τ`,` δ`) arr` that also includes a *dimension* component δ, and arrays of different size must instantiate δ differently.

Of course, δ must be a type, not a number. Thus, we need a new type for every new dimension value. We should also make sure that no more than one type is used for any dimension value, so we set up an *infinite family* of types that once and for all time establishes a fixed mapping between its members and the non-negative integers.

3.1.1 An infinite family of types

Consider the following ML signature:

```
type dec and α dg0 and ... and α dg9 and α dim
```

Its type constructors can be seen as a "little language" for writing numbers in decimal notation.[4] For example, `dec dg1 dg3 dim` stands for 13_{10} and `dec dg1 dg0 dg2 dg4 dim` can be read as 1024_{10}.[5] This provides more than enough types to work with. To prevent the formation of values with unintended types such as `(real * int) dim` we restrict value constructors to these:

```
val dec: dec dim
val dg0 : α dim -> α dg0 dim
⋮
val dg9 : α dim -> α dg9 dim
```

For example, the expression `dg0 (dg2 (dg3 dec))` produces the `dim`-value corresponding to 320. The value for *any* non-negative integer $n = d_k d_{k-1} \dots d_{0_{10}}$ is constructed by dgd_0 ($\cdots$ (dgd_{k-1} (dgd_k dec)) $\cdots$). Induction on the number of applications of dg$d$ shows that all `dim`-values will have types of the form dec dgd_1 dgd_2 $\cdots$ dgd_k dim.

If we can rule out leading zeros, then (by uniqueness of decimal number representation) the mapping between non-negative integers and types of dimension values becomes bijective. Since we are working with strictly positive dimensions, we could replace the single `dec` constructor (representing 0) with nine new constructors representing 1 through 9. Our actual implementation avoids such extra constructors and makes types "smarter": an additional type parameter tracks "zeroness" and `dg0` is restricted to non-zero arguments.[6]

Let us now show an implementation of the above type family. The only type that requires non-trivial representation is `dim`, all other types are *phantom types* without meaningful values; we arbitrarily use the `unit` type for them:

```
structure Dim :> sig ... (* as before *) end = struct
  type dec = unit
  type α dg0 = unit and ... and α dg9 = unit
  type α dim = int
  val dec = 0
  local fun dg d n = 10 * n + d in
    val (dg0, dg1, ..., dg0) = (dg 0, dg 1, ..., dg 9)
  end
  fun toInt n = n
```

[4] Binary notation requires fewer constructors but is less convenient for human programmers.

[5] Type constructor application is left-associative in SML.

[6] The original typing convenience is provided by a type abbreviation. But notice that dimension types do not have to be spelled out very often because the ML compiler can usually infer them.

```
  end
```

The *opaque signature match* `:>` is crucial: it is ML's way of giving a fresh identity to each of the type constructors `dec`, `dg`*d*, and `dim`. Their representation types (and any type equalities between representation types) do not shine through. Thanks to polymorphism, we can extract the integer underlying any given `dim`-value using function `toInt`.[7] Because of how we implemented our value constructors, its numeric value will always be the one that is spelled out in decimal by the `dim`-value's type.

3.1.2 Dimension-carrying array types

To fill the dimension component of our `arr` type we use the δ component of the δ `dim` type that corresponds to the array's size:

```
  type (τ, δ) arr
  val create : δ Dim.dim -> τ -> (τ, δ) arr
```

Thus, the type of a 512-element integer array is `(int, dec dg5 dg1 dg2) arr`; an instance could be created by `create (dg2 (dg1 (dg5 dec))) 0`.

The expressiveness of these array types goes slightly beyond that of C's because one can write functions that are polymorphic in an array's dimension and even enforce simple constraints (e.g., "two arrays have the same length"). But we shall not overstate the usefulness of this because the extra power is still very limited.

3.2 Pointers, objects, and lvalues

Our implementation represents every C pointer by a simple address (using a sufficiently wide word type as its concrete representation).[8] Of course, exposing this representation directly would make programming very error-prone.

C's "`*`" type constructor tracks two facts about each address: the type of value pointed to and whether or not this value is to be considered mutable. We do the same in ML and dress up our low-level pointer representation with an abstract type:

```
  type ro and rw
  type (τ, ξ) ptr
```

Here, `ro` and `rw` are phantom types used to instantiate ξ. They indicate whether or not the object pointed to has been declared `const` in C. The instantiation of τ is more complicated; it describes the C type of the value the pointer points to.

Assignment to memory-stored objects need to know the address of that object. In C, however, one does not provide a pointer value on the left-hand side of the assignment. Instead, the left-hand side is one of a restricted class of expressions called *lvalues* and the compiler will implicitly insert the necessary

[7] The inverse `fromInt` cannot be added without breaking our type construction.

[8] Code samples where word size matters assume a 32-bit architecture. Different machines require different representation types, but our technique still works.

address-of operation for it. In ML, where the compiler will never insert any implicit operators, we essentially have no choice but to use explicit pointer values. In our implementation, we create the illusion of a distinction between objects and pointers by providing a separate type constructor `obj`, which (among other things) is used on the left-hand side of assignments. Internally, `ptr` and `obj` are the same and conversions between them are identity functions.

```
type (τ, ξ) ptr and (τ, ξ) obj
val |*| : (τ, ξ) ptr -> (τ, ξ) obj  (* dereference *)
val |&| : (τ, ξ) obj -> (τ, ξ) ptr  (* address-of  *)
```

C's conceptual "subtyping" relation governing constant and mutable objects is modeled by providing a polymorphic injection function (internally implemented by an identity):

```
val ro : (τ, ξ) obj -> (τ, ro) obj
```

3.3 Memory fetches and stores

Fetching from memory does not work the same way for all types because it depends on the size of the representation. In ML we cannot provide a polymorphic *fetch* function that takes objects of type `(τ, ξ) obj` to values of type τ because the representations of the values involved are not uniform.

On the other hand, C itself cannot fetch from arbitrary objects (arrays are the primary example), essentially distinguishing between *first-class* values which can be fetched and stored and other, *second-class* values. In ML, we can cover the whole range of C's first-class types with a relatively small, finite set of individual fetch- and store-operations: we only need to cover base types such as `int` or `double` as well as pointers.[9]

Fetch operations are polymorphic in the object's `const`-ness, store operations require `rw`. Fetching and storing of pointers is polymorphic in the pointer's target type because the underlying operations on address values are uniform.

```
type sint and ... and double   (* base types t *)

val get_t: (t, ξ) obj -> t           (* for base types t *)
val get_ptr: ((τ, κ) ptr, ξ) obj -> (τ, κ) ptr

val set_t: (t, rw) obj * t -> unit    (* for base types t *)
val set_ptr: ((τ, κ) ptr, rw) obj * (τ, κ) ptr -> unit
```

We can now state more precisely what the τ type parameter means: For types of *first-class* C values the parameter τ is instantiated to the (ML-side) abstract type of that value. For second-class values, however, there are no values of type τ, so τ is a true phantom type in this case.

[9] If ML had programmer-defined overloading, then these operations could be presented using a single, uniform-looking interface.

3.4 Arrays

As we have explained, there are no array values, only array objects. The phantom type constructor `(τ, δ) arr` works exactly as shown earlier: we use the types and values from the `Dim` module to statically specify the number of elements in each array.

In C, one can explain most operations over arrays using operations over pointers because in almost all contexts an array will *decay* into a pointer to its first element. In ML, we make this explicit by providing a function `decay` (which internally is yet another identity):

```
  val decay: ((τ, δ) arr, ξ) obj -> (τ, ξ) ptr
```

Array subscript could be explained in terms of pointer arithmetic (see below), but our implementation provides a separate function that—unlike C—performs a bounds check at runtime:

```
  val sub: ((τ, δ) arr, ξ) obj * int -> (τ, ξ) obj
```

3.5 Pointers, pointer arithmetic, and runtime type information

We would like to define an operation with the following signature for adding pointers and integers:

```
  val ptr_add: (τ, ξ) ptr * int -> (τ, ξ) ptr
```

Suppose we internally let some word type represent `(τ,ξ) ptr`. Incrementing such a pointer means adding the *size of the target type* to the address value. How can we communicate size information (which depends on how τ is instantiated) to the `ptr_add` operation?

3.5.1 Explicit type parameters

One approach to modeling "functions over types" such as C's `sizeof` operator is to use explicit passing of *runtime type information* (RTTI). In the simplest case we just need a single integer that specifies the number of bytes occupied by an object of the given type.

But using a single static type `t` for all RTTI is dangerous because pointer arithmetic would then have to be typed as:

```
  val ptr_add : t -> (τ, ξ) ptr * int -> (τ, ξ) ptr
```

The problem with this is that nothing would stop us from passing the size of one object and use it in an operation on another, differently-sized one. To prevent such misuse we give *static types to dynamic type values*! RTTI for type τ will have type `τ typ`. Individual operators such as `ptr_add` can then enforce a correct match-up:

```
  val ptr_add: τ typ -> (τ, ξ) ptr * int -> (τ, ξ) ptr
```

Our implementation provides RTTI values for all of C's base types and value constructors for all of C's type constructors:

```
  type τ typ
  val sint_typ : sint typ ...
  val ptr_typ : τ typ -> (τ, rw) ptr typ
  val arr_typ : τ typ * (δ, ζ) Dim.dim -> (τ, δ) arr typ
```

Internally, τ `typ` is just a synonym for `int`:

```
type τ typ = int
val sint_typ = 4 ...
fun ptr_typ _ = 4
fun arr_typ (s, d) = s * Dim.toInt d
```

Since types (in C-like code) are statically known, so can be their corresponding RTTI values. A modicum of cross-module inlining [3] will transport size constants from their definitions to wherever they are being used. This enables the ML compiler to generate code for pointer arithmetic that is just as efficient as its C counterpart.

Thus, we have the somewhat paradoxical situation that the ML compiler is unable to infer size information and, thus, forces the programmer to help out, but it does have enough information to stop the programmer from making mistakes in the process. In languages with programmable access to *intensional* type information, for example Haskell's *type classes* [8], it might be possible to hide explicit RTTI arguments, creating more of an illusion of automatic "size inference."

3.5.2 Keeping RTTI "behind the scenes"

If we are willing to sacrifice some of the low-level efficiency, then we can eliminate explicit type arguments even in the ML case. We change our concrete representation of objects and pointers so that addresses are paired up with their RTTI. But we must also change the representation of that RTTI itself since it is no longer sufficient to pass simple size constants. Instead, RTTI will have to be structured.

To see this, consider fetching from a pointer object. The object is represented as a pair consisting of the object's address and the stored value's RTTI, i.e., the RTTI of a pointer. Once we fetch from the object we get the address that *is* the pointer, and we must pair it up with RTTI *for the object the pointer points to.* Our only hope to recover the latter is to have it be part of the pointer's RTTI. This leads to the following implementation:

```
datatype tinfo = BASE of int | PTR of tinfo | ARR of tinfo * int
type τ typ = tinfo
val sint_typ = BASE 4 ...     (* base types *)
fun ptr_typ t = PTR t
fun arr_typ (t, d) = ARR (t, Dim.toInt d)
fun sizeof (BASE s) = s
  | sizeof (PTR _) = 4
  | sizeof (ARR (t, d)) = d * sizeof t
```

Here is the corresponding implementation for type constructors (τ, ξ) `obj` and (τ, ξ) `ptr`:

```
type (τ, ξ) obj = addr * τ typ
type (τ, ξ) ptr = addr * τ typ
fun fetch_ptr (a, PTR t) = (load_addr a, t)
  | fetch_ptr _ = raise Impossible
```

By reasoning about types, a compiler could prove that the `Impossible` case is truly impossible or that `sizeof(t)` can be reduced to a constant for any `t` of ground type. However, such reasoning is complex and unlikely to benefit "ordinary" ML code. Thus, there is no realistic hope for these optimizations to be implemented in real ML compilers. Aside from the obvious representational overhead, this is the reason why keeping type information behind the scenes is less efficient than explicit RTTI passing. Our implementation provides both the explicit and the implicit version of RTTI and lets the programmer decide which trade-off between performance and ease-of-use is best in each situation.

3.6 `void *`

We model `void*` as a separate ML type called `voidptr`. Since `voidptr` acts as a supertype of all `ptr` types, we provide the corresponding polymorphic injection function. A pointer "cast" takes us in the opposite direction—just as unsafely as in C, of course.[10] RTTI is passed to the cast function as a way of specifying the desired target type.

```
val ptr_inject : (τ, ξ) ptr -> voidptr
val ptr_cast : (τ, ξ) ptr typ -> voidptr -> (τ, ξ) ptr
```

3.7 Function pointers

Function pointers are first-class C values whose abstract ML-side type is ϕ `fptr` where ϕ will be instantiated to some `A->B`. Their low-level representation is a machine address. A polymorphic `call` instruction dispatches C function calls:

```
val call: (α -> β) fptr * α -> β
```

The exact sequence of machine instructions necessary to invoke a C function depends on how α and β are instantiated. We encapsulate this aspect into the corresponding RTTI. Here is how we would like to revise the definition of type `typ`:

```
datatype φ tinfo = (* ... as before *) | FPTR of addr -> φ
type τ typ = φ tinfo
```

Unfortunately, this code will not compile. The type abbreviation τ `typ` cannot silently drop the type parameter ϕ for ϕ `tinfo`. To make the design work, we either must add ϕ as another type parameter to `typ` and therefore also to `ptr` and `obj` (which would "infect" almost all types with seemingly gratuitous ϕs), or we must "cheat." Our implementation avoids the type argument ϕ and defines `FPTR` as:

```
... | FPTR of Unsafe.Object.object
```

Internally, the library then uses ML "casts" wherever necessary to make the types work out. Fortunately, thanks to the the public interface, this is in fact still safe: no "unsafe object" will ever be forced into a type other than its original type.[11]

[10] Notice that some uses of `void*` can be expressed safely using polymorphism in ML.

[11] A rigorous proof for this can be derived from the fact that a typing for the same imple-

3.8 Function arguments and results

The type of a function pointer is `(`α` -> `β`) fptr` where α will be instantiated to some tuple type whose elements correspond to the arguments of the C function and where β will be instantiated to the type of the function's result (which is always a first-class value).[12]

Arguments that are first-class C values use the by now familiar ML encoding. Since we model `struct`s and `union`s as second-class values, we represent them using `obj` types when they appear as function arguments. Function results of `struct`- or `union`-type are handled by taking an additional mutable object argument that the result is written into.

4 Handling `struct` and `union`

C programs usually declare their own "new" types using `struct` and `union`. Let us focus on `struct` types (`union` is handled in a very similar way) and discuss how ML can model them.

4.1 Fully defined `struct`*s*

It is tempting to view the mention of a C `struct` as a *generative* type declaration like ML's `datatype`. However, this is not quite correct. An ML compiler that encounters two syntactically identical instances of a generative declaration will construct two distinct types that are not considered equal by the type checker. This runs counter to how C's `struct` declarations work.

One way of modeling C in ML is to use a predefined infinite family of `struct` tags where each individual program selects some of the members of this family and chooses an abstract interface for the corresponding types. A `struct` declaration does not *create* a new type, it takes an *existing* type and *defines an interface* for it. The responsibility for setting up the ML code for this lies with `ml-nlffigen`.[13]

Let `s_node` be the tag type for some `struct node`. The phantom type describing (second-class) values is then `s_node su`, and objects holding such values can be accessed using the interface implemented by a structure `S_node`. Most of this interface consists of field access operators that correspond to C's "."-notation and map a `struct` object to the object that represents the selected field. Example:

```
struct node { const int i; struct node *next; };
```

becomes

```
type s_node = ... (* select tag from tag type family *)
```

mentation (but without casts) using the aforementioned cumbersome ϕ parameters exists.

[12] C functions "returning" `void` become ML functions returning `unit`.

[13] The infinite family of tag types is similar to the `dim`-type family. Both families are provided by our support library.

```
  val typ : s_node su typ
  val f_i : (s_node su, ξ) obj -> (sint, ro) obj
  val f_next : (s_node su, ξ) obj ->
                        ((s_node su, rw) ptr, ξ) obj
```

Notice how `const` qualifiers are properly taken into account by the types of field accessors.

4.2 Incomplete `struct`*s*

In C, a pointer type can act as a form of "abstract type" if its target is a so-called *incomplete type*, i.e., a `struct` that is only known by its tag but whose fields have not been declared. Unfortunately, there is no sufficiently close correspondence with ML's abstract types for the latter to model C's incomplete pointers. The problem is that the same C type can be abstract in one part of the program and concrete in another, but abstract and concrete version of the type must be considered equivalent where they meet.

A proper ML solution to this puzzle is based on parameterized modules ("functors") and handles everything from simple incomplete types, incomplete types that get "completed," and even mutual recursion among incomplete types (and their respective completions). Since we wanted to support as much of C as possible, `ml-nlffigen` actually implements all that. But the solution is rather complicated and handles just a small corner of the language, so we will not discuss its details here but ask the interested reader to consult the documentation of our implementation at `http://cm.bell-labs.com/cm/cs/what/smlnj/`.

5 Low-level implementation

5.1 Two-stage encoding of C types

The code that actually implements the encoding of C types defines a structure `C_Int`. A second structure called `C` is obtained from `C_Int` by applying a more restrictive signature match. We use the library mechanism of CM to hide `C_Int` from the ordinary programmer. The extensions to `C` contained in `C_Int` would invalidate many of the invariants that structure `C` was designed to guarantee. But some low-level code generated by `ml-nlffigen` must be able to access `C_Int` directly. The implementation is done entirely within ML, the ML compiler has no a-priori understanding of the C type system.

5.2 Raw memory access

We modified the SML/NJ compiler to provide primitive operations (*primops*) for fetching from and storing into raw memory. Our representation of memory addresses is simply a sufficiently wide word type. Memory access primops are provided for `char`-, `short`-, `int`-, and `long`-sized integral types, for pointers (addresses), and for single- as well as double-precision floating point numbers.

5.3 Representing first-class values

SML/NJ currently does not have the full variety of precisions for integral and floating-point types that a typical C compiler would provide. Therefore, the same ML type must often represent several different C types. For example, fetching a C `float` value (i.e., a 32-bit floating point number) from memory yields an ML `Real64.real`. Implicit promotions and truncations are built into the respective memory access operations.

The high-level interface makes types such as `float` and `double` distinct even though their representations are the same. Otherwise incompatible types like `(float,` ξ`) ptr` and `(double,` ξ`) ptr` would be considered equal and, e.g., size information for the two could be confused. Client programs must use a set of separately provided conversion functions to translate from abstract C types to concrete ML types and vice versa. These conversion functions exist only for typing reasons. On the implementation side they are identities.

5.4 Field access

Access to a `struct` field translates the `struct` address to the field address by adding the field's *offset*. Offsets are machine- and compiler-specific. The `ml-nlffigen` tool mainly consists of a C compiler's front end (implemented by SML/NJ's *CKIT* library), so it can easily calculate offset values which are then used to specialize a generic field-access function provided by structure `C_Int`.

5.5 Function calls

Implementing direct calls to C functions from ML code required somewhat more extensive changes to the ML compiler. To avoid the need for outright syntax changes, C calls were added as yet another new primop. However, some considerable "magic" was needed in its implementation.

5.5.1 C function prototypes and calling protocol

The code generator must know the prototype of any C function to be called. This prototype happens to be encoded in the corresponding ϕ `fptr` type but the compiler has no knowledge of this encoding. The trick is to code the prototype into the type of an otherwise unused argument of the `rawccall` primop. This primop is pro-forma polymorphic but any actual use must be monomorphically constrained. One of its arguments is the address of the function to be called, another one is a tuple of ML values representing the actual parameters of the C function. The C function's return value is then similarly represented by the return value of `rawccall`.

The value of the third (extra) argument to `rawccall` will be ignored at runtime, what's important is its type. We defined a "little language" expressed in ML types[14] that describes certain ML values which are used internally (at compile-time) by MLRISC [7] for describing C function prototypes.[15] In the process of translating instances of `rawccall`, the type gets decoded and a corresponding MLRISC value is formed. This enables the backend to generate correct code for the C call.

5.5.2 Efficient signal handling

When a SML/NJ program is interrupted by an asynchronous signal, then execution must first advance to the next *safe point* before an ML signal handler can be invoked. Low-level signal handlers (which are part of the C runtime system) record the arrival of a signal and code generated by the ML compiler checks for this condition at regular intervals. A popular technique that eliminates extra runtime overhead for signal polling is to make the *heap-limit check* do double duty: The C handler records the arrival of a signal by setting the current heap-limit to zero. This causes the next heap-limit check to fail, and subsequent tests (which are no longer on the critical path) can then distinguish between genuine heap overflows and signals.

But the heap-limit is often implemented as a register. Blindly setting this register to zero while anything but ML code is executing is dangerous. Therefore, setting up a C call from ML involves temporarily turning off this form of signal handling. In SML/NJ, this is done by setting the `inML`-flag to 0 before the call and back to 1 after its return. The old FFI avoids losing signals by branching into a special runtime routine after returning from the C call. The routine checks for interrupts that may have arrived while signal handling was suspended. This technique is safe but expensive.

Our new implementation avoids much of the runtime penalty because it does not need to check for pending signals explicitly and can fully rely on the next heap-limit check: Before performing a C call, a few instructions of in-line code first set the (new) `limitPtrMask` to the all-ones bit pattern and `inML` to 0. After the call returns, `inML` is restored to 1. A final instruction then atomically performs:

```
  limitPtr <- bitwise-and (limitPtr, limitPtrMask)
```

The low-level signal handler stores 0 into `limitPtr` (as before) if `inML` is set but also stores 0 into `limitPtrMask` *regardless* of the state of `inML`. This arrangement guarantees that any signal eventually causes `limitPtr` to be 0 no matter when it arrives. The atomicity of the bit operation is key to avoiding races.

[14] and spoken exclusively between `ml-nlffigen` and the SML/NJ compiler

[15] The type encoding was chosen in such a way that we were able to avoid any runtime penalty for passing the extra parameter.

5.6 Dynamic linking

Dynamic linking is currently done using an interface to `dlopen`. Thus, one can painlessly link with existing shared libraries and no longer needs to alter the runtime system in the process.

Unfortunately, libraries loaded using `dlopen` do not stay alive across heap exports. Therefore, our ML-side dynamic linkage module represents dynamically loaded libraries and addresses obtained from them as abstract handles and automatically re-validates them whenever necessary. The C encoding represents all global variables as ML "thunks" (functions taking `unit` as an argument and returning the actual value). Exported functions are represented by similarly "thunkified" function pointers, but the generated interface also contains wrapper functions to invoke them more conveniently.

6 Related work

Virtually all implementations of high-level languages provide some form of FFI, and it would be difficult to list even just a small fraction of them here. There are many IDL-based approaches where the programmer writes a specification of the interface and uses a special compiler to generate glue code on both the C- and the high-level language side. Examples include H/Direct [4] and Camlidl [10]. (Our approach also falls in here: the IDL is C, and C-side stub generation is trivial.)

Much closer in spirit as well as implementation is the work on data-level interoperability for Moby [5], although Moby takes a less ambitious approach to modeling the full C type system. On the implementation side, Moby's FFI takes advantage of the fact that the compiler's intermediate representation *BOL* has been specifically designed with data-level interoperability in mind. In contrast, we showed here that C types can also be modeled with only very limited compiler support, using the abstraction facilities of the high-level language.

The *phantom type* trick that we used so extensively has come up many times in the past, even in other FFI designs such as H/Direct [4] where it is used to model a subtyping relationship between COM interfaces. New, perhaps, is the extreme to which we have taken an old trick: modeling everything from size information and pointer arithmetic over run-time types and function prototypes to array dimensions and incomplete types.

7 Preliminary results and conclusions

The current implementation is fully operational on x86/Linux and on Sparc/Solaris. Work is under way to fill in the missing pieces for all other backends supported by SML/NJ. Benchmarking results are still very preliminary.

C function calls perform well as the following numbers will show. We looked at four different versions of the `Math` structure. These versions differ in how square root, sine, cosine, and arctangent are being implemented: 1.using the corresponding Pentium machine instructions (our baseline for comparison), 2. using the C library via the old FFI, 3. using the C library via the new FFI, and 4. using portable ML code. We compiled the relatively short but floating-point intensive `nucleic` benchmark using these `Math` implementations and ran the resulting code 100 times in succession on a lightly loaded 800 MHz Pentium III system running the Linux 2.2.14 kernel. These are the cumulative timing results (elapsed time in seconds):

machine: 2.64   old FFI: 3.75   new FFI: 2.95   ML: 5.50

Calling C- or assembly-code using either of the two mechanisms wins over the native ML solution, but call overhead eats up nearly half of the advantage in the case of the old FFI. The new FFI incurs less than one third of that penalty and could be even better had SML/NJ's cross-module inliner [11] been working. Most concrete operations over abstract C types are very simple, and inlining those is essential to performance. This becomes even more apparent when we look at data-level interoperability: We found that traversing a 16-level deep complete binary tree generated by a C program is almost 3 times slower in ML than in C while hand-inlining all operations brings the overhead to within 30%. These are the numbers that we shall expect once the cross-module inliner has been debugged.[16]

We conclude that we have already succeeded in provided an FFI that is faster and much easier to use than its predecessors. It is unique in the way it fully encodes C's type system within ML. Nearly everything a C programmer can do has a direct (although perhaps sometimes clumsy-looking) equivalent on the ML side.

Missing from our type encoding is a way of fully handling variable-argument functions such as C's `printf`, a shortcoming that we intend to address soon. We also chose not to encode `enum` types and simply use `int` in their place. Our implementation currently does not support *callbacks*—calls of ML functions from C. Callbacks require that the state of the ML world is accessible from the C side, so at some point we will probably add a second version of `rawccall` which would then save the ML state in a well-defined way, probably at the expense of being somewhat slower.[17]

[16] The remaining 30% are probably due to unrelated effects such as SML/NJ's habit of allocating stack frames on the heap. In a recent test, the Moby compiler (which also uses an MLRISC backend) achieved better performance than C on this simple benchmark [6].

[17] It is relatively easy to use ML functions as C callbacks if those callbacks receive some kind of "context information" that can be used for passing closures. Otherwise, however, involved techniques such as runtime code generation [9] are required.

References

[1] Andrew W. Appel and David B. MacQueen. Standard ML of New Jersey. In Martin Wirsing, editor, *3rd International Symp. on Prog. Lang. Implementation and Logic Programming*, pages 1–13, New York, August 1991. Springer-Verlag.

[2] Matthias Blume. CM: The SML/NJ compilation and library manager. Manual accompanying SML/NJ software, 2001.

[3] Matthias Blume and Andrew W. Appel. Lambda-splitting: A higher-order approach to cross-module optimizations. In *Proc. 1997 ACM SIGPLAN International Conference on Functional Programming (ICFP'97)*, pages 112–124. ACM Press, June 1997.

[4] S. Finne, D. Leijen, E. Meijer, and S. Peyton Jones. H/Direct: A binary foreign language interface for Haskell. In *Proc. 1998 ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, pages 153–162, September 1998.

[5] Kathleen Fisher, Riccardo Pucella, and John Reppy. Data-level interoperability. Bell Labs Technical Memorandum, April 2000.

[6] Kathleen Fisher, Riccardo Pucella, and John Reppy. A framework for interoperability. In *BABEL'01*, volume to appear, September 2001.

[7] Lal George. MLRISC: Customizable and reusable code generators. Technical report, Bell Laboratories, May 1997.

[8] Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip L. Wadler. Type classes in Haskell. *ACM Transactions on Programming Languages and Systems*, 18(2):109–138, March 1996.

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

[10] Xavier Leroy. CamlIDL user's manual. available from `http://caml.inria.fr/camlidl/`, March 1999.

[11] Stefan Monnier, Matthias Blume, and Zhong Shao. Cross-function inlining in FLINT. Technical Report YALEU/DCS/RR-1189, Dept. of Computer Science, Yale University, March 1999.