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

Matthias Blume
Lucent Technologies, Bell Labs

September 28, 2001

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 *complete*[1] 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 “new” C types (`struct` or `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. As a competent ML programmer you would not even have to know C to be able to use this new facility!

1 An example

Suppose you are an ML programmer and you need to link your program with some C routines. In the following simple example there are two C functions: `input` to read a list of records from a file and `findmin` to find the record with the smallest `i` in a given list. The implementors of the C library provided a header file `ixdb.h` to describe this interface:

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

Let us now write ML code to read a list of records from file `ixdb1` and find the `x` corresponding to the smallest `i`. We use the SML/NJ implementation [2] and start by setting up a CM library[2] `ixdb.cm`:

```
Library structure IXDBClient
is $/basis.cm
   $/c.cm
   ixdb.h : shell (target:ixdb.h.cm
                   ml-nlffigen ixdb.h)
   glue.sml
   client.sml
```

This specification declares:

- The library exports a structure `IXDBClient`.
- Libraries `$/basis.cm` (representing the Standard ML Basis library [13]) and `$/c.cm` (providing C types to ML programmers) are available.
- Library `ixdb.h.cm` will also be available after having been generated from `ixdb.h` using the shell command `ml-nlffigen` applied to `ixdb.h`.
- The library's implementation is provided by two ML source files: `glue.sml` and `client.sml`.

Some glue code is necessary to instantiate a functor exported from `ixdb.h.cm` which implements the ML interface to `ixdb.h`. Its argument describes how dynamic linkage is to be performed. If we assume that the library's name is `ixdb.so`, then `glue.sml` could look like this:

```
structure IXDB = Ixdb.IxdbFn
   (val library = DynLinkage.openlib
     { name = "ixdb.so",
       lazy = true, global = true })
```

The actual client code now uses structure `IXDB` as defined above and several modules, including structures `C` (the encoding of C types) and `ZString` (routines for dealing with C's NUL-terminated strings), both exported from `$/c.cm`:

```
structure IXDBClient = struct
  fun minx () = let
    val l = IXDB.fn_input
       (valOf (ZString.dupML "ixdb1"))
    val m = IXDB.fn_findmin l
  in C.Cvt.ml_double
      (C.Get.double
```

[1] well, almost

[2] CM is the compilation manager [4] of SML/NJ.

```
          (IXDB.S_record.f_x
            (C.Ptr.|*| m)))
    end
  end
```

Here is the key to this code:

- `ZString.dupML` allocates a C character array of sufficient size and copies the given ML string into it; the result is an option because allocation might fail—hence the use of `valOf`.[3]
- `IXDB.fn_input` represents the C function `input` that was declared in `ixdb.h`.
- `IXDB.fn_findmin` represents `findmin`.
- `C.Ptr.|*|` dereferences a pointer, yielding object it was pointing to.
- `IXDB.S_record.f_x` selects field `x` from a `struct record` object as declared in `ixdb.h`.
- `C.Get.double` fetches the contents of a `double` object.
- `C.Cvt.ml_double` converts a `double` value from the abstract C version to its concrete ML equivalent of type `real`.

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

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

2 Introduction

When faced with the decision of whether or not to implement a major piece of software in a modern, type-safe, higher-order language such as ML, it often comes down to how well existing code can be integrated into the project. But 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. Therefore, serious implementations of high-level languages must provide a foreign-function interface (FFI).

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 in order to pass arguments from the ML side to `f`
- invokes `f` on behalf of its caller
- translates C data to ML data in order to pass results from `f` back to the ML side

This takes care of some of the data conversion problems, but even calling `f_stub` from ML might be difficult. For exmple, SML/NJ's old FFI needs carefully written assembly glue for this. 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.

Having stub code on both sides of the language boundary is unsatisfactory for at least two reasons:

- It adds too many intermediate steps that slow down foreign-function invocation.
- It takes apart into two phases the translation from high-level ML data to C data. Combining these two phases can potentially be made more efficient.

Most C code "out there" that an ML program might want to call does not know anything about ML. Therefore, there is no a-priori need to make the ML side of the world visible during a foreign call. But C-side stubs that wrap C results into ML values might need to allocate from the ML heap. This means that a considerable portion of the ML state must be available on the C side, that the garbage collector might be invoked, and that the ML side might find its state altered after the foreign call returns. Thus, avoiding C-side stubs would significantly simplify the protocol for cross-language calls.

2.1 Data-level interoperability

Another source of inefficiencies in the stub-routine approach (and any *marshalling* scheme) is that the translation of large data structures can be very expensive. Its overhead can completely dominate any potential savings from calling a fast C routine if the ML side is interested in no more than a tiny portion of the result. Since marshalling of data usually involves copying, useful properties such as sharing might also get lost in the process.

[3]`ZString` is provided merely as a convenience. It is implemented on top of what structure `C` has to offer.

An approach that suffers from none of these problems, avoiding all marshalling operations as well as the need for C-side stubs, is to rely on *data-level interoperability* [7]. The idea here is that 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.

As we will see, one can encode the entire C type system using ML types. In fact, this can be done entirely within ML, i.e., with (almost) no support from the compiler. Based on this encoding, ML programs can traverse, inspect, modify, and create any C data structure *without marshalling.* The FFI that we describe here also 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[4] 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. (In theory, it is not even necessary for a capable ML programmer to know C!)
- No knowledge of low-level data representations is required. This includes the representations of C- as well as those of ML-data structures.
- 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.

Of course, by providing a C-like type system embedded within the ML type system, we get programs that are no more safe than C programs (in those parts that use the new facility). However, we believe that this in itself does not sacrifice safety any more than the act of linking with C code itself. In fact, the new approach might be considered *safer* than other foreign-function interfaces because it at least respects the types of the original C interface without going through some completely typeless stub layer. (Systems where all stub code is generated automatically from some typed interface-definition language are, of course, just as safe.)

[4]Variable-argument functions are the only major feature of the C type system that we do not handle.

3 Encoding the C type system

The encoding of C's types in the ML type system is mostly based on the "folklore" trick that expresses constraints on values of a given type T by making this type the representation type of a new abstract n-ary type constructor C (where $n > 0$). Invariants concerning values of T can then often be expressed by cleverly-chosen type constraints on instantiations of C.

3.1 Array dimensions

As a warmup, let us consider in full detail a concrete example of how the technique works. Incidentally, this is not a toy example but rather an actual piece of the larger picture.

In ML, size is a runtime concept which is not apparent in an array's type. In contrast, a C compiler will statically distinguish between types `(int[3])` and `(int[4])`. But the same effect *can* be achieved in ML. Instead of using a unary type constructor of the form τ `array` that only talks about the array's element type τ, we could try and define an new type constructor `(`τ`, `δ`) darray` that also includes a *dimension* component δ. Arrays of different size will instantiate δ differently.

The ML type system requires δ to be instantiated to a type; it cannot be a number. Fortunately, we are not really interested in the actual numeric value of the array's dimension (at least not for now). We only require differently sized arrays to have different types.

As a first approximation we could simply make up a brand new type every time a new dimension value is needed. This would work, but it requires some unpleasant bookkeeping to avoid ending up with different types for the same dimension value (rendering incompatible what should be equivalent array types).

A better method is to set up once and for all time an *infinite family of types* together with a 1-1 mapping to non-negative integers.

3.1.1 An infinite family of types

Consider the following ML signature:

```
sig type bin
    type α dg0
    type α dg1
    type α dim
end
```

These type constructors can be seen as a "little language" for writing numbers in binary notation.[5] For example, `bin dg1 dg0 dim` would stand for 10_2 which is 2_{10} and `bin dg1 dg0 dg1 dg1 dim` can be read as 1011_2 or 11_{10}.

[5]Our actual implementation uses decimal number representation which has more type constructors but otherwise works the same way.

Clearly, what we have here is a sufficiently rich, infinite family of types. It provides enough room for a mapping from any non-negative integer to its own type. One remaining problem is that we cannot prevent anyone from forming unintended types such as `(real * int) dim`. However, we *can* prevent the formation of values with such unintended types by providing no more than a very limited set of constructors for values whose types fit the α `dim` scheme:

```
sig ...
    val bin: bin dim
    val dg0 : α dim -> α dg0 dim
    val dg1 : α dim -> α dg1 dim
end
```

For example, saying `dg0 (dg1 (dg1 bin))` produces the `dim`-value corresponding to 6. We can construct a value for *any* non-negative integer n by first representing n in binary, say $d_k d_{k-1} \ldots d_0$, and then invoking `dg`d_0 `(`$\cdots$ `(dg`d_{k-1} `(dg`d_k `bin))` $\cdots$`)`.

If the set of value constructors is restricted to the above, then we can prove in a straightforward way (by induction on the number of applications of `dg`d) that the only values that are *constructable* will have types of the form `bin dg`d_1 `dg`d_2 $\cdots$ `dg`d_k `dim`. The missing piece for a proof of a 1-1 mapping between non-negative integers and constructable `dim`-values is the well-known uniqueness of the binary number represention—except that such uniqueness requires the absence of *leading zeros*.

To prevent leading zeros we need to restrict the application of `dg0` to values whose type is not `bin dim`. Since negative constraints like this cannot be expressed directly in ML, we introduce a second type parameter to track "nonzeroness":

```
sig type zero and nonzero
    type bin and α dg0 and α dg1
    type (α, ζ) dim
    val bin: (bin, zero) dim
    val dg0: (α, nonzero) dim ->
                 (α dg0, nonzero) dim
    val dg1: (α, ζ) dim ->
                 (α dg1, nonzero) dim
end
```

Notice how `dg0` enforces its argument to be nonzero while `dg1` is polymorphic in ζ, indicating its "don't care" attitude.[6]

Finally, we must provide a concrete implementation of this signature. The only type that requires non-trivial representation is `(`α`,`ζ`) dim`; all other types are *phantom types* without meaningful values. It does not matter how one represents those; the `unit` type used here already contains one more value than strictly necessary:

```
structure Dim :> sig ... (* as before *)
    val toInt: (α, ζ) dim -> int
  end = struct
    type zero = unit
    type nonzero = unit
    type bin = unit
    type α dg0 = unit
    type α dg1 = unit
    type (α, ζ) dim = int
    val bin = 0
    fun dg0 d = 2 * d
    fun dg1 d = 2 * d + 1
    fun toInt d = d
  end
```

The *opaque signature match* `:>` is crucial: it gives a fresh identity to each of the type constructors `bin`, `dg0`, `dg1`, `dim`, `zero`, and `nonzero`. Their representation types (and any type equality between representation types) do not shine through.

Thanks to polymorphism, we can add a function `toInt` that returns the integer behind any given `dim`-value.

3.1.2 Dimension-carrying array types

To fill the dimension component of our `darray` type we use the δ component of the `(`δ`, `ζ`) dim` type that corresponds to the array's size:

```
open Dim
sig type (τ, δ) darray
    val create : (δ, ζ) dim ->
            τ -> (τ, δ) darray
end
```

Thus, the type of a 4-element integer array is `(int, bin dg1 dg0 dg0) darray`; an instance could be created as follows:

```
val four = dg0 (dg0 (dg1 bin))
val a = create four 0
```

Notice that it is not necessary to write out any of the types. The ML type checker will automatically infer the type of the array (and its dimension) correctly.[7]

3.1.3 Dimension polymorphism

We can now emulate C's monomorphic array types, but ML lets us do even more: It is possible to write code that is polymorphic in an array's size. For example, here is a function that takes two arrays that are statically known to be of *equal size*:

```
val dotproduct:
      (real, δ) darray * (real, δ) darray
   -> real
```

[6] A 1-1 mapping for *positive* integers is easier to achive: Simply don't provide a zero value such as our `bin` and start with `val one: bin dg1 dim`.

[7] Expressions constructing `dim`-values spell numbers with their "bits" reversed. One can fix this cosmetic problem by using continuation-passing style for value constructors. At the expense of making their individual types more complicated, their use ends up looking more "pleasant."

One must be careful not to overstate the usefulness of this because only fairly simple constraints are expressible. There is still no type that, for example, would force two otherwise arbitrary arrays to differ in size by exactly one.

3.2 Pointers, objects, and lvalues

Our implementation represents every C pointer by a simple address (using a sufficiently wide word type as the concrete representation). Of course, exposing this representation directly would make programming very error-prone. The C language itself (whose implementations typically use the same low-level representation) provides a type constructor "`*`" for pointers so that two facts about each address are being tracked: the type of value pointed to and whether or not this value is to be considered mutable.

Therefore, we dress up our low-level pointer representation with an abstract type that models C's behavior:

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

The types `ro` and `rw` are again 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 fully describes the (declared) C type of the value the pointer points to.

A pointer points to a value residing somewhere in memory. Such values are called *objects*. (Non-object values might reside in machine registers. They do not necessarily have an address and cannot be pointed to by a pointer.)

In C there is the related notion of *lvalues*—expressions that denote objects. C-like lvalues are a bit troublesome because they have the unusual (from an ML-point-of-view) property that they mean different things depending on the context they appear in: on the left-hand side of an assignment operator they refer to the object itself (i.e., its address), on the right-hand side they mutate into "rvalues" and refer to the value stored at that address.

ML values cannot mimic this behavior; the conversion from lvalue to rvalue (which is the operation of fetching from memory) must be made explicit. There are two (fairly equivalent) ways of going about this:

1. One could abandon the notion of lvalues and require a pointer to appear on the left-hand side of every assignment operator instead. This is the route taken by ML's own `ref` type, but C code (which relies on assignment much more than typical ML code does) would probably look very awkward.

2. To make assignments look more like those in C one could have an explicit notion of objects and object-valued expressions. Fetching from objects must be done explicitly. This is the route taken in our FFI design.

The representation of objects is really the same as that of pointers; the distinction is made for purely esthetic reasons. This means that both *address-of* (`|&|`) and *dereference* (`|*|`) are identities in low-level terms:

```
...
type (τ, ξ) ptr
type (τ, ξ) obj
...
val |*| : (τ, ξ) ptr -> (τ, ξ) obj
val |&| : (τ, ξ) obj -> (τ, ξ) ptr
```

In C there is a conceptual "subtyping" relation governing constant and mutable objects. A pointer to a mutable object can be passed into a context that accepts a pointer to a constant object without drawing a compiler warning, but the other way around requires a cast.[8]

In ML we model this subtyping by providing a polymorphic injection function (internally represented by an identity function) which turns any object into its corresponding read-only object:

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

3.3 Memory fetches and stores

Ideally we would like to have a polymorphic *fetch* function that takes any object of type `(τ, ξ) obj` and returns the value of type τ stored therein. This would mean that the instantiation of τ will be more than a phantom type—it would have to be an actual abstract type of some value.

Unfortunately, our *fetch* operation cannot be polymorphic in the ML sense. Remember that all concrete representations behind our abstract types are whatever the C compiler uses for representing its data. Fetching from an `unsigned char` object is not operationally the same as fetching from a `double` object. Therefore, parametric polymorphism does not work here.

We are saved by the fact that C itself cannot fetch from arbitrary objects. For example, there is no way to fetch the contents of an entire array. In essence, C distinguishes between *first-class* values (those that can be fetched and stored) and other, *second-class* values. It is possible to 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` and pointers.[9]

[8]This means that C's `const` qualifier indicates a *promise* rather than an *expectation* by the context. Otherwise the subtyping relationship would have to be the other way around.

[9]In a language that (unlike ML) provides programmer-definable overloading, these sets of operations could even be presented using a single, uniform-looking interface.

All fetch operations are polymorphic in the object's constness, all store operations require the object's constness to be `rw`. Either operation for pointers is polymorphic in the pointer's target type because fetching or storing an address always works the same way regardless of what happens to be stored at that address.

```
type sint (* signed int *)
type uint (* unsigned int *)
...
type float
type double

val get_sint:
  (sint, ξ) obj -> sint
...
val get_double:
  (double, ξ) obj -> double
val get_ptr:
  ((τ, κ) ptr, ξ) obj -> (τ, κ) ptr

val set_sint:
  (sint, rw) obj * sint -> unit
...
val set_double:
  (double, rw) obj * double -> unit
val set_ptr:
  ((τ, κ) ptr, rw) obj * (τ, κ) ptr
  -> unit
```

Coming back to the discussion of what the τ type parameter means, we can now say the following: 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.

3.4 Arrays

As we have explained, there are no array values, only array objects. The phantom type describing arrays uses our `Dim` module for dealing with dimensions. This makes it possible for the ML compiler to statically distinguish between arrays of different sizes but with equal element types—just like the C compiler would.

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

In some sense the most important operation on array objects is `decay`. In C, array objects decay to pointers to their respective first elements in almost all contexts. Once this has happened, operations over array objects can be explained in terms of operations over pointers. Our implementation still provides a separate array-subscript operator `sub` which, unlike in C, performs dynamic bounds-checking.

While `decay` is yet another identity function hidden behind fancy types, implementing `sub` is much more tricky. Array subscript (as well as pointer arithmetic) requires information about the size of individual elements. Thus, the polymorphism here is not really "parametric" because the low-level operation depends on what τ has been instantiated to.

We will discuss this problem and our solution to it next.

3.5 Pointers, pointer arithmetic, and runtime type information

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

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

Suppose we represent `(τ, ξ) ptr` as `Word32.word`. Saying

```
ptr_add (p: (sint, rw) ptr, i)
```

would then correspond to `p+i*4` while

```
ptr_add(p: (double, rw) ptr, i)
```

is `p+i*8`. "Magic" multipliers such as 4 and 8 are sizes of C types specific to the particular system. How can we communicate size information to the operation?

3.5.1 Explicit type parameters

One approach to modelling "functions over types" such as C's `sizeof` operator is to use explicit passing of runtime type information (RTTI). The information we need here is simply a single number representing the number of bytes occupied by an object of a given type.

Imagine `t` to be the ML type of RTTI values. Pointer arithmetic would then be typed as follows:

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

The problem with this is that nothing would prevent us from passing dynamic type information for one type and use it to perform pointer arithmetic on another. Therefore, to prevent such misuse we use *static typing for dynamic type values*! RTTI for type τ will have type `τ typ`. Individual operators such as `ptr_add` can then enforce a correct matchup:

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

We define type `τ typ` in a submodule `T`. The same module also carries predefined RTTI values for all of C's base types and value constructors for all of C's type constructors:

```
structure T :> sig
  type τ typ
  val sint : sint typ
  ...
  val double : double typ
  val ptr : τ typ -> (τ, rw) ptr typ
  val arr : τ typ * (δ, ζ) Dim.dim
```

```
        -> (τ, δ) arr typ
  end = struct
    type τ typ = int
    val sint = 4
    ...
    val double = 8
    fun ptr _ = 4
    fun arr (s, d) = s * Dim.toInt d
  end
```

Every C type has a fixed corresponding RTTI value, and when types are statically known, then so can be their corresponding values. A modicum of cross-module inlining [5] will transport size constants from the `T` module to wherever type information is being used. As a result it is possible for an ML compiler to generate pointer arithmetic code that is just as efficient as its C counterpart.

Thus, we have the somewhat paradoxical situation that our ML compiler is unable to provide size information all by itself and therefore 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* [9], it might be possible to remove the remaining burden on the programmer's part (namely the need for explicit RTTI arguments).

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 our ML solution. We change our concrete representation of objects and pointers to one where addresses are paired up with corresponding RTTI. We must also change the representation of that RTTI itself. It is no longer sufficient to pass simple size constants. Instead, RTTI will have to be structured.

To see this, consider how we would implement fetching from a pointer object. The object is represented as a pair consisting of the object's address and its RTTI—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*. Where does the latter come from? Our only hope is to recover it from the RTTI of the pointer itself. This leads to the following design:

```
  structure T :> sig
    type τ typ
    val sint : sint typ
    ...
    val double : double typ
    val ptr : τ typ -> (τ, rw) ptr typ
    val arr : τ typ * (δ, ζ) dim
        -> (τ, δ) arr typ
    val sizeof : τ typ -> int
  end
  = struct
    datatype tinfo =
        BASE of int
      | PTR of tinfo
      | ARR of tinfo * int
    type τ typ = tinfo
    val sint = BASE 4
    ...
    val double = BASE 8
    fun ptr t = PTR t
    fun arr (t, d) = ARR (t, Dim.toInt d)
    fun sizeof (BASE s) = s
      | sizeof (PTR _) = 4
      | sizeof (ARR (t, d)) = d * sizeof t
  end
```

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

```
  structure C :> sig
    type (τ, ξ) obj
    type (τ, ξ) ptr
    ...
    val fetch_ptr :
      ((τ, κ) ptr, ξ) obj -> (τ, κ) ptr
    ...
  end = struct
    type (τ, ξ) obj = addr * τ T.typ
    type (τ, ξ) ptr = addr * τ T.typ
    ...
    fun fetch_ptr (a, T.PTR t) =
        (load_addr a, t)
      | fetch_ptr _ = raise Impossible
    ...
  end
```

Notice that one can prove based on reasoning about types that the second clause of `fetch_ptr` is truly impossible. However, there is no realistic hope that an ML compiler would actually perform this type of reasoning and eliminate the runtime check.

Another case of a theoretically possible optimization that is unlikely to be found in real compilers is the following: There is a 1-1 mapping between RTTI values and static types. Therefore, the compiler could in principle reduce any monomorphic call of `sizeof` to a constant. In practice, however, the compiler will not understand the nature of the mapping and therefore generate code to calculate sizes at runtime.

Aside from the obvious representational overhead, these are the reasons why keeping type information behind the scenes is going to be less efficient than explicit RTTI passing.

3.5.3 Providing both interfaces

There are tradeoffs (efficiency vs. ease-of-use) between the "light-weight" explicit-RTTI-passing interface and the "heavy-weight" RTTI-behind-the-scenes approach. Our implementation lets the programmer make the choice and provides both together with ways of mixing the two in the same program. To make this work, type constructors such as `(τ, ξ) ptr` or `(τ, ξ) obj` come in two flavors: one of them is implemented simply as a machine address which then requires explicit type parameters, the other is a pair of machine address and associated structured type information.

Since the light-weight representation does not require structured type information, there are also two kinds of runtime types: `τ typ` is the ML type for the structured variety while `τ size` provides a "size only" variety of RTTI.

3.6 void *

Our encoding has no direct equivalent for C's `void` since it is not truly a type and means different things in different contexts. The only use of `void` outside of function prototypes is in `void*`, C's generic pointer type. We model `void*` as a separate ML type called `voidptr`.

Conceptually, `voidptr` is a supertype of all `ptr` types. Therefore, we provide a corresponding polymorphic injection function:

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

Generic pointers would be useless if one could not also go the other way around. Of course, this is not a safe operation and can be used to subvert the type system. However, since we are linking with C code, this does not cause any *additional* problems; our ML code simply does directly what normally would have to be done just as unsafely on the C side.

ML "casts" could be provided by functions that are "too polymorphic," requiring the programmer to write an explicit type constraint. However, we have typed RTTI and can use it to good effect instead:

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

In fact, in the heavy-weight version of `ptr` where RTTI is part of the representation, `ptr_cast` needs explicit RTTI to be able to form the value. (One could imagine keeping RTTI for `voidptr` values to make casting safe. However, on the ML side we already have an even better mechanism for this: we can write code that is polymorphic in a pointer's target type τ. The use of `voidptr` can be restricted to cases where either the interface to existing C code explicitly requires it or where one truly wants to cheat the type checker.)

3.7 Function pointers

Function pointers are first-class C values. Their abstract ML-side type is `φ fptr` where ϕ will be instantiated to `A->B` for some `A` and `B`. While the low-level representation of function pointers is simply another machine address, it is very interesting to look at the kind of RTTI that we use.

We want to provide a polymorphic operator `call` that takes a function pointer along with suitable arguments and invokes the underlying C function with these arguments:

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

The underlying mechanism, i.e., the exact sequence of machine instructions necessary to dispatch a call to a C function, is highly dependent on how α and β are instantiated. Therefore, we encapsulate this by letting the RTTI for a function pointer carry the dispatch code. Here is a revised implementation of structure `T`:

```
structure T :> sig ... end = struct
  datatype φ tinfo =
      BASE of int
    | PTR of tinfo
    | ARR of tinfo * int
    | FPTR of addr -> φ
  ...
  type τ typ = φ tinfo
end
```

Unfortunately, this code will not compile; the type abbreviation `τ typ` cannot silently drop the type parameter ϕ for `φ tinfo` on the floor. To make this design work, we must add ϕ as an additional type parameter to `T.typ` and therefore also to `ptr` and `obj`:

```
type (τ, φ, ξ) obj
type (τ, φ, ξ) ptr
type φ fptr
structure T : sig
  type (τ, φ) typ
  ...
end
```

The rule for `(τ, φ, ξ) obj`, `(τ, φ, ξ) ptr`, and also `(τ, φ) T.typ` is that whenever `F fptr` occurs within the instantiation of τ, then ϕ must be instantiated to `F`. In all other cases, ϕ will be instantiated to `unit`. With properly set up interface types, it would be easy to prove that values violating this rule cannot be constructed.

However, in our actual implementation we decided to "cheat." We completely avoid the type argument ϕ and use the following clause for `FPTR`:

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

To make the types within our implementation work out, we then use SML/NJ's `Unsafe.cast` operation where needed. It turns out that the types in the public interface guarantee that all casts are "safe" after all: no value that was converted into an unsafe object will ever be converted to a value of type other than its original type.

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). Arguments that are first-class C values use their familiar ML type encoding. Since we have chosen to model `struct`s and `union`s as second-class values, we must use objects instead of values on the ML side to pass them as arguments. As we will see later, the phantom type for, say, `struct node` is `s_node su`. A function argument of type `struct node` will then be passed using a corresponding read-only object of type[10] `(s_node su, ro) obj`.

Function results of `struct`- or `union`-type are handled by taking an additional writable object argument that the result is written into.

4 Handling `struct` and `union`

So far we have seen the encoding of the *fixed part* of C's type system. But C programs usually declare their own "new" types using `struct` and `union`. Here we will focus on `struct`; unions are handled in a very similar way (namely as `struct`s where all fields have the same offset 0).

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. In C, however, separate mentions of `struct`s with the same tag within the same C program always refer to one and the same type.

One way of modeling this is to assume a predefined infinite family of `struct` tags where each individual C program can select some of the members of this family and choose 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.

To represent `struct` types in ML we use precisely this model. Consider some `struct node`. When this type is encountered by the `ml-nlffigen` glue code generator, it will first issue code to select a tag type by combining type constructors defined in structure `Tag`[11]:

```
structure ... : sig
  type s_node
  ...
end = struct
  local open Tag in
    type s_node = s t_n t_o t_d t_e
  end
  ...
end
```

The phantom type describing (second-class) `struct node` values is `s_node su`. An associated structure `S_node` then implements the interface for values of the abstract object type `(s_node su, ξ) obj` consisting of RTTI, run-time size information, and field access operators.

Field access operators correspond to C's "."-notation and map a given `struct` object to the object that describes the selected field. For example, suppose `struct node`'s definition is:

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

With this, `ml-nlffigen` would produce the following signature for structure `S_node`:

```
sig
  type tag = s_node
   val size : s_node su S.size
   val typ : s_node su T.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
end
```

The treatment of `const` qualifiers in field declarations is the following: The accessor for a constant field such as `i` ignores its argument's constness and always returns a read-only field object. The accessor for a mutable field such as `next` will make the constness of its result whatever the constness of the argument was. This produces the expected behavior: field access yields a constant object unless both the field itself as well as the object it is part of were declared mutable.

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.

We could model a pointer to an incomplete type declared in file A using some fresh, abstract ML type t. The problem with this is that should the complete type be revealed in another file B, then then the corresponding `ptr` type would not be equal to t.

What we need is as way of keeping the identity of a type hidden in one part of the program (the part that has to be written and compiled without the benefit of knowing the complete definition of the `struct`) while other parts can still know this identity. ML functors (parameterized modules) are the answer to this puzzle.

The `ml-nlffigen`-generated code for a C interface always has the form of a functor. At the minimum, it

[10]Object- and pointer arguments use their light-weight encoding without implicit RTTI.

[11]The structure has a type constructor for every character that can occur in a C identifier.

takes one parameter that is used to mediate dynamic linkage. It then takes one additional argument per incomplete pointer type in the interface.

For example, a reference to some `struct str *` will produce a functor argument of the form:

```
  structure I_S_str :
    POINTER_TO_INCOMPLETE_TYPE
```

Signature `POINTER_TO_INCOMPLETE_TYPE` (which our C library predefines) specifies a type ξ `iptr`. One can think of it as an abbreviation for `(s_str su, 'c) ptr`. However, this connection between the incomplete and the complete type is not spelled out yet because type `s_str` itself is not available at this point.

4.2.1 Completion

Client code knowing the full definition of `struct str` (by having access to structure `S_str`) can fill in the `I_S_str` functor argument by using a pre-defined "conversion" functor `PointerToCompleteType` which has been set up in such a way that it equates pointers to `struct`s described by its argument with the `iptr` type constructor of its result:

```
  structure I_S_str =
    PointerToCompleteType (S_str)
```

4.2.2 Mutual recursion among incomplete types

Structure `S_str` itself is the result of applying the functor representing the interface where `struct str` was defined. Thus, we would have a serious chicken-and-egg problem if we had one interface `a.c` that defines `struct a` containing a pointer to an incomplete `struct b` while `b.c` defines `struct b` containing a pointer to an incomplete `struct a`.

To account for this, `ml-nlffigen` will actually produce code with the following pattern. Interface `a.c` translates to an ML structure `A` containing not only the aforementioned functor but also a *partial* definition of structure `S_a` which is not dependent on that functor:

```
structure A = struct
  type s_a = ...
  structure S_a = struct
    type tag = s_a
    val size = ...
    val typ = ...
  end
  functor AFn
     (... structure I_S_b :
            POINTER_TO_INCOMPLETE_TYPE) =
  struct
    type s_a = s_a ...
    structure S_a = struct
      open S_a   (* inherit from partial version *)
      ...          (* provide missing pieces *)
    end
  end
  ...
end
```

Code for `b.c` will look very similar, with the rôles of `a` and `b` swapped.

Since `PointerToCompleteType` does not require the full structure `S_a` but can be applied to its partial version, we can break the dependency cycle:

```
  structure A' = A.AFn
    (... structure I_S_b =
           PointerToCompleteType (B.S_b))
  structure B' = B.BFn
    (... structure I_S_a =
           PointerToCompleteType (B.S_a))
```

4.2.3 Keeping types incomplete

Sometimes an incomplete type stays abstract. To provide a placeholder for the corresponding functor argument one can invoke functor `PointerToIncompleteType`. This functor does not need any arguments and causes its result's `iptr` type constructor to be fresh:

```
  structure I_S_str =
    PointerToIncompleteType ()
```

5 Low-level implementation

The implementation is based on the following main foundations:

1. a two-stage implementation of the C type encoding consisting of the client-visible high-level structure `C` and a corresponding low-level structure `C_Int` which is like the former but also contains a number of "unsound" extensions
2. an abstraction of raw memory
3. a representation of C's first-class values as ML values
4. a low-level operation for directly dispatching calls to C functions from within ML code

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 the SML/NJ compilation manager 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 desigend to guarantee. Only code generated by `ml-nlffigen` is supposed to use `C_Int` directly.

The implementation is done entirely within ML, the ML compiler has no a-priory 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 do double- (or triple-) duty by representing 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 types like `(float,` ξ`) ptr` on the one hand and `(double,` ξ`) ptr` on the other would be considered equal, and it would be possible to perform arithmetic on `double*` using RTTI for `float*`, something that is guaranteed to produce unintended results.

```
  ptr_add (T.ptr T.float)
          (p : (double, rw) ptr, 3)
```

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 simply identities.

5.4 Field access

Access to a `struct` field translates `struct` addresses to field addresses 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 calculate the required numbers easily. Field access functions are then generated by specializing a generic field-access function provided by structure `C_Int`. To give a concrete picture of this, assume we are using the lightweight representation of objects without implicit RTTI:

```
  structure C_Int :> sig ...
    val mk_field : int ->
        (σ su, ξ) obj -> (μ, κ) obj
  end = struct ...
    fun mk_field i a = a + i
  end
```

An `int` field `i` at offset 4 in `struct node` will cause `ml-nlffigen` to produce the following glue code for it:

```
  fun f_i (x: (s_node, ξ) obj) =
     C_Int.mk_field 4 x : (sint, ξ) obj
```

Clearly, `C_Int.mk_field` is rather dangerous, which is why access to it is restricted to trusted code (generated by `ml-nlffigen`) only.[12]

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 main hurdle is the fact that we somehow need to communicate to the code generator what the prototype of the C function to be called is. This prototype happens to be encoded in the corresponding ϕ `fptr` type. However, the compiler is not supposed to have any built-in knowledge of this encoding.

The C call primop operates at a level where the types of actual values to be passed to the C function are concrete ML values: integers, words, and reals of various precision. Since the mapping from C types to these ML representation types is many-to-one, the compiler cannot recover the C prototype from here. Therefore, we use the following trick:

The primop (called `rawccall`) has three arguments, the first being a word representing the function's address. The second argument is a tuple of ML values representing the arguments of the function (including the address of a return `struct` when necessary). Finally, the third argument is a dummy argument whose type describes the C function's prototype precisely enough to allow for correct code generation.

In the compiler's primitive environment[13], the type of `rawccall` is:

```
  val rawccall : Word32.word * α * β -> ξ
```

However, any actual use of `rawccall` must constrain this type monomorphically. In particular, α will be of the form $t_1 \times \cdots \times t_k$ and ξ will be some t_0 where all the t_i are either `Word32.word`, `Int32.int`, or `Real64.real`.

Generating code for C function calls uses the existing **c-calls** facility of MLRISC [8]. (The SML/NJ backend is based on MLRISC, but **c-calls** had been added to MLRISC to support other projects; it had not been used by SML/NJ before.) The API of this facility relies

[12]An experienced programmer could gain access to `C_Int` by referring to the CM library named `$/c-int.cm`. CM implements a mechanism for tracking access permissions, but there is still no enforcement.

[13]on a 32-bit machine

on a description—coded as an ML value—of the low-level prototype of the C function it is calling. Thus, to implement `rawccall`, we needed a way of constructing the correct prototype value so we can pass it to ML-RISC.

This is where β comes in. We defined in an ad-hoc way a "little language" expressed in ML types that, when decoded, can uniquely describe any prototype value. Here is a simple example. The C prototype

```
  double f (int, int *, double);
```

would be encoded as

```
  (unit * int * string * real -> real) list
```

As the reader might guess, `int` stands for `int`, `string` stands for (any) pointer, and `real` stands for `double`. The `unit` type has been added to the front of the tuple to avoid anomalies in cases where there are fewer than 2 arguments.

The final twist is the `list` type constructor. The semantics of `rawccall` have been set up in such a way that it eventually ignores its third argument. Nevertheless, since it has been declared, it is formally necessary to provide the third argument. By making its type a list type we can simply pass the empty list `nil` in any actual call. A `nil` argument obviously has no effect and can safely be dropped by the optimizer. As a result, the third-argument typing trick achieves its goal (communicating the C prototype to the backend) without causing any runtime overhead.

5.5.2 Frame pointer

SML/NJ allocates all its stack frames (continuations) on the heap [1, 3]. As a result, it normally does not use the usual system stack for function calls. C functions, on the other hand, do use the system stack.

In principle this would cause no major problems because the system stack exists even while pure ML code is running. If for nothing else, a properly set up stack is needed by the operating system when invoking signal handlers. In effect, from the point of view of a C program, ML code acts as a single non-recursive routine that does not perform any function calls—as long as there are no FFI calls.

Before the advent of the new FFI, it was literally true that the machine stack pointer would stay constant at all times during the execution of ML code. Therefore, the SML/NJ backend took a few shortcuts and treated the stack pointer as a *frame pointer*. In particular, on machines with limited register sets such as the `x86`, the "frame" pointed to by this frame pointer is being used to hold ML temporaries (pseudo-registers and stack-spills). Thus, since addressing of temporaries involved the stack pointer register, changing the value of this register when setting up a foreign call would break the compiler's assumptions.

To avoid this problem, MLRISC now implements a *virtual frame pointer* (`vfp`) and SML/NJ uses it to address the frame. This avoids wasting another machine register for a real frame pointer. During code generation, addressing modes relative to `vfp` get rewritten to addressing modes relative to the stack pointer—with offsets adjusted to account for temporary changes to the stack pointer register's value. (This is a fairly common optimization in many compilers which until now was not needed for SML/NJ itself. Its addition is expected to be beneficial not only for the FFI but also for other projects that use the MLRISC framework.)

5.5.3 Efficient signal handling

Like most systems with precise collectors, SML/NJ cannot be interrupted by asynchronous signals at arbitrary points. Instead, execution must advance to the next *safe point* before an ML signal handler can be invoked. Therefore, low-level signal handlers (which are part of the C runtime system) will simply record the arrival of a signal and code generated by the ML compiler will check for this condition at regular intervals.

A well-known technique that eliminates (additional) 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 arrivals of signals.

However, the heap-limit is often implemented as a register, and blindly setting this register to zero while anything but ML code is executing is unsafe. Therefore, setting up a C call from ML involves temporarily turning off this form of signal handling. SML/NJ's old FFI will first set the so-called `inML`-flag to 0 before dispatching the call, and after returning from the call it will invoke another run-time system routine to explicitly check for signals that might have arrived in the meantime. This technique, while safe, suffers from increased overhead since it amounts to doing two instead of one function call.

Our new implementation of C calls avoids much of the runtime penalty using the following trick:

- It first sets a new `limitPtrMask` to 0xFFFFFFFF (i.e., the all-ones bit pattern).
- Then, like the old FFI, it sets `inML` to 0.
- After returning from the C call, it sets `inML` back to 1.
- Finally, it atomically performs:

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

The runtime system's signal handler has been modified to always set `limitPtrMask` (which, just like `inML`, resides somewhere in main memory) to 0, regardless of the state of the `inML` flag. If `inML` is currently 1, then it also sets the actual heap limit pointer (`limitPtr`) to 0.

It is fairly straightforward to see that any signal eventually causes `limitPtr` to be 0 regardless of when it arrives. The C signal handler will either set it to zero directly or it will become zero as a result of the bit operation (whose atomicity is key to avoiding races). As a result, the new C call mechanism does not need to check for pending signals explicitly; it can rely on the next heap-limit check which is guaranteed to come soon.

5.6 Dynamic linking

Our implementation for x86/Linux provides an interface to `dlopen` via structure `DynLinkage`. This makes it possible to painlessly link with existing shared libraries without having to recompile or relink the runtime system.

One problem, however, is that dynamic linking does not work well (at least not automatically) with SML/NJ's export-heap facility. The ML garbage collector does not know about C data structures, so they will not be valid after an exported heap gets resurrected. Therefore, it is the programmer's responsibility that no C data structures are being alive when a heap is exported.[14]

Addresses of symbols exported by a shared libraries (corresponding to global C objects and global function pointers) are represented by abstract *handles* that do stay valid across heap exports; the `DynLinkage` package will re-validate them when necessary.

In our C encoding this is expressed by the fact that all global variables are represented by ML "thunks": functions taking `unit` as an argument and returning the actual value. Exported functions are represented by similarly "thunkified" function pointers. However, the ML interface will also contain ML functions that internally, when called, obtain the corresponding function pointer from its thunk and dispatch the call through that. Thus, there is some non-trivial overhead for such function calls which could be factored out of inner loops by first obtaining the function pointer value and then performing all calls through it.

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 [6] and Camlidl [11]. (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 [7]. Moby takes a less ambitious approach to modelling the full C type system. In particular, an older version of its FFI did not have type constructors corresponding to C's `*` and used a program generator to make separate abstract Moby types along with a full set of operations for every C pointer type in a given interface.[15] On the implementation side, Moby's FFI takes advantage of the fact that the language has been specifically designed with data-level interoperability in mind. Moreover, it has direct access to the intermediate language *BOL* of the compiler. In contrast, we showed here that C types can also be modelled with only very limited compiler support, using the abstraction facilities of the high-level language.

The *phantom type* trick we used so extensively here has come up many times in the past. It has even been used in other FFI designs, for example in H/Direct [6] to model a subtyping relationship between COM interfaces. New, perhaps, is the extreme to which we have taken an old trick: modelling 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 on x86/Linux is fully operational. Work is under way to provide the missing bits and pieces for all other backends supported by SML/NJ. Benchmarking results are still very preliminary.

Function calls dispatched via function pointers 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 then compiled the relatively short but floating-point intensive `nucleic` benchmark relative to these `Math` implementations and ran the resulting code 100 times in succession on a lightly loaded 800 MHz dual-processor Pentium III system running the Linux 2.2.14

[14]In some cases it might be possible to use SML/NJ's `CleanUp` facility to save and restore certain C data structures.

[15]Discussions with Kathleen Fisher and John Reppy have recently led to a fairly extensive exchange of technology. Moby adopted a type encoding similar to ours (although it does not go overboard like we do, ignoring `const` and other wrinkles of the C language).

kernel. These are the cumulative results, times are in seconds elapsed time:

machine	old FFI	new FFI	ML
2.64	3.75	2.95	5.50

Calling highly tuned C- or assembly-code using either of the two mechanisms turns out to be a win over the native ML solution. But call overhead eats up close to half of the advantage in the case of the old FFI while our new FFI incurs less than one third of a that penalty. It could be even better had SML/NJ's cross-module inlining facility [12] been working properly. Most operations over abstract C types are very simple: identities, projections, and address arithmetic. Inlining those is essential to performance, but at the moment it does not happen.

Lack of cross-module inlining has an even more serious impact on performance when we look at data-level interoperability: We found that traversing a 16-level deep complete binary tree that was generated by a C program to be almost 3 times slower than the same traversal expressed in C. After inlining all operations by hand, performance comes within 30% of C. These are the numbers we shall expect once the cross-module inliner has been debugged.[16]

In conclusion, we can say 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. Once we resolve the remaining performance problems there are few reasons left why an ML programmer should have to resort to writing C—even when the objective is to reuse existing C code.

Missing from our type encoding is a way of handling variable-argument functions such as C's `printf`. We also chose not to encode `enum` types and simply use `int` in their place. (Even in C itself the distinctions between `enum` and `int` are very weak.)

Lastly, our implementation currently does not support *callbacks*—calls of ML functions (which perhaps have been passed as arguments) from C. Callbacks require that the state of the ML world is accessible from the C side. We could add a second version of our `rawccall` primop which would save the ML state in a well-defined way, perhaps at the expense of being somewhat slower. Passing ML functions as C callbacks is even more tricky: it requires "dressing up" an ML closure value so that it looks like a C function pointer. One way of handling this is runtime code generation [10].[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 achieved better performance than C on this simple benchmark.

[17]Callbacks actually present a much more fundamental problem because of SML/NJ's first-class continuations. The usual solution is to ignore this problem, i.e., to simply hope that callback code does not exhibit non-local flow of control that crosses the boundary between ML and C.

8 Acknowledgments

John Reppy suggested the use of τ `size` as a lightweight variant of τ `typ`. He also designed the **c-calls** API within MLRISC and provided an implementation for the `x86` architecture. Lal George did most of the hard work in implementing virtual frame pointers and helped getting the new set of memory access primitives right. Many thanks also to Allen Leung for providing a LaTeXpackage for typesetting SML code.

References

[1] Andrew W. Appel. *Compiling with Continuations.* Cambridge University Press, 1992.

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

[3] Andrew W. Appel and Zhong Shao. Empirical and analytic study of stack versus heap cost for languages with closures. *J. Functional Programming*, 6(1):47–74, January 1996.

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

[5] 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.

[6] 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.

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

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

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

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

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

[12] 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.

[13] John Reppy and Emden Gansner. Standard ML Basis Library. (not yet in paper form), 1999.