The No Longer Foreign Function Interface

Jeff Vaughan

November 5, 2004

Contents

- Motivation
- Array Dimension Types
- Objects and Pointers
- Structs and Function Pointers

Practical languages need foreign function support.

- Multilanguage development.
- Support for legacy code.
- Interaction with the operating system.

The NLFFI is…

- an embedding of C types in SML types.
- an SML library for manipulating C data in SML.
- a data conversion library.
- a stub code generator.

SML/NJ data is not directly compatible with C libraries.

- SML/NJ's garbage collector uses least significant bits to distinguish references from unboxed data.
- Normal SML integers are 31 bits.
- 32 bit integers are a boxed type.
- In C, an int is a 32 bit word.
- None of these types are the same.

FFI's carry interesting semantic issues.

- There are bigger problems then representing primitives.
- What is the ML analog to …
  - struct?
  - enum?
  - union?
  - void*?
- As good type-conscious programmers, we don't want to think about void *, but as lazy library users, we need to.

OCaml pushes complexity into the C layer.

- To make a C call (easy):
  - Import the function:
    `external foo: int -> char = "foo"`
  - Call it: `let s = foo 3`
- Things aren't really this easy.
- The C function must take type `value`.
- Such values are manipulated with C macros.
- Therefore, you can only call into stub code expecting OCaml values.

Penn

MLton provides a simple FFI.

- To make a C call (also easy):
  - Import the function:
    `import foo: int -> char = "foo"`
  - Call it: `val s = foo 3`
- MLton's primitives match the C runtime and can interface directly with library code.
- Structs, enums, etc… are still problematic.
  - Code stubs in C can convert between primitives and structured data.
  - MLton allows for nasty pointer hacks too.

Penn

The old SML/NJ FFI was better, but still quite limited.

- Some support for structs, pointers, functions.
- On a C function call the FFI converted SML values to corresponding C values.
- New values are copied into and out of the C heap when crossing the SML/C boundary.
- Cyclic C types not permitted.
- Only word length parameters are permitted.

The NLFFI seeks to do more by doing less.

- All conversions explicit (though the NLFFI will automatically add some).
- SML programs can manipulate unconverted C values directly.

NLFFI supports an encoding of fixed length array types.

- How can we express `int[3]` ?
- We build a new array type with two parameters `(δ, τ) array`.
  - Parameter τ represents the element type.
  - Parameter δ is the array dimension.
- For now, we'll just make sure array of different lengths have different δs.

Encoding lengths requires types representing ints.

- Encoding binary numbers is easy:

```
type bin          type α dg1
type α dg0        type α dim
```

- We can represent 4 as
  `bin dg1 dg0 dg0 dim`

Array length representations need to sensible.

- The array dimension `(int * bool) dim` has no meaning.
- Providing a limited set of constructors stops values from inhabiting such types.

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

Array length representations need to be unique.

- Binary numbers are only unique if leading zeros are forbidden.
- Adding an extra type parameter to dim enforces this.
- (next slide)

Array length representations need to be unique. (II)

```
Signature BinSig = 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
```

We need values to inhabit the dimension type.

- To construct arrays, we need to provide values of type `bin … dim`.
- Features of these values:
  - Can only be built using appropriate constructors (requires opaque signature).
  - Can be constructed without excessive typing (compare to a unary encoding).
  - Values can implement at `toInt` function.
  - The NLFFI implementation uses decimal, which analogous, but requires more constructors.

We need values to inhabit the dimension type. (II)

```
structure Dim :> DimSig = 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
```

Penn

We have enough machinery to type an array constructor.

- The signature holds only one type and the constructor.

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

- We can build an array of four ints with

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

C programs classify data as pointers and objects.

- C code refers to data in two ways.
  - L-values or objects represent actual bits.
  - Pointers are the addresses of l-values.
- In C we explicitly convert between between objects and pointers using * and &.

C converts between objects and references automatically.

- For example:

```
int x = 3; /* store into &x */
printf(…, x); /* read from x */
```

- ML variables don't have this dual property. We'll need to explicitly convert.

The NLFFI distinguishes pointers from objects.

- Pointer and object types are

```
type (τ , ξ) ptr
type (τ , ξ) obj
```

- The parameters are as follow
  - Parameter τ represents the reference type.
  - Parameter ξ represents const-ness using

```
type ro and rw
```

Penn

Library functions convert pointers and objects.

- Converting between pointers and objects using `|&|` and `|*|`.

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

- Following C semantics, read/write types may be promoted to read only (const) types.

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

Penn

Get and set functions provide a way to assign to objects.

- Get and set only defined for primitive C types.
- Note that the types forbid setting a read only object.

```
type sint (* signed int *)
val get_sint: (sint, ξ) obj -> sint
val set_sint: (sint, rw) obj * sint ->
                                    unit
```

Manipulating C-arrays uses pointers and objects.

- The NLFFI supports bounds checked array access.

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

- An arrays can also be “decayed” to a pointer.

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

Implementing sub requires pointer arithmetic

- Pointer arithmetic requires knowledge of the size of objects.
- Directly passing this size to a `ptr_add` function is unsafe.
- Instead we use "light-weight" type constraints to ensure that a suitable size is passed.

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

Type constraints ensure safer pointer arithmetic.

```
structure T :> sig
  type τ typ
  val sint: sint typ
     ...
  val ptr: τ typ->(τ,rw) ptr typ
  val arr: τ typ * (δ, ζ) Dim.dim
end =
struct
  val sint = 4
     ...
  fun ptr _ = 4
  fun arr (t,d) = t * Dim.toInt(d)
end
```

NLFFI also supports a “heavy-weight” object representation.

- Heavy-weight objects are represented as an address $\times$ type pair.
- A `sizeof` function traverses the type value to find the size of the object.
- This could theoretically be optimized away (but not with current compilers).
- Using heavy-weight objects adds considerable overhead to computations.

Sometimes C code expects unsafe pointer casts.

- Casting to void * is easy to type:

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

- Using the typ type we can cast back:

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

- This is unsafe, but we lost safety when we linked with C.

Structures are represented using the module system.

- The encoding is generally straight forward.
- Multiple identical structure declarations refer to the same type.
  - This is accomplished using tags.
  - It’s messy and not on the agenda.

Structs are represented with an abstract type.

- Accessor functions provide access to individual fields.
- Field objects are returned with appropriate constness.
- (next slide)

Structs are represented with an abstract type. (II)

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

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 NLFFI supports first class function pointers.

- Function pointers are first class C values and are encoded as type φ `fptr`.
- Function calls are made with

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

- The code generator wraps all statically available functions.
- Programmers only need to use `call` when C code returns function pointers.

Penn

References

- NLFFI:
  http://ttic.uchicago.edu/~blume/papers/nlffi-entcs.pdf

  SML/NJ 110.50 distribution
- OCaml FFI:
  http://caml.inria.fr/ocaml/htmlman/manual032.html
- MLton FFI:
  http://mlton.org/doc/user-guide/Foreign_function_interface.html

Penn