

NLFFI
A new SML/NJ Foreign-Function Interface
(for SML/NJ version 110.46 and later)
User Manual

Matthias Blume
Toyota Technological Institute at Chicago

December 24, 2020

Contents

1	Introduction	2
2	The C Library	2
3	Translation conventions	2
3.1	External variables	2
3.2	Functions	3
3.3	Type definitions (typedef)	6
3.4	struct and union	7
3.5	Enumerations (enum)	10

1 Introduction

Introduce...

2 The C Library

The C library...

3 Translation conventions

The `ml-nlffigen` tool generates one ML structure for each exported C definition. In particular, there is one structure per external variable, function, `typedef`, `struct`, `union`, and `enum`. Each generated ML structure contains the ML type and values necessary to manipulate the corresponding C item.

3.1 External variables

An external C variable v of type t_C is represented by an ML structure G_v . This structure always contains a type t encoding t_C and a value obj' providing (“light-weight”) access to the memory location that v stands for in C. If t_C is *complete*, then G_v will also contain a value obj (the “heavy-weight” equivalent of obj') as well as value typ holding run-time type information corresponding to t_C (and t).

Details

type t is the type to be substituted for τ in $(\tau, \zeta) C.obj$ to yield the correct type for ML values representing C memory objects of type t_C (i.e., v ’s type). (This assumes a properly instantiated ζ based on whether or not the corresponding object was declared `const`.)

!val typ is the run-time type information corresponding to type t . The ML type of typ is $t C.T.typ$. This value is not present if t_C is *incomplete*.

!val obj is a function that returns the ML-side representative of the C object (i.e., the memory location) referred to by v . Depending on whether or not v was declared `const`, the type of obj is either `unit -> (t, C.ro) C.obj` or `unit -> (t, C.rw) C.obj`. The result of $obj()$ is “heavy-weight,” i.e., it implicitly carries run-time type information. This value is not present if t_C is *incomplete*.

val obj' is analogous to `val obj`, the only difference being that its result is “light-weight,” i.e., without run-time type information. The type of `val obj'` is either `unit -> (t, C.ro) C.obj` or `unit -> (t, C.rw) C.obj`.

(Elements that are subject to omission due to incompleteness of types are marked with an exclamation mark(!).)

Examples

C declaration	signature of ML-side representation
<code>extern int i;</code>	<pre> structure G_i : sig type t = C.sint val typ : t C.T.typ val obj : unit -> (t, C.rw) C.obj val obj' : unit -> (t, C.rw) C.obj' end </pre>
<code>extern const double d;</code>	<pre> structure G_d : sig type t = C.double val typ : t C.T.typ val obj : unit -> (t, C.ro) C.obj val obj' : unit -> (t, C.ro) C.obj' end </pre>
<code>extern struct str s1;</code> <code>/* str complete */</code>	<pre> structure G_s1 : sig type t = (S_str.tag C.su, rw) C.obj C.ptr val typ : t C.T.typ val obj : unit -> (t, C.rw) C.obj val obj' : unit -> (t, C.rw) C.obj' end </pre>
<code>extern struct istr s2;</code> <code>/* istr incomplete */</code>	<pre> structure G_s2 : sig type t = (ST_istr.tag C.su, rw) C.obj C.ptr val obj' : unit -> (t, C.rw) C.obj' end </pre>

3.2 Functions

An external C function f is represented by an ML structure F_f . Each such structure always contains at last three values: `typ`, `fptr`, and `f'`. Variable `typ` holds run-time type information regarding function pointers that share f 's prototype. The most important part of this information is the code that implements native C calling conventions for these functions. Variable `fptr` provides access to a C pointer to f . And `f'` is an ML function that dispatches a call of f (through `fptr`), using “light-weight” types for arguments and results. If the result type of f is *complete*, then F_f will also contain a function `f`, using “heavy-weight” argument- and result-types.

Details

`val typ` holds run-time type information for pointers to functions of the same prototype. The ML type of `typ` is $(A \rightarrow B) \text{ C.fptr } \text{C.T.typ}$ where A and B are types encoding f 's argument list and result type, respectively. A description of A and B is given below.

`val fptr` is a function that returns the (heavy-weight) function pointer to f . The type of `fptr` is $\text{unit} \rightarrow (A \rightarrow B) \text{ C.fptr}$. The encodings of argument- and result types in A and B is the same as the one used for `typ` (see below). Notice that although `fptr` is a heavy-weight value carrying run-time type information, pointer arguments within A or B still use the light-weight version!

`!val f` is an ML function that dispatches a call to f via `fptr`. For convenience, `f` has built-in conversions for arguments (from ML to C) and the result (from C to ML). For example, if f has an argument of type `double`, then `f` will take an argument of type `MLRep.Real.real` in its place and implicitly convert it to its C equivalent using `C.Cvt.c_double`. Similarly, if f returns an `unsigned int`, then `f` has a result type of `MLRep.Unsigned.word`. This is done for all types that have a conversion function in `C.Cvt`. Pointer values (as well as the object argument used for struct- or union-return values) are taken and returned in their heavy-weight versions. Function `f` will not be generated if the return type of f is incomplete.

val f' is the light-weight equivalent to f . a light-weight function. The main difference is that pointer- and object-values are passed and returned in their light-weight versions.

Type encoding rules for $(A \rightarrow B) \text{ C.fptr}$

A C function f 's prototype is encoded as an ML type $A \rightarrow B$. Calls of f from ML take an argument of type A and produce a result of type B .

- Type A is constructed from a sequence $\langle T_1, \dots, T_k \rangle$ of types. If that sequence is empty, then $A = \text{unit}$; if the sequence has only one element T_1 , then $A = T_1$. Otherwise A is a tuple type $T_1 * \dots * T_k$.
- If f 's result is neither a `struct` nor a `union`, then T_1 encodes the type of f 's first argument, T_2 that of the second, T_3 that of the third, and so on.
- If f 's result is some `struct` or some `union`, then T_1 will be $(\tau \text{ C.su}, \text{ C.rw}) \text{ C.obj}'$ with τ instantiated to the appropriate `struct`- or `union`-tag type. Moreover, we then also have $B = T_1$. T_2 encodes the type of f 's first argument, T_3 that of the second. (In general, T_{i+1} will encode the type of the i th argument of f in this case.)
- The encoding of the i th argument of f (T_i or T_{i+1} depending on f 's return type) is the light-weight ML equivalent of the C type of that argument.
- An argument of C `struct`- or `union`-type corresponds to $(\tau \text{ C.su}, \text{ C.ro}) \text{ C.obj}'$ with τ instantiated to the appropriate tag type.
- If f 's result type is `void`, then $B = \text{unit}$. If the result type is not a `struct`- or `union`-type, then B is the light-weight ML encoding of that type. Otherwise $B = T_1$ (see above).

Examples

C declaration	signature of ML-side representation
<code>void f1 (void);</code>	<pre> structure F_f1 : sig val typ : (unit -> unit) C.fptr C.T.typ val fptr : unit -> (unit -> unit) C.fptr val f : unit -> unit val f' : unit -> unit end </pre>
<code>int f2 (void);</code>	<pre> structure F_f2 : sig val typ : (C.sint -> unit) C.fptr C.T.typ val fptr : unit -> (C.sint -> unit) C.fptr val f : MLRep.Signed.int -> unit val f' : MLRep.Signed.int -> unit end </pre>
<code>void f3 (int);</code>	<pre> structure F_f3 : sig val typ : (unit -> C.sint) C.fptr C.T.typ val fptr : unit -> (unit -> C.sint) C.fptr val f : unit -> MLRep.Signed.int val f' : unit -> MLRep.Signed.int end </pre>
<code>void f4 (double, struct s*);</code>	<pre> structure F_f4 : sig val typ : (C.double * (ST_s.tag C.su, C.rw) C.obj C.ptr' -> unit) C.fptr C.T.typ val fptr : unit -> (C.double * (ST_s.tag C.su, C.rw) C.obj C.ptr' -> unit) C.fptr val f : MLRep.Real.real * (ST_s.tag C.su, C.rw) C.obj C.ptr -> unit val f' : MLRep.Real.real * (ST_s.tag C.su, C.rw) C.obj C.ptr' -> unit end </pre>

C declaration	signature of ML-side representation
<pre>struct s *f5 (float); /* s incomplete */</pre>	<pre>structure F_f5 : sig val typ : (C.float -> (ST_s.tag C.su, C.rw) C.obj C.ptr') C.fptr C.T.typ val fptr : unit -> (C.float -> (ST_s.tag C.su, C.rw) C.obj C.ptr') C.fptr val f' : MLRep.Real.real -> (ST_s.tag C.su, C.rw) C.obj C.ptr' end</pre>
<pre>struct t *f6 (float); /* t complete */</pre>	<pre>structure F_f6 : sig val typ : (C.float -> (S_t.tag C.su, C.rw) C.obj C.ptr') C.fptr C.T.typ val fptr : unit -> (C.float -> (S_t.tag C.su, C.rw) C.obj C.ptr') C.fptr val f : MLRep.Real.real -> (S_t.tag C.su, C.rw) C.obj C.ptr val f' : MLRep.Real.real -> (S_t.tag C.su, C.rw) C.obj C.ptr' end</pre>
<pre>struct t f7 (int, double); /* t complete */</pre>	<pre>structure F_f7 : sig val typ : ((S_t.tag C.su, C.rw) C.obj' * C.sint * C.double -> (S_t.tag C.su, C.rw) C.obj') C.fptr C.T.typ val fptr : unit -> ((S_t.tag C.su, C.rw) C.obj' * C.sint * C.double -> (S_t.tag C.su, C.rw) C.obj') C.fptr val f : (S_t.tag C.su, C.rw) C.obj * MLRep.Signed.int * MLRep.Real.real -> (S_t.tag C.su, C.rw) C.obj val f' : (S_t.tag C.su, C.rw) C.obj' * MLRep.Signed.int * MLRep.Real.real -> (S_t.tag C.su, C.rw) C.obj' end</pre>

3.3 Type definitions (typedef)

In C a `typedef` declaration associates a type name t with a type t_C . On the ML side, t is represented by an ML structure $T.t$. This structure contains a type abbreviation \mathfrak{t} for the ML encoding of t_C and, provided t_C is not *incomplete*, a value `typ` of type $\mathfrak{t} \rightarrow C.T.typ$ with run-time type information regarding t_C .

Examples

C declaration	signature of ML-side representation
<code>typedef int t1;</code>	<pre> structure T_t1 : sig type t = C.sint val typ : t C.T.typ end </pre>
<code>typedef struct s t2;</code> <code>/* s incomplete */</code>	<pre> structure T_t2 : sig type t = ST_s.tag C.su end </pre>
<code>typedef struct s *t3;</code> <code>/* s incomplete */</code>	<pre> structure T_t3 : sig type t = (ST_s.tag C.su, C.rw) C.obj C.ptr end </pre>
<code>typedef struct t t4;</code> <code>/* t complete */</code>	<pre> structure T_t4 : sig type t = ST_t.tag C.su val typ : t T.typ end </pre>

3.4 struct and union

The type identity of a named C struct (or union) is provided by a unique ML *tag* type. There is a 1-1 correspondence between C tag names t for structs on one side and ML tag types s_t on the other. An analogous correspondence exists between C tag names t for unions and ML tag types u_t . Notice that these correspondences are *independent of the actual declaration* of the C struct or union in question.

A C type of the form `struct t` is represented in ML as s_t C.su, a type of the form `union t` as u_t C.su. For example, this means that a heavy-weight non-constant memory object of C type `struct t` has ML type $(s_t$ C.su, C.rw) C.obj which can be abbreviated to $(s_t$ C.su, C.rw) C.obj.

All ML types $(\tau$ C.su, ζ) C.obj are originally completely abstract: they does not come with any operations that could be applied to their values. In C, the operations to be applied to a struct- or union-value is field selection. Field selection *does* depend on the actual C declaration, so it is ml-nlffigen’s job to generate a set of ML-side field-accessors that correspond to field-access operations in C.

Each field is represented by a function mapping a memory object of the struct- or union-type to an object of the respective field type. Let `int i;` and `const double d;` be fields of some struct `t` and let `tag` be the ML tag type corresponding to `t`. Here are the types of the (heavy-weight) access functions for `i` and `d`:

```

int i;           ~> val f_i : (tag C.su, 'c) C.obj -> (C.sint, 'c) C.obj
const double d; ~> val f_d : (tag C.su, 'c) C.obj -> (C.double, C.ro) C.obj

```

Notice how each field access function is polymorphic in the `const` property of the argument object. For fields declared `const`, the result always uses `C.ro` while for ordinary fields the argument’s type is used—reflecting the idea that a field is considered writable if it has not been declared `const` and, at the same time, the enclosing struct or union is writable.

Incomplete declarations

If the struct or union is incomplete (i.e., if only its tag t is known), then ml-nlffigen will merely generate an ML structure (called `ST.t` for struct and `UT.t` for union) with a single type `tag` that is an abbreviation for the library-defined type that corresponds to tag t .

Complete declarations

If the `struct` or `union` with tag t is complete, then `ml-nlffigen` will generate an ML structure (called `St` for `struct` and `Ut` for `union`) which contains at least:

`type tag` — an abbreviation for the library-defined type that corresponds to t

`val size` — a value representing information about the size of memory objects of this `struct`- or `union`-type. The ML type of `size` is `tag C.su C.S.size`.

`val typ` — a value representing run-time type information corresponding to this `struct`- or `union`-type. The ML type of `typ` is `tag C.su C.T.typ`.

Fields

In addition to `type tag`, `val size`, and `val typ`, the `ml-nlffigen` tool will generate a small set of structure elements for each field f of the `struct` or `union`. Let t_f be the type of f :

`type tf` is an abbreviation for the ML encoding of t_f .

`!val typf` holds runtime type information regarding t_f . If t_f is incomplete, then `typf` is omitted.

`!val ff` is the heavy-weight access function for f . It maps a value of type `(tag C.su, ζ) C.obj` to a value of type `(tf, ζ_f) C.obj` and is polymorphic in ζ . If f was declared `const`, then $\zeta_f = C.ro$. Otherwise $\zeta_f = \zeta$. If t_f is incomplete, then `ff` is omitted.

`val ff'` is the light-weight access function for f . It maps a value of type `(tag C.su, ζ) C.obj'` to a value of type `(tf, ζ_f) C.obj'` and is polymorphic in ζ . If f was declared `const`, then $\zeta_f = C.ro$. Otherwise $\zeta_f = \zeta$.

Bitfields

If f is a bitfield, then two access functions are generated:

`val ff` is the heavy-weight access function, mapping values of type `(tag C.su, ζ) C.obj` to either ζ_f `C.sbf` or ζ_f `C.ubf`, depending on whether the type of f is signed or unsigned. The function is polymorphic in ζ . If f was declared `const`, then $\zeta_f = C.ro$. Otherwise, $\zeta_f = \zeta$.

`val ff'` is the light-weight access function, mapping values of type `(tag C.su, ζ) C.obj'` to either ζ_f `C.sbf` or ζ_f `C.ubf`, using the same conventions as those used for `ff`.

Example

C declaration	signature of ML-side representation
<pre> struct t { int i; const double d; struct t *nx; /* complete */ struct s *ms; /* incomplete */ const int f : 2; unsigned g : 3; }; </pre>	<pre> structure S_t : sig type tag = ... val size : tag C.su C.S.size val typ : tag C.su C.T.typ type t_f_i = C.T.sint val typ_f_i : t_f_i C.T.typ val f_i : (tag C.su, 'c) obj -> (t_f_i, 'c) C.obj val f_i' : (tag C.su, 'c) obj' -> (t_f_i, 'c) C.obj' type t_f_d = C.T.double val typ_f_d : t_f_d C.T.typ val f_d : (tag C.su, 'c) obj -> (t_f_d, C.ro) C.obj val f_d' : (tag C.su, 'c) obj' -> (t_f_d, C.ro) C.obj' type t_f_nx = (tag C.su, C.rw) C.obj C.ptr val typ_f_nx : t_f_nx C.T.typ val f_nx : (tag C.su, 'c) obj -> (t_f_nx, 'c) C.obj val f_nx' : (tag C.su, 'c) obj' -> (t_f_nx, 'c) C.obj' type t_f_ms = (ST_s.tag C.su, C.rw) C.obj C.ptr val f_ms' : (tag C.su, 'c) obj' -> (t_f_ms, 'c) C.obj' val f_f : (tag C.su, 'c) C.obj -> C.ro C.sbf val f_f' : (tag C.su, 'c) C.obj' -> C.ro C.sbf val f_g : (tag C.su, 'c) C.obj -> 'c C.ubf val f_g' : (tag C.su, 'c) C.obj' -> 'c C.ubf end </pre>

Unnamed structs or unions

Each occurrence of an unnamed struct or union in C has its own type identity. The `ml-nlffigen` tool models this by artificially generating a unique tag for each such occurrence. The tags are chosen in such a way that they cannot clash with real tag names that might occur elsewhere in the C code. After choosing a fresh tag t , `ml-nlffigen` produces ML code according to the same rules that it uses when t is a real tag explicitly present in the C code.

Here are the rules for generating tags:

- If the struct- or union-declaration occurs at top level, i.e., not within the context of a typedef or another struct- or union-declaration, the generated tag consists of a sequence of decimal digits and can be read as a non-negative number.
- If the immediate context of the unnamed struct or union is a typedef for a type name t , then the generated tag will be $'t$.
- The tag of an unnamed struct or union is another (named or unnamed) struct or union with (real or generated) tag t is chosen to be $t'n$ where n is a fresh sequence of decimal digits that can be read as a non-negative number.

Examples

C declaration	signature of ML-side representation
<pre>struct { int i; };</pre>	<pre>structure S_0 : sig type tag = ... val size : tag C.su C.S.size val typ : tag C.su C.T.typ type t_f_i = C.T.sint val typ_f_i : t_f_i C.T.typ val f_i : (tag C.su, 'c) obj -> (t_f_i, 'c) C.obj val f_i' : (tag C.su, 'c) obj' -> (t_f_i, 'c) C.obj' end</pre>
<pre>typedef struct { int j; } s;</pre>	<pre>structure S_'s : sig type tag = ... val size : tag C.su C.S.size val typ : tag C.su C.T.typ type t_f_j = C.T.sint val typ_f_j : t_f_j C.T.typ val f_j : (tag C.su, 'c) obj -> (t_f_j, 'c) C.obj val f_j' : (tag C.su, 'c) obj' -> (t_f_j, 'c) C.obj' end</pre>
<pre>struct s { struct { int j; } x; };</pre>	<pre>structure S_s'0 : sig type tag = ... val size : tag C.su C.S.size val typ : tag C.su C.T.typ type t_f_j = C.sint val typ_f_j : t_f_j C.T.typ val f_j : (tag C.su, 'c) C.obj -> (t_f_j, 'c) C.obj val f_j' : (tag C.su, 'c) C.obj' -> (t_f_j, 'c) C.obj' end structure S_s : sig type tag = ... val size : tag C.su C.S.size val typ : tag C.su C.T.typ type t_f_x = S_s'0.tag C.su val typ_f_x : t_f_x C.T.typ val f_x : (tag C.su, 'c) C.obj -> (t_f_x, 'c) C.obj val f_x' : (tag C.su, 'c) C.obj' -> (t_f_x, 'c) C.obj' end</pre>

3.5 Enumerations (enum)

A C enumeration of constants c_1, c_2, \dots, c_k declared via `enum` is represented by k ML values of a chosen ML representation type. By default, that type is `MLRep.Signed.int`, i.e., the same type that also represents the C type `int`. A command line switch (`-enum-constructors` or `-ec`) to `ml-nlffigen` can change this behavior in such a way that whenever possible the representation type for an enumeration becomes an ML datatype, thus making it possible to perform pattern-matching on constants. The representation type cannot be a datatype if two or more `enum` constants share the same value as in:

```
enum ab { A = 12, B = 12 };
```

Complete enumerations

Let t be the tag of the `C.enum` declaration, and let c_1, \dots, c_k be its set of constants. The ML-side representative of such a declaration is a structure `E.t` which contains $10 + k$ elements, the first 10 being:

```
type tag The ML-side encoding of type enum  $t$  is tag C.enum. Values of this type are abstract. They can
      be converted to and from concrete integer values of type MLRep.Signed.int using C.Cvt.c2i_enum and
      C.Cvt.i2c_enum, respectively. Like in the case of struct or union, type tag is an abbreviation for the
      pre-defined type that uniquely corresponds to the tag name  $t$ .

type mlrep This is the type of concrete ML-side values representing the  $c_1, \dots, c_k$ . This type is not the same as tag
      C.enum and defaults to MLRep.Signed.int. As mentioned above, by specifying the -enum-constructors
      or -ec command-line flag one can force ml-nlffigen to generate a datatype definition for type mlrep.

val m2i This is a function for converting mlrep values to values of type MLRep.Signed.int. If the former is the
      same type as the latter (see above), then m2i is the identity function. Otherwise ml-nlffigen generates explicit
      code to map each mlrep constructor to an integer value.

val i2m This is the inverse of m2i. If mlrep is a datatype, then m2i will raise exception Domain when the argument
      does not correspond to one of the constructors.

val c Function c converts values of type mlrep to values of type tag C.enum. It is merely a composition of
      C.Cvt.i2c_enum and m2i.

val ml Function ml is the composition of i2m and C.Cvt.c2i_enum and converts values of type tag C.enum to
      values of type mlrep. It can raise exception Domain if the C type system had been subverted (which is always a
      real possibility).

val get Function get fetches a value of type mlrep from a memory object of type (tag C.enum, ζ) C.obj.
      It is a composition of i2m and C.Get.enum.

val get' Function get' fetches a value of type mlrep from a memory object of type (tag C.enum, ζ) C.obj'.
      It is a composition of i2m and C.Get.enum'.

val set Function set stores a value of type mlrep into a memory object of type (tag C.enum, C.rw) C.obj.
      It is a composition of m2i and C.Set.enum.

val set' Function set' stores a value of type mlrep into a memory object of type (tag C.enum, C.rw) C.obj'.
      It is a composition of m2i and C.Set.enum'.
```

Each of the remaining k elements corresponds to one of the enumeration constants c_i . Concretely, the element generated for c_i is `val e_ci` and has type `mlrep`. If `mlrep` is a datatype, then the `e_ci` are constructors which can be used in ML patterns.

Examples

C declaration	signature of ML-side representation
<pre>enum e { A, B, C }; /* default treatment */</pre>	<pre>structure E_e : sig type tag = ... type mlrep = MLRep.Signed.int val e_A : mlrep (* = 0 *) val e_B : mlrep (* = 1 *) val e_C : mlrep (* = 2 *) val m2i : mlrep -> MLRep.Signed.int val i2m : MLRep.Signed.int -> mlrep val c : mlrep -> tag C.enum val ml : tag C.enum -> mlrep val get : (tag C.enum, 'c) C.obj -> mlrep val get' : (tag C.enum, 'c) C.obj' -> mlrep val set : (tag C.enum, C.rw) C.obj * mlrep -> unit val set' : (tag C.enum, C.rw) C.obj' * mlrep -> unit end</pre>
<pre>enum e { A, B, C }; /* -enum-constructors */</pre>	<pre>structure E_e : sig type tag = ... datatype mlrep = e_A e_B e_C val m2i : mlrep -> MLRep.Signed.int val i2m : MLRep.Signed.int -> mlrep val c : mlrep -> tag C.enum val ml : tag C.enum -> mlrep val get : (tag C.enum, 'c) C.obj -> mlrep val get' : (tag C.enum, 'c) C.obj' -> mlrep val set : (tag C.enum, C.rw) C.obj * mlrep -> unit val set' : (tag C.enum, C.rw) C.obj' * mlrep -> unit end</pre>
<pre>enum e { A = 0, B = 1, C = 0 }; /* with or without * -enum-constructors */</pre>	<pre>structure E_e : sig type tag = ... type mlrep = MLRep.Signed.int val e_A : mlrep (* = 0 *) val e_B : mlrep (* = 1 *) val e_C : mlrep (* = 0 *) val m2i : mlrep -> MLRep.Signed.int val i2m : MLRep.Signed.int -> mlrep val c : mlrep -> tag C.enum val ml : tag C.enum -> mlrep val get : (tag C.enum, 'c) C.obj -> mlrep val get' : (tag C.enum, 'c) C.obj' -> mlrep val set : (tag C.enum, C.rw) C.obj * mlrep -> unit val set' : (tag C.enum, C.rw) C.obj' * mlrep -> unit end</pre>

Incomplete enumerations

If the enumeration is incomplete, i.e., if only its tag t is known, then no structure $E.t$ is generated. Instead, a structure $ET.t$ takes its place which merely contains the type `tag` as described above.

Unnamed enumerations

Anonymous enumerations (enums without a tag) are handled in a way that is very similar to the treatment of unnamed structs and unions. In particular, the rules for assigning a generated tag are the same if the enum occurs in the context of a typedef or another struct or union.

However, by default all constants in unnamed top-level enums get collected into one single virtual enumeration whose tag is ' (apostrophe). If this is not desired, then the command line flag `-nocollect` turns this off and lets `ml-nlffigen` fall back to the exact same rules that are used for unnamed top-level structs and unions: a fresh “numeric” tag gets generated for each such enum.

Examples for collected unnamed enumerations

C declaration	signature of ML-side representation
<pre>enum { A, B }; enum { C, D }; /* with or without * -enum-constructors */</pre>	<pre>structure E_\' : sig type tag = ... type mlrep = MLRep.Signed.int val e_A : mlrep (* = 0 *) val e_B : mlrep (* = 1 *) val e_C : mlrep (* = 0 *) val e_D : mlrep (* = 1 *) ... end</pre>
<pre>enum { A, B }; enum { C = 2, D }; /* -enum-constructors */</pre>	<pre>structure E_\' : sig type tag = ... datatype mlrep = e_A e_B e_C e_D ... end</pre>