FFI: PLT Scheme Foreign Interface

Version 4.1.5

Eli Barzilay

March 21, 2009

```
(require scheme/foreign)
```

The `scheme/foreign` library enables the direct use of C-based APIs within Scheme programs—without writing any new C code. From the Scheme perspective, functions and data with a C-based API are *foreign*, hence the term *foreign interface*. Furthermore, since most APIs consist mostly of functions, the foreign interface is sometimes called a *foreign function interface*, abbreviated *FFI*.

Important: Many of the bindings documented here (the ones in sections with titles starting "Unsafe") are available only after an `(unsafe!)` declaration in the importing module.

Contents

1 Overview

Although using the FFI requires writing no new C code, it provides very little insulation against the issues that C programmer faces related to safety and memory management. An FFI programmer must be particularly aware of memory management issues for data that spans the Scheme–C divide. Thus, this manual relies in many ways on the information in § "**Inside**: PLT Scheme C API", which defines how PLT Scheme interacts with C APIs in general.

Since using the FFI entails many safety concerns that Scheme programmers can normally ignore, merely importing `scheme/foreign` with `(require scheme/foreign)` does not import all of the FFI functionality. Only safe functionality is immediately imported. For example, `ptr-equal?` can never cause memory corruption or an invalid memory access, so it is immediately available on import.

Use `(unsafe!)` at the top-level of a module that imports `scheme/foreign` to make unsafe features accessible. (For additional safety, the `unsafe!` is itself protected; see §13.9 "Code Inspectors".) Using this macro should be considered as a declaration that your code is itself unsafe, therefore can lead to serious problems in case of bugs: it is your responsibility to provide a safe interface. Bindings that become available only via `unsafe!` are documented in this manual in sections with titles starting "Unsafe."

For examples of common FFI usage patterns, see the defined interfaces in the `"ffi"` collection.

2 Loading Foreign Libraries

The FFI is normally used by extracting functions and other objects from shared objects (a.k.a. *shared libraries* or *dynamically loaded libraries*). The `ffi-lib` function loads a shared object.

```
(ffi-lib? v) → boolean>
  v : any/c
```

Returns `#t` if *v* is the result of `ffi-lib`, `#f` otherwise.

2.1 Unsafe Library Functions

```
(ffi-lib path [version]) → any
  path : (or/c path-string? #f)
  version : (or/c string? (listof string?) #f) = #f
```

Returns an foreign-library value. If *path* is a path, the result represents the foreign library, which is opened in an OS-specific way (using `LoadLibrary` under Windows, and `dlopen` under Unix and Mac OS X).

The path is not expected to contain the library suffix, which is added according to the current platform. If adding the suffix fails, several other filename variations are tried — retrying without an automatically added suffix, and using a full path of a file if it exists relative to the current directory (since the OS-level library function usually searches, unless the library name is an absolute path). An optional *version* string can be supplied, which is appended to the name after any added suffix. If you need any of a few possible versions, use a list of version strings, and `ffi-lib` will try all of them.

If *path* is `#f`, then the resulting foreign-library value represents all libraries loaded in the current process, including libraries previously opened with `ffi-lib`. In particular, use `#f` to access C-level functionality exported by the run-time system (as described in § "**Inside**: PLT Scheme C API").

Note: `ffi-lib` tries to look for the library file in a few places like the PLT libraries (see `get-lib-search-dirs`), a relative path, or a system search. However, if `dlopen` cannot open a library, there is no reliable way to know why it failed, so if all path combinations fail, it will raise an error with the result of `dlopen` on the unmodified argument name. For example, if you have a local `"foo.so"` library that cannot be loaded because of a missing symbol, using `(ffi-lib "foo.so")` will fail with all its search options, most because the library is not found, and once because of the missing symbol, and eventually produce an error message that comes from `dlopen("foo.so")` which will look like the file is not found. In such cases try to specify a full or relative path (containing slashes, e.g., `"./foo.so"`).

```
(get-ffi-obj objname lib type [failure-thunk]) → any
  objname : (or/c string? bytes? symbol?)
  lib : (or/c ffi-lib? path-string? #f)
  type : ctype?
  failure-thunk : (or/c (-> any) #f) = #f
```

Looks for the given object name *objname* in the given *lib* library. If *lib* is not a foreign-library value produced by `ffi-lib`, it is converted to one by calling `ffi-lib`. If *objname* is found in *lib*, it is converted to Scheme using the given *type*. Types are described in §3 "C Types"; in particular the `get-ffi-obj` procedure is most often used with function types created with `_fun`.

Keep in mind that `get-ffi-obj` is an unsafe procedure; see §1 "Overview" for details.

If the object is not found, and *failure-thunk* is provided, it is used to produce a return value. For example, a failure thunk can be provided to report a specific error if an object is not found:

```
(define foo
  (get-ffi-obj "foo" foolib (_fun _int -> _int)
    (lambda ()
      (error 'foolib
             "installed foolib does not provide \"foo\""))))
```

The default (also when *failure-thunk* is provided as `#f`) is to raise an exception.

```
(set-ffi-obj! objname lib type new) → void?
  objname : (or/c string? bytes? symbol?)
  lib : (or/c ffi-lib? path-string? #f)
  type : ctype?
  new : any/c
```

Looks for *objname* in *lib* similarly to `get-ffi-obj`, but then it stores the given *new* value into the library, converting it to a C value. This can be used for setting library customization variables that are part of its interface, including Scheme callbacks.

```
(make-c-parameter objname lib type) → (and/c (-> any)
                                             (any/c -> void?))
  objname : (or/c string? bytes? symbol?)
  lib : (or/c ffi-lib? path-string? #f)
  type : ctype?
```

Returns a parameter-like procedure that can either references the specified foreign value, or set it. The arguments are handled as in `get-ffi-obj`.

A parameter-like function is useful in case Scheme code and library code interact through a library value. Although `make-c-parameter` can be used with any time, it is not recommended to use this for foreign functions, since each reference through the parameter will construct the low-level interface before the actual call.

```
(define-c id lib-expr type-expr)
```

Defines *id* behave like a Scheme binding, but *id* is actually redirected through a parameter-like procedure created by `make-c-parameter`. The *id* is used both for the Scheme binding and for the foreign object's name.

```
(ffi-obj-ref objname lib [failure-thunk]) → any
  objname : (or/c string? bytes? symbol?)
  lib : (or/c ffi-lib? path-string? #f)
  failure-thunk : (or/c (-> any) #f) = #f
```

Returns a pointer object for the specified foreign object. This procedure is for rare cases where `make-c-parameter` is insufficient, because there is no type to cast the foreign object to (e.g., a vector of numbers).

3 C Types

C types are the main concept of the FFI, either primitive types or user-defined types. The FFI deals with primitive types internally, converting them to and from C types. A user type is defined in terms of existing primitive and user types, along with conversion functions to and from the existing types.

3.1 Type Constructors

```
(make-ctype type scheme-to-c c-to-scheme) → ctype?
  type : ctype?
  scheme-to-c : (or/c #f (any/c . -> . any))
  c-to-scheme : (or/c #f (any/c . -> . any))
```

Creates a new C type value whose representation for foreign code is the same as *type*'s. The given conversions functions convert to and from the Scheme representation of *type*. Either conversion function can be `#f`, meaning that the conversion for the corresponding direction is the identity function. If both functions are `#f`, *type* is returned.

```
(ctype? v) → boolean?
  v : any/c
```

Returns `#t` if *v* is a C type, `#f` otherwise.

```
(ctype-sizeof type) → exact-nonnegative-integer?
  type : ctype?
(ctype-alignof type) → exact-nonnegative-integer?
  type : ctype?
```

Returns the size or alignment of a given *type* for the current platform.

```
(ctype->layout type) → (flat-rec-contract rep
                         symbol?
                         (listof rep))
  type : ctype?
```

Returns a value to describe the eventual C representation of the type. It can be any of the following symbols:

```
'int8 'uint8 'int16 'uint16 'int32 'uint32 'int64 'uint64
'float 'double 'bool 'void 'pointer 'fpointer
```

'bytes 'string/ucs-4 'string/utf-16

The result can also be a list, which describes a C struct whose element representations are provided in order within the list.

(compiler-sizeof *sym*) → exact-nonnegative-integer?
  sym : symbol?

Possible values for symbol are 'int, 'char, 'short, 'long, '*, 'void, 'float, 'double. The result is the size of the correspond type according to the C sizeof operator for the current platform. The compiler-sizeof operation should be used to gather information about the current platform, such as defining alias type like _int to a known type like _int32.

3.2 Numeric Types

_int8 : ctype?
_sint8 : ctype?
_uint8 : ctype?
_int16 : ctype?
_sint16 : ctype?
_uint16 : ctype?
_int32 : ctype?
_sint32 : ctype?
_uint32 : ctype?
_int64 : ctype?
_sint64 : ctype?
_uint64 : ctype?

The basic integer types at various sizes. The s or u prefix specifies a signed or an unsigned integer, respectively; the ones with no prefix are signed.

_byte : ctype?
_sbyte : ctype?
_ubyte : ctype?
_short : ctype?
_sshort : ctype?
_ushort : ctype?
_int : ctype?
_sint : ctype?
_uint : ctype?
_word : ctype?

```
_sword : ctype?
_uword : ctype?
_long : ctype?
_slong : ctype?
_ulong : ctype?
```

Aliases for basic integer types. The `_byte` aliases correspond to `_int8`. The `_short` and `_word` aliases correspond to `_int16`. The `_int` aliases correspond to `_int32`. The `_long` aliases correspond to either `_int32` or `_int64`, depending on the platform.

```
_fixnum : ctype?
_ufixnum : ctype?
```

For cases where speed matters and where you know that the integer is small enough, the types `_fixnum` and `_ufixnum` are similar to `_long` and `_ulong` but assume that the quantities fit in PLT Scheme's immediate integers (i.e., not bignums).

```
_fixint : ctype?
_ufixint : ctype?
```

Like `_fixnum` and `_ufixnum`, but coercions from C are checked to be in range.

```
_float : ctype?
_double : ctype?
_double* : ctype?
```

The `_float` and `_double` types represent the corresponding C types. The type `_double*` that implicitly coerces any real number to a C `double`.

3.3 Other Atomic Types

```
_bool : ctype?
```

Translates `#f` to a `0` `_int`, and any other value to `1`.

```
_void : ctype?
```

Indicates a Scheme `#<void>` return value, and it cannot be used to translate values to C. This type cannot be used for function inputs.

3.4 String Types

3.4.1 Primitive String Types

The five primitive string types correspond to cases where a C representation matches MzScheme's representation without encodings.

The form `_bytes` form can be used type for Scheme byte strings, which corresponds to C's `char*` type. In addition to translating byte strings, `#f` corresponds to the `NULL` pointer.

`_string/ucs-4 : ctype?`

A type for Scheme's native Unicode strings, which are in UCS-4 format. These correspond to the C `mzchar*` type used by PLT Scheme. As usual, the types treat `#f` as `NULL` and vice-versa.

`_string/utf-16 : ctype?`

Unicode strings in UTF-16 format. As usual, the types treat `#f` as `NULL` and vice-versa.

`_path : ctype?`

Simple `char*` strings, corresponding to Scheme's paths. As usual, the types treat `#f` as `NULL` and vice-versa.

`_symbol : ctype?`

Simple `char*` strings as Scheme symbols (encoded in UTF-8). Return values using this type are interned as symbols.

3.4.2 Fixed Auto-Converting String Types

`_string/utf-8 : ctype?`
`_string/latin-1 : ctype?`
`_string/locale : ctype?`

Types that correspond to (character) strings on the Scheme side and `char*` strings on the C side. The bridge between the two requires a transformation on the content of the string. As usual, the types treat `#f` as `NULL` and vice-versa.

_string*/utf-8 : ctype?
_string*/latin-1 : ctype?
_string*/locale : ctype?

Similar to _string/utf-8, etc., but accepting a wider range of values: Scheme byte strings are allowed and passed as is, and Scheme paths are converted using path->bytes.

3.4.3 Variable Auto-Converting String Type

The _string/ucs-4 type is rarely useful when interacting with foreign code, while using _bytes is somewhat unnatural, since it forces Scheme programmers to use byte strings. Using _string/utf-8, etc., meanwhile, may prematurely commit to a particular encoding of strings as bytes. The _string type supports conversion between Scheme strings and char* strings using a parameter-determined conversion.

_string : ctype?

Expands to a use of the default-_string-type parameter. The parameter's value is consulted when _string is evaluated, so the parameter should be set before any interface definition that uses _string.

(default-_string-type) → ctype?
(default-_string-type *type*) → void?
  type : ctype?

A parameter that determines the current meaning of _string. It is initially set to _string*/utf-8. If you change it, do so *before* interfaces are defined.

3.4.4 Other String Types

_file : ctype?

Like _path, but when values go from Scheme to C, cleanse-path is used on the given value. As an output value, it is identical to _path.

_bytes/eof : ctype?

Similar to the _bytes type, except that a foreign return value of NULL is translated to a Scheme eof value.

`_string/eof` : `ctype?`

Similar to the `_string` type, except that a foreign return value of `NULL` is translated to a Scheme `eof` value.

3.5 Pointer Types

`_pointer` : `ctype?`

Corresponds to Scheme "C pointer" objects. These pointers can have an arbitrary Scheme object attached as a type tag. The tag is ignored by built-in functionality; it is intended to be used by interfaces. See §6.1 "Tagged C Pointer Types" for creating pointer types that use these tags for safety.

`_scheme` : `ctype?`

This type can be used with any Scheme object; it corresponds to the `Scheme_Object*` type of PLT Scheme's C API (see § "**Inside**: PLT Scheme C API"). It is useful only for libraries that are aware of PLT Scheme's C API.

`_fpointer` : `ctype?`

Similar to `_pointer`, except that when an `_fpointer` is extracted from a pointer produced by `ffi-obj-ref`, then a level of indirection is skipped. A level of indirection is similarly skipped when extracting a pointer via `get-ffi-obj`. Like `_pointer`, `_fpointer` treats `#f` as `NULL` and vice-versa.

A type generated by `_cprocedure` builds on `_fpointer`, and normally `_cprocedure` should be used instead of `_fpointer`.

3.6 Function Types

```
(_cprocedure  input-types
              output-type
             [#:abi abi
              #:atomic? atomic?
              #:wrapper wrapper
              #:keep keep])      → any
  input-types : (list ctype?)
```

```
output-type : ctype?
abi : (or/c symbol/c #f) = #f
atomic? : any/c = #f
wrapper : (or/c #f (procedure? . -> . procedure?)) = #f
keep : (or/c boolean? box? (any/c . -> . any/c)) = #t
```

A type constructor that creates a new function type, which is specified by the given *input-types* list and *output-type*. Usually, the `_fun` syntax (described below) should be used instead, since it manages a wide range of complicated cases.

The resulting type can be used to reference foreign functions (usually `ffi-obj`s, but any pointer object can be referenced with this type), generating a matching foreign callout object. Such objects are new primitive procedure objects that can be used like any other Scheme procedure. As with other pointer types, `#f` is treated as a `NULL` function pointer and vice-versa.

A type created with `_cprocedure` can also be used for passing Scheme procedures to foreign functions, which will generate a foreign function pointer that calls the given Scheme procedure when it is used. There are no restrictions on the Scheme procedure; in particular, its lexical context is properly preserved.

The optional *abi* keyword argument determines the foreign ABI that is used. `#f` or `'default` will use a platform-dependent default; other possible values are `'stdcall` and `'sysv` (the latter corresponds to "cdecl"). This is especially important on Windows, where most system functions are `'stdcall`, which is not the default.

If *atomic?* is true, then when a Scheme procedure is given this procedure type and called from foreign code, then the PLT Scheme process is put into atomic mode while evaluating the Scheme procedure body. In atomic mode, other Scheme threads do not run, so the Scheme code must not call any function that potentially synchronizes with other threads, or else it may deadlock. In addition, the Scheme code must not perform any potentially blocking operation (such as I/O), it must not raise an uncaught exception, it must not perform any escaping continuation jumps, and its non-tail recursion must be minimal to avoid C-level stack overflow; otherwise, the process may crash or misbehave.

The optional *wrapper*, if provided, is expected to be a function that can change a callout procedure: when a callout is generated, the wrapper is applied on the newly created primitive procedure, and its result is used as the new function. Thus, *wrapper* is a hook that can perform various argument manipulations before the foreign function is invoked, and return different results (for example, grabbing a value stored in an "output" pointer and returning multiple values). It can also be used for callbacks, as an additional layer that tweaks arguments from the foreign code before they reach the Scheme procedure, and possibly changes the result values too.

Sending Scheme functions as callbacks to foreign code is achieved by translating them to a foreign "closure", which foreign code can call as plain C functions. Additional care must be taken in case the foreign code might hold on to the callback function. In these cases you

must arrange for the callback value to not be garbage-collected, or the held callback will become invalid. The optional *keep* keyword argument is used to achieve this. It can have the following values:

- `#t` makes the callback value stay in memory as long as the converted function is. In order to use this, you need to hold on to the original function, for example, have a binding for it. Note that each function can hold onto one callback value (it is stored in a weak hash table), so if you need to use a function in multiple callbacks you will need to use one of the the last two options below. (This is the default, as it is fine in most cases.)
- `#f` means that the callback value is not held. This may be useful for a callback that is only used for the duration of the foreign call — for example, the comparison function argument to the standard C library `qsort` function is only used while `qsort` is working, and no additional references to the comparison function are kept. Use this option only in such cases, when no holding is necessary and you want to avoid the extra cost.
- A box holding `#f` (or a callback value) — in this case the callback value will be stored in the box, overriding any value that was in the box (making it useful for holding a single callback value). When you know that it is no longer needed, you can "release" the callback value by changing the box contents, or by allowing the box itself to be garbage-collected. This is can be useful if the box is held for a dynamic extent that corresponds to when the callback is needed; for example, you might encapsulate some foreign functionality in a Scheme class or a unit, and keep the callback box as a field in new instances or instantiations of the unit.
- A box holding `null` (or any list) – this is similar to the previous case, except that new callback values are consed onto the contents of the box. It is therefore useful in (rare) cases when a Scheme function is used in multiple callbacks (that is, sent to foreign code to hold onto multiple times).
- Finally, if a one-argument function is provided as *keep*, it will be invoked with the callback value when it is generated. This allows you to grab the value directly and use it in any way.

```
(_fun fun-option ... maybe-args type-spec ... -> type-spec
      maybe-wrapper)
```

```
   fun-option = #:abi  abi-expr
              | #:keep keep-expr
              | #:atomic? atomic?-expr

   maybe-args =
              | (id ...) ::
              | id ::
              | (id ... . id) ::

    type-spec = type-expr
              | (id : type-expr)
              | (type-expr = value-expr)
              | (id : type-expr = value-expr)

maybe-wrapper =
              | -> output-expr
```

Creates a new function type. The `_fun` form is a convenient syntax for the `_cprocedure` type constructor. In its simplest form, only the input *type-expr*s and the output *type-expr* are specified, and each types is a simple expression, which creates a straightforward function type.

In its full form, the `_fun` syntax provides an IDL-like language that can be used to create a wrapper function around the primitive foreign function. These wrappers can implement complex foreign interfaces given simple specifications. The full form of each of the type specifications can include an optional label and an expression. If a `= value-expr` is provided, then the resulting function will be a wrapper that calculates the argument for that position itself, meaning that it does not expect an argument for that position. The expression can use previous arguments if they were labeled with *id* `:`. In addition, the result of a function call need not be the value returned from the foreign call: if the optional *output-expr* is specified, or if an expression is provided for the output type, then this specifies an expression that will be used as a return value. This expression can use any of the previous labels, including a label given for the output which can be used to access the actual foreign return value.

In rare cases where complete control over the input arguments is needed, the wrapper's argument list can be specified as `args`, in any form (including a "rest" argument). Identifiers in this place are related to type labels, so if an argument is there is no need to use an expression.

For example,

```
(_fun (n s) :: (s : _string) (n : _int) -> _int)
```

specifies a function that receives an integer and a string, but the foreign function receives the string first.

```
(function-ptr ptr-or-proc fun-type) → cpointer?
  ptr-or-proc : (or cpointer? procedure?)
  fun-type : ctype?
```

Casts *ptr-or-proc* to a function pointer of type *fun-type*.

3.6.1 Custom Function Types

The behavior of the `_fun` type can be customized via *custom function types*, which are pieces of syntax that can behave as C types and C type constructors, but they can interact with function calls in several ways that are not possible otherwise. When the `_fun` form is expanded, it tries to expand each of the given type expressions, and ones that expand to certain keyword-value lists interact with the generation of the foreign function wrapper. This expansion makes it possible to construct a single wrapper function, avoiding the costs involved in compositions of higher-order functions.

Custom function types are macros that expand to a sequence `(key: val ...)`, where each `key:` is from a short list of known keys. Each key interacts with generated wrapper functions in a different way, which affects how its corresponding argument is treated:

- `type:` specifies the foreign type that should be used, if it is `#f` then this argument does not participate in the foreign call.
- `expr:` specifies an expression to be used for arguments of this type, removing it from wrapper arguments.
- `bind:` specifies a name that is bound to the original argument if it is required later (e.g., `_box` converts its associated value to a C pointer, and later needs to refer back to the original box).
- `1st-arg:` specifies a name that can be used to refer to the first argument of the foreign call (good for common cases where the first argument has a special meaning, e.g., for method calls).
- `prev-arg:` similar to `1st-arg:`, but refers to the previous argument.
- `pre:` a pre-foreign code chunk that is used to change the argument's value.
- `post:` a similar post-foreign code chunk.

The `pre:` and `post:` bindings can be of the form `(id => expr)` to use the existing value. Note that if the `pre:` expression is not `(id => expr)`, then it means that there is no input for this argument to the `_fun`-generated procedure. Also note that if a custom type is used as an output type of a function, then only the `post:` code is used.

Most custom types are meaningful only in a `_fun` context, and will raise a syntax error if used elsewhere. A few such types can be used in non-`_fun` contexts: types which use only `type:`, `pre:`, `post:`, and no others. Such custom types can be used outside a `_fun` by expanding them into a usage of `make-ctype`, using other keywords makes this impossible, because it means that the type has specific interaction with a function call.

`(define-fun-syntax id transformer-expr)`

Binds *id* as a custom function type. The type is expanded by applying the procedure produced by *transformer-expr* to a use of the custom function type.

`_?`

A custom function type that is a marker for expressions that should not be sent to the foreign function. Use this to bind local values in a computation that is part of an ffi wrapper interface, or to specify wrapper arguments that are not sent to the foreign function (e.g., an argument that is used for processing the foreign output).

`(_ptr mode type-expr)`

```
mode = i
     | o
     | io
```

Creates a C pointer type, where *mode* indicates input or output pointers (or both). The *mode* can be one of the following:

- `i` — indicates an *input* pointer argument: the wrapper arranges for the function call to receive a value that can be used with the `type` and to send a pointer to this value to the foreign function. After the call, the value is discarded.
- `o` — indicates an *output* pointer argument: the foreign function expects a pointer to a place where it will save some value, and this value is accessible after the call, to be used by an extra return expression. If `_ptr` is used in this mode, then the generated wrapper does not expect an argument since one will be freshly allocated before the call.
- `io` — combines the above into an *input/output* pointer argument: the wrapper gets the Scheme value, allocates and set a pointer using this value, and then references the value after the call. The "`_ptr`" name can be confusing here: it means that the foreign function expects a pointer, but the generated wrapper uses an actual value. (Note that if this is used with structs, a struct is created when calling the function, and a copy of the return value is made too—which is inefficient, but ensures that structs are not modified by C code.)

For example, the `_ptr` type can be used in output mode to create a foreign function wrapper that returns more than a single argument. The following type:

```
(_fun (i : (_ptr o _int))
      -> (d : _double)
      -> (values d i))
```

creates a function that calls the foreign function with a fresh integer pointer, and use the value that is placed there as a second return value.

`_box`

A custom function type similar to a `(_ptr io type)` argument, where the input is expected to be a box holding an appropriate value, which is unboxed on entry and modified accordingly on exit.

`(_list mode type maybe-len)`

```
     mode = i
          | o
          | io

maybe-len =
          | len-expr
```

A custom function type that is similar to `_ptr`, except that it is used for converting lists to/from C vectors. The optional *maybe-len* argument is needed for output values where it is used in the post code, and in the pre code of an output mode to allocate the block. In either case, it can refer to a previous binding for the length of the list which the C function will most likely require.

`(_vector mode type maybe-len)`

A custom function type like `_list`, except that it uses Scheme vectors instead of lists.

`(_bytes o len-expr)`
`_bytes`

A custom function type that can be used by itself as a simple type for a byte string as a C pointer. Alternatively, the second form is for a pointer return value, where the size should be explicitly specified.

There is no need for other modes: input or input/output would be just like `_bytes`, since the string carries its size information (there is no real need for the `o` part of the syntax, but it is

present for consistency with the above macros).

```
(_cvector mode type maybe-len)
_cvector
```

Like `_bytes`, `_cvector` can be used as a simple type that corresponds to a pointer that is managed as a safe C vector on the Scheme side; see §6.2 “Safe C Vectors”. The longer form behaves similarly to the `_list` and `_vector` custom types, except that `_cvector` is more efficient; no Scheme list or vector is needed.

3.7 C Struct Types

```
(make-cstruct-type types) → ctype?
  types : (listof ctype?)
```

The primitive type constructor for creating new C struct types. These types are actually new primitive types; they have no conversion functions associated. The corresponding Scheme objects that are used for structs are pointers, but when these types are used, the value that the pointer *refers to* is used, rather than the pointer itself. This value is basically made of a number of bytes that is known according to the given list of *types* list.

```
(_list-struct type ...+) → ctype?
  type : ctype?
```

A type constructor that builds a struct type using `make-cstruct-type` function and wraps it in a type that marshals a struct as a list of its components. Note that space for structs must to be allocated; the converter for a `_list-struct` type immediately allocates and uses a list from the allocated space, so it is inefficient. Use `define-cstruct` below for a more efficient approach.

```
(define-cstruct id/sup ([field-id type-expr] ...))

id/sup = _id
       | (_id super-id)
```

Defines a new C struct type, but unlike `_list-struct`, the resulting type deals with C structs in binary form, rather than marshaling them to Scheme values. The syntax is similar to `define-struct`, providing accessor functions for raw struct values (which are pointer objects). The new type uses pointer tags to guarantee that only proper struct objects are used. The *_id* must start with `_`.

The resulting bindings are as follows:

- `_id` : the new C type for this struct.
- `_id-pointer`: a pointer type that should be used when a pointer to values of this struct are used.
- `id?`: a predicate for the new type.
- `id-tag`: the tag string object that is used with instances.
- `make-id` : a constructor, which expects an argument for each type.
- `id-field-id` : an accessor function for each `field-id`.
- `set-id-field-id!` : a mutator function for each `field-id`.

Objects of the new type are actually C pointers, with a type tag that is a list that contains the string form of `id`. Since structs are implemented as pointers, they can be used for a `_pointer` input to a foreign function: their address will be used. To make this a little safer, the corresponding cpointer type is defined as `_id-pointer`. The `_id` type should not be used when a pointer is expected, since it will cause the struct to be copied rather than use the pointer value, leading to memory corruption.

If the first field is itself a cstruct type, its tag will be used in addition to the new tag. This feature supports common cases of object inheritance, where a sub-struct is made by having a first field that is its super-struct. Instances of the sub-struct can be considered as instances of the super-struct, since they share the same initial layout. Using the tag of an initial cstruct field means that the same behavior is implemented in Scheme; for example, accessors and mutators of the super-cstruct can be used with the new sub-cstruct. See the example below.

Providing a `super-id` is shorthand for using an initial field named `super-id` and using `_super-id` as its type. Thus, the new struct will use `_super-id`'s tag in addition to its own tag, meaning that instances of `_id` can be used as instances of `_super-id`. Aside from the syntactic sugar, the constructor function is different when this syntax is used: instead of expecting a first argument that is an instance of `_super-id`, the constructor will expect arguments for each of `_super-id`'s fields, in addition for the new fields. This adjustment of the constructor is, again, in analogy to using a supertype with `define-struct`.

Note that structs are allocated as atomic blocks, which means that the garbage collector ignores their content. Currently, there is no safe way to store pointers to GC-managed objects in structs (even if you keep a reference to avoid collecting the referenced objects, a the 3m variant's GC will invalidate the pointer's value). Thus, only non-pointer values and pointers to memory that is outside the GC's control can be placed into struct fields.

As an example, consider the following C code:

```
typedef struct { int x; char y; } A;
typedef struct { A a; int z; } B;
```

```
A* makeA() {
  A *p = malloc(sizeof(A));
  p->x = 1;
  p->y = 2;
  return p;
}

B* makeB() {
  B *p = malloc(sizeof(B));
  p->a.x = 1;
  p->a.y = 2;
  p->z   = 3;
  return p;
}

char gety(A* a) {
  return a->y;
}
```

Using the simple `_list-struct`, you might expect this code to work:

```
(define makeB
  (get-ffi-obj 'makeB "foo.so"
    (_fun -> (_list-struct (_list-struct _int _byte) _int))))
(makeB) ; should return '((1 2) 3)
```

The problem here is that `makeB` returns a pointer to the struct rather than the struct itself. The following works as expected:

```
(define makeB
  (get-ffi-obj 'makeB "foo.so" (_fun -> _pointer)))
(ptr-ref (makeB) (_list-struct (_list-struct _int _byte) _int))
```

As described above, `_list-struct`s should be used in cases where efficiency is not an issue. We continue using `define-cstruct`, first define a type for `A` which makes it possible to use `makeA`:

```
(define-cstruct _A ([x _int] [y _byte]))
(define makeA
  (get-ffi-obj 'makeA "foo.so"
    (_fun -> _A-pointer))) ; using _A is a memory-corrupting bug!
(define a (makeA))
(list a (A-x a) (A-y a))
; produces an A containing 1 and 2
```

Using `gety` is also simple:

```
(define gety
  (get-ffi-obj 'gety "foo.so"
    (_fun _A-pointer -> _byte)))
(gety a) ; produces 2
```

We now define another C struct for `B`, and expose `makeB` using it:

```
(define-cstruct _B ([a _A] [z _int]))
(define makeB
  (get-ffi-obj 'makeB "foo.so"
    (_fun -> _B-pointer)))
(define b (makeB))
```

We can access all values of `b` using a naive approach:

```
(list (A-x (B-a b)) (A-y (B-a b)) (B-z b))
```

but this is inefficient as it allocates and copies an instance of `A` on every access. Inspecting the tags `(cpointer-tag b)` we can see that `A`'s tag is included, so we can simply use its accessors and mutators, as well as any function that is defined to take an `A` pointer:

```
(list (A-x b) (A-y b) (B-z b))
(gety b)
```

Constructing a `B` instance in Scheme requires allocating a temporary `A` struct:

```
(define b (make-B (make-A 1 2) 3))
```

To make this more efficient, we switch to the alternative `define-cstruct` syntax, which creates a constructor that expects arguments for both the super fields ands the new ones:

```
(define-cstruct (_B _A) ([z _int]))
(define b (make-B 1 2 3))
```

3.8 Enumerations and Masks

Although the constructors below are describes as procedures,they are implemented as syntax, so that error messages can report a type name where the syntactic context implies one.

`(_enum` *symbols* `[`*basetype*`])` → `ctype?`
  symbols : `list?`
  basetype : `ctype?` = `_ufixint`

Takes a list of symbols and generates an enumeration type. The enumeration maps between the given *symbols* and integers, counting from `0`.

The list *symbols* can also set the values of symbols by putting '= and an exact integer after the symbol. For example, the list '(x y = 10 z) maps 'x to 0, 'y to 10, and 'z to 11.

The *basetype* argument specifies the base type to use.

```
(_bitmask symbols [basetype]) → ctype?
  symbols : (or symbol? list?)
  basetype : ctype? = _uint
```

Similar to _enum, but the resulting mapping translates a list of symbols to a number and back, using bitwise-or. A single symbol is equivalent to a list containing just the symbol. The default *basetype* is _uint, since high bits are often used for flags.

4 Pointer Functions

```
(cpointer? v) → boolean?
  v : any/c
```

Returns `#t` if *v* is a C pointer or a value that can be used as a pointer: `#f` (used as a `NULL` pointer), byte strings (used as memory blocks), or some additional internal objects (`ffi-obj`s and callbacks, see §7 "Unexported Primitive Functions"). Returns `#f` for other values.

```
(ptr-equal? cptr1 cptr2) → boolean?
  cptr1 : cpointer?
  cptr2 : cpointer?
```

Compares the values of the two pointers. Two different Scheme pointer objects can contain the same pointer.

```
(ptr-add cptr offset [type]) → cpointer?
  cptr : cpointer?
  offset : exact-integer?
  type : ctype? = _byte
```

Returns a cpointer that is like *cptr* offset by *offset* instances of `ctype`.

The resulting cpointer keeps the base pointer and offset separate. The two pieces are combined at the last minute before any operation on the pointer, such as supplying the pointer to a foreign function. In particular, the pointer and offset are not combined until after all allocation leading up to a foreign-function call; if the called function does not itself call anything that can trigger a garbage collection, it can safely use pointers that are offset into the middle of a GCable object.

```
(offset-ptr? cptr) → boolean?
  cptr : cpointer?
```

A predicate for cpointers that have an offset, such as pointers that were created using `ptr-add`. Returns `#t` even if such an offset happens to be 0. Returns `#f` for other cpointers and non-cpointers.

```
(ptr-offset cptr) → exact-integer?
  cptr : cpointer?
```

Returns the offset of a pointer that has an offset. The resulting offset is always in bytes.

4.1 Unsafe Pointer Operations

```
(set-ptr-offset! cptr offset [ctype]) → void?
  cptr : cpointer?
  offset : exact-integer?
  ctype : ctype? = _byte
```

Sets the offset component of an offset pointer. The arguments are used in the same way as `ptr-add`. If *cptr* has no offset, the `exn:fail:contract` exception is raised.

```
(ptr-add! cptr offset [ctype]) → void?
  cptr : cpointer?
  offset : exact-integer?
  ctype : ctype? = _byte
```

Like `ptr-add`, but destructively modifies the offset contained in a pointer. The same operation could be performed using `ptr-offset` and `set-ptr-offset!`.

```
(ptr-ref cptr type [offset]) → any
  cptr : cpointer?
  type : ctype?
  offset : exact-nonnegative-integer? = 0
(ptr-ref cptr type abs-tag offset) → any
  cptr : cpointer?
  type : ctype?
  abs-tag : (one-of/c 'abs)
  offset : exact-nonnegative-integer?
(ptr-set! cptr type val) → void?
  cptr : cpointer?
  type : ctype?
  val : any/c
(ptr-set! cptr type offset val) → void?
  cptr : cpointer?
  type : ctype?
  offset : exact-nonnegative-integer?
  val : any/c
(ptr-set! cptr type abs-tag offset val) → void?
  cptr : cpointer?
  type : ctype?
  abs-tag : (one-of/c 'abs)
  offset : exact-nonnegative-integer?
  val : any/c
```

The ptr-ref procedure returns the object referenced by *cptr*, using the given *type*. The ptr-set! procedure stores the *val* in the memory *cptr* points to, using the given *type* for the conversion.

In each case, *offset* defaults to 0 (which is the only value that should be used with ffi-obj objects, see §7 "Unexported Primitive Functions"). If an *offset* index is non-0, the value is read or stored at that location, considering the pointer as a vector of *type*s — so the actual address is the pointer plus the size of *type* multiplied by *offset*. In addition, a 'abs flag can be used to use the *offset* as counting bytes rather then increments of the specified *type*.

Beware that the ptr-ref and ptr-set! procedure do not keep any meta-information on how pointers are used. It is the programmer's responsibility to use this facility only when appropriate. For example, on a little-endian machine:

```
> (define block (malloc _int 5))
> (ptr-set! block _int 0 196353)
> (map (lambda (i) (ptr-ref block _byte i)) '(0 1 2 3))
(1 255 2 0)
```

In addition, ptr-ref and ptr-set! cannot detect when offsets are beyond an object's memory bounds; out-of-bounds access can easily lead to a segmentation fault or memory corruption.

```
(memmove cptr src-cptr count [type]) → void?
  cptr : cpointer?
  src-cptr : cpointer?
  count : exact-nonnegative-integer?
  type : ctype? = _byte
(memmove cptr offset src-cptr count [type]) → void?
  cptr : cpointer?
  offset : exact-integer?
  src-cptr : cpointer?
  count : exact-nonnegative-integer?
  type : ctype? = _byte
(memmove  cptr
          offset
          src-cptr
          src-offset
          count
         [type])      → void?
  cptr : cpointer?
  offset : exact-integer?
  src-cptr : cpointer?
  src-offset : exact-integer?
  count : exact-nonnegative-integer?
```

```
  type : ctype? = _byte
```

Copies to *cptr* from *src-cptr*. The destination pointer can be offset by an optional *offset*, which is in *type* instances. The source pointer can be similarly offset by *src-offset*. The number of bytes copied from source to destination is determined by *count*, which is in *type* instances when supplied.

```
(memcpy cptr src-cptr count [type]) → void?
  cptr : cpointer?
  src-cptr : cpointer?
  count : exact-nonnegative-integer?
  type : ctype? = _byte
(memcpy cptr offset src-cptr count [type]) → void?
  cptr : cpointer?
  offset : exact-integer?
  src-cptr : cpointer?
  count : exact-nonnegative-integer?
  type : ctype? = _byte
(memcpy  cptr
         offset
         src-cptr
         src-offset
         count
        [type])      → void?
  cptr : cpointer?
  offset : exact-integer?
  src-cptr : cpointer?
  src-offset : exact-integer?
  count : exact-nonnegative-integer?
  type : ctype? = _byte
```

Like memmove, but the result is undefined if the destination and source overlap.

```
(memset cptr byte count [type]) → void?
  cptr : cpointer?
  byte : byte?
  count : exact-nonnegative-integer?
  type : ctype? = _byte
(memset cptr offset byte count [type]) → void?
  cptr : cpointer?
  offset : exact-integer?
  byte : byte?
  count : exact-nonnegative-integer?
  type : ctype? = _byte
```

Similar to memmove, but the destination is uniformly filled with byte (i.e., an exact integer between 0 and 255 inclusive).

```
(cpointer-tag cptr) → any
  cptr : cpointer?
```

Returns the Scheme object that is the tag of the given cptr pointer.

```
(set-cpointer-tag! cptr tag) → void?
  cptr : cpointer?
  tag : any/c
```

Sets the tag of the given cptr. The tag argument can be any arbitrary value; other pointer operations ignore it. When a cpointer value is printed, its tag is shown if it is a symbol, a byte string, a string. In addition, if the tag is a pair holding one of these in its car, the car is shown (so that the tag can contain other information).

4.2 Unsafe Memory Management

For general information on C-level memory management with PLT Scheme, see § "**Inside**: PLT Scheme C API".

```
(malloc  bytes-or-type
        [type-or-bytes
         cptr
         mode
         fail-mode])    → cpointer?
  bytes-or-type : (or/c exact-nonnegative-integer? ctype?)
  type-or-bytes :  (or/c exact-nonnegative-integer? ctype?)
                = absent
  cptr : cpointer? = absent
  mode : (one-of/c 'nonatomic 'stubborn 'uncollectable = absent
                   'eternal 'interior 'atomic-interior
                   'raw)
  fail-mode : (one-of/c 'failok) = absent
```

Allocates a memory block of a specified size using a specified allocation. The result is a cpointer to the allocated memory. Although not reflected above, the four arguments can appear in any order since they are all different types of Scheme objects; a size specification is required at minimum:

- If a C type bytes-or-type is given, its size is used to the block allocation size.

- If an integer *bytes-or-type* is given, it specifies the required size in bytes.
- If both *bytes-or-type* and *type-or-bytes* are given, then the allocated size is for a vector of values (the multiplication of the size of the C type and the integer).
- If a *cptr* pointer is given, its content is copied to the new block.
- A symbol *mode* argument can be given, which specifies what allocation function to use. It should be one of `'nonatomic` (uses `scheme_malloc` from PLT Scheme's C API), `'atomic` (`scheme_malloc_atomic`), `'stubborn` (`scheme_malloc_stubborn`), `'uncollectable` (`scheme_malloc_uncollectable`), `'eternal` (`scheme_malloc_eternal`), `'interior` (`scheme_malloc_allow_interior`), `'atomic-interior` (`scheme_malloc_atomic_allow_interior`), or `'raw` (uses the operating system's `malloc`, creating a GC-invisible block).
- If an additional `'failok` flag is given, then `scheme_malloc_fail_ok` is used to wrap the call.

If no mode is specified, then `'nonatomic` allocation is used when the type is any pointer-based type, and `'atomic` allocation is used otherwise.

(free *cptr*) → void
  cptr : cpointer?

Uses the operating system's `free` function for `'raw`-allocated pointers, and for pointers that a foreign library allocated and we should free. Note that this is useful as part of a finalizer (see below) procedure hook (e.g., on the Scheme pointer object, freeing the memory when the pointer object is collected, but beware of aliasing).

(end-stubborn-change *cptr*) → void?
  cptr : cpointer?

Uses `scheme_end_stubborn_change` on the given stubborn-allocated pointer.

(malloc-immobile-cell *v*) → cpointer?
  v : any/c

Allocates memory large enough to hold one arbitrary (collectable) Scheme value, but that is not itself collectable or moved by the memory manager. The cell is initialized with *v*; use the type `_scheme` with `ptr-ref` and `ptr-set!` to get or set the cell's value. The cell must be explicitly freed with `free-immobile-cell`.

(free-immobile-cell *cptr*) → void?

```
  cptr : cpointer?
```

Frees an immobile cell created by `malloc-immobile-cell`.

```
(register-finalizer obj finalizer) → void?
  obj : any/c
  finalizer : (any/c . -> . any)
```

Registers a finalizer procedure `finalizer-proc` with the given *obj*, which can be any Scheme (GC-able) object. The finalizer is registered with a will executor; see `make-will-executor`. The finalizer is invoked when *obj* is about to be collected. (This is done by a thread that is in charge of triggering these will executors.)

Finalizers are mostly intended to be used with cpointer objects (for freeing unused memory that is not under GC control), but it can be used with any Scheme object—even ones that have nothing to do with foreign code. Note, however, that the finalizer is registered for the *Scheme* object. If you intend to free a pointer object, then you must be careful to not register finalizers for two cpointers that point to the same address. Also, be careful to not make the finalizer a closure that holds on to the object.

For example, suppose that you're dealing with a foreign function that returns a C string that you should free. Here is an attempt at creating a suitable type:

```
(define bytes/free
  (make-ctype _pointer
              #f ; a Scheme bytes can be used as a pointer
              (lambda (x)
                (let ([b (make-byte-string x)])
                  (register-finalizer x free)
                  b))))
```

The above code is wrong: the finalizer is registered for `x`, which is no longer needed once the byte string is created. Changing this to register the finalizer for `b` correct this problem, but then `free` will be invoked on it instead of on `x`. In an attempt to fix this, we will be careful and print out a message for debugging:

```
(define bytes/free
  (make-ctype _pointer
              #f ; a Scheme bytes can be used as a pointer
              (lambda (x)
                (let ([b (make-byte-string x)])
                  (register-finalizer b
                    (lambda (ignored)
                      (printf "Releasing ~s\n" b)
                      (free x)))
                  b))))
```

but we never see any printout. The problem is that the finalizer is a closure that keeps a reference to `b`. To fix this, you should use the input argument to the finalizer. Simply changing `ignored` to `b` will solve this problem. (Removing the debugging message also avoids the problem, since the finalization procedure would then not close over `b`.)

`(make-sized-byte-string` *cptr length*`)` → `bytes?`
  cptr : `cpointer?`
  length : `exact-nonnegative-integer?`

Returns a byte string made of the given pointer and the given length. No copying is done. This can be used as an alternative to make pointer values accessible in Scheme when the size is known.

If *cptr* is an offset pointer created by `ptr-add`, the offset is immediately added to the pointer. Thus, this function cannot be used with `ptr-add` to create a substring of a Scheme byte string, because the offset pointer would be to the middle of a collectable object (which is not allowed).

5 Miscellaneous Support

```
(regexp-replaces objname substs) → string?
  objname : (or/c string? bytes? symbol?)
  substs : (listof (list regexp? string?))
```

A function that is convenient for many interfaces where the foreign library has some naming convention that you want to use in your interface as well. The *objname* argument can be any value that will be used to name the foreign object; it is first converted into a string, and then modified according to the given *substs* list in sequence, where each element in this list is a list of a regular expression and a substitution string. Usually, `regexp-replace*` is used to perform the substitution, except for cases where the regular expression begins with a `^` or ends with a `$`, in which case `regexp-replace` is used.

For example, the following makes it convenient to define Scheme bindings such as `foo-bar` for foreign names like `MyLib_foo_bar`:

```
(define mylib (ffi-lib "mylib"))
(define-syntax defmyobj
  (syntax-rules (:)
    [(_ name : type ...)
     (define name
       (get-ffi-obj
        (regexp-replaces 'name '((#rx"-" "_")
                                 (#rx"^" "MyLib_")))
        mylib (_fun type ...)))]))
(defmyobj foo-bar : _int -> _int)
```

```
(list->cblock lst type) → any
  lst : list>
  type : ctype?
```

Allocates a memory block of an appropriate size, and initializes it using values from *lst* and the given *type*. The *lst* must hold values that can all be converted to C values according to the given *type*.

```
(vector->cblock vector type) → any
  vector : any/c
  type : type?
```

Like `list->cblock`, but for Scheme vectors.

5.1 Unsafe Miscellaneous Operations

```
(cblock->list cblock type length) → list?
  cblock : any/c
  type : ctype?
  length : exact-nonnegative-integer?
```

Converts C *cblock*, which is a vector of *type*s, to a Scheme list. The arguments are the same as in the `list->cblock`. The *length* must be specified because there is no way to know where the block ends.

```
(cblock->vector cblock type length) → vector?
  cblock : any/c
  type : ctype?
  length : exact-nonnegative-integer?
```

Like `cblock->vector`, but for Scheme vectors.

6 Derived Utilities

6.1 Tagged C Pointer Types

The unsafe cpointer-has-tag? and cpointer-push-tag! operations manage tags to distinguish pointer types.

```
(_cpointer  tag
            [ptr-type
             scheme-to-c
             c-to-scheme]) → ctype
  tag : any/c
  ptr-type : ctype? = _pointer
  scheme-to-c : (any/c . -> . any/c) = values
  c-to-scheme : (any/c . -> . any/c) = values
(_cpointer/null  tag
                 [ptr-type
                  scheme-to-c
                  c-to-scheme]) → ctype
  tag : any/c
  ptr-type : ctype? = _pointer
  scheme-to-c : (any/c . -> . any/c) = values
  c-to-scheme : (any/c . -> . any/c) = values
```

Construct a kind of a pointer that gets a specific tag when converted to Scheme, and accept only such tagged pointers when going to C. An optional *ptr-type* can be given to be used as the base pointer type, instead of _pointer.

Pointer tags are checked with cpointer-has-tag? and changed with cpointer-push-tag! which means that other tags are preserved. Specifically, if a base *ptr-type* is given and is itself a _cpointer, then the new type will handle pointers that have the new tag in addition to *ptr-type*'s tag(s). When the tag is a pair, its first value is used for printing, so the most recently pushed tag which corresponds to the inheriting type will be displayed.

Note that tags are compared with eq? (or memq), which means an interface can hide its value from users (e.g., not provide the cpointer-tag accessor), which makes such pointers unfake-able.

_cpointer/null is similar to _cpointer except that it tolerates NULL pointers both going to C and back. Note that NULL pointers are represented as #f in Scheme, so they are not tagged.

```
(define-cpointer-type _id)
(define-cpointer-type _id scheme-to-c-expr)
```

```
(define-cpointer-type _id scheme-to-c-expr c-to-scheme-expr)
```

A macro version of `_cpointer` and `_cpointer/null`, using the defined name for a tag string, and defining a predicate too. The *_id* must start with `_`.

The optional expression produces optional arguments to `_cpointer`.

In addition to defining *_id* to a type generated by `_cpointer`, *_id*`/null` is bound to a type produced by `_cpointer/null` type. Finally, *id*`?` is defined as a predicate, and *id*`-tag` is defined as an accessor to obtain a tag. The tag is the string form of *id*.

6.1.1 Unsafe Tagged C Pointer Functions

```
(cpointer-has-tag? cptr tag) → boolean?
  cptr : any/c
  tag : any/c
(cpointer-push-tag! cptr tag) → void
  cptr : any/c
  tag : any/c
```

These two functions treat pointer tags as lists of tags. As described in §4 "Pointer Functions", a pointer tag does not have any role, except for Scheme code that uses it to distinguish pointers; these functions treat the tag value as a list of tags, which makes it possible to construct pointer types that can be treated as other pointer types, mainly for implementing inheritance via upcasts (when a struct contains a super struct as its first element).

The `cpointer-has-tag?` function checks whether if the given *cptr* has the *tag*. A pointer has a tag *tag* when its tag is either `eq?` to *tag* or a list that contains (in the sense of `memq`) *tag*.

The `cpointer-push-tag!` function pushes the given *tag* value on *cptr*'s tags. The main properties of this operation are: (a) pushing any tag will make later calls to `cpointer-has-tag?` succeed with this tag, and (b) the pushed tag will be used when printing the pointer (until a new value is pushed). Technically, pushing a tag will simply set it if there is no tag set, otherwise push it on an existing list or an existing value (treated as a single-element list).

6.2 Safe C Vectors

The `cvector` form can be used as a type C vectors (i.e., a pointer to a memory block).

```
(make-cvector type length) → cvector?
  type : ctype?
  length : exact-nonnegative-integer?
```

Allocates a C vector using the given *type* and *length*.

```
(cvector type val ...) → cvector?
  type : ctype?
  val : any/c
```

Creates a C vector of the given *type*, initialized to the given list of *val*s.

```
(cvector? v) → boolean?
  v : any/c
```

Returns #t if *v* is a C vector, #f otherwise.

```
(cvector-length cvec) → exact-nonnegative-integer?
  cvec : cvector?
```

Returns the length of a C vector.

```
(cvector-type cvec) → ctype?
  cvec : cvector?
```

Returns the C type object of a C vector.

```
(cvector-ptr cvec) → cpointer?
  cvec : cvector?
```

Returns the pointer that points at the beginning block of the given C vector.

```
(cvector-ref cvec k) → any
  cvec : cvector?
  k : exact-nonnegative-integer?
```

References the *k*th element of the *cvec* C vector. The result has the type that the C vector uses.

```
(cvector-set! cvec k val) → void?
  cvec : cvector?
  k : exact-nonnegative-integer?
  val : any
```

Sets the *k*th element of the *cvec* C vector to *val*. The *val* argument should be a value that can be used with the type that the C vector uses.

```
(cvector->list cvec) → list?
  cvec : cvector?
```

Converts the *cvec* C vector object to a list of values.

```
(list->cvector lst type) → cvector?
  lst : list?
  type : ctype?
```

Converts the list *lst* to a C vector of the given *type*.

6.2.1 Unsafe C Vector Construction

```
(make-cvector* cptr type length) → cvector?
  cptr : any/c
  type : ctype?
  length : exact-nonnegative-integer?
```

Constructs a C vector using an existing pointer object. This operation is not safe, so it is intended to be used in specific situations where the *type* and *length* are known.

6.3 SRFI-4 Vectors

SRFI-4 vectors are similar to C vectors (see §6.2 "Safe C Vectors"), except that they define different types of vectors, each with a hard-wired type.

An exception is the u8 family of bindings, which are just aliases for byte-string bindings: make-u8vector, u8vector, u8vector?, u8vector-length, u8vector-ref, u8vector-set!, list->u8vector, u8vector->list.

```
(make-u8vector len) → u8vector?
  len : exact-nonnegative-integer?
(u8vector val ...) → u8vector?
  val : number?
(u8vector? v) → boolean?
  v : any/c
(u8vector-length vec) → exact-nonnegative-integer?
  vec : u8vector?
(u8vector-ref vec k) → number?
  vec : u8vector?
```

```
  k : exact-nonnegative-integer?
(u8vector-set! vec k val) → void?
  vec : u8vector?
  k : exact-nonnegative-integer?
  val : number?
(list->u8vector lst) → u8vector?
  lst : (listof number?)
(u8vector->list vec) → (listof number?)
  vec : u8vector?
```

Like `_cvector`, but for vectors of `_byte` elements. These are aliases for `byte` operations.

```
(_u8vector mode maybe-len)
_u8vector
```

Like `_cvector`, but for vectors of `_uint8` elements.

```
(make-s8vector len) → s8vector?
  len : exact-nonnegative-integer?
(s8vector val ...) → s8vector?
  val : number?
(s8vector? v) → boolean?
  v : any/c
(s8vector-length vec) → exact-nonnegative-integer?
  vec : s8vector?
(s8vector-ref vec k) → number?
  vec : s8vector?
  k : exact-nonnegative-integer?
(s8vector-set! vec k val) → void?
  vec : s8vector?
  k : exact-nonnegative-integer?
  val : number?
(list->s8vector lst) → s8vector?
  lst : (listof number?)
(s8vector->list vec) → (listof number?)
  vec : s8vector?
```

Like `make-vector`, etc., but for `_int8` elements.

```
(_s8vector mode maybe-len)
_s8vector
```

Like `_cvector`, but for vectors of `_int8` elements.

```
(make-s16vector len) → s16vector?
  len : exact-nonnegative-integer?
(s16vector val ...) → s16vector?
  val : number?
(s16vector? v) → boolean?
  v : any/c
(s16vector-length vec) → exact-nonnegative-integer?
  vec : s16vector?
(s16vector-ref vec k) → number?
  vec : s16vector?
  k : exact-nonnegative-integer?
(s16vector-set! vec k val) → void?
  vec : s16vector?
  k : exact-nonnegative-integer?
  val : number?
(list->s16vector lst) → s16vector?
  lst : (listof number?)
(s16vector->list vec) → (listof number?)
  vec : s16vector?
```

Like `make-vector`, etc., but for `_int16` elements.

```
(_s16vector mode maybe-len)
_s16vector
```

Like `_cvector`, but for vectors of `_int16` elements.

```
(make-u16vector len) → u16vector?
  len : exact-nonnegative-integer?
(u16vector val ...) → u16vector?
  val : number?
(u16vector? v) → boolean?
  v : any/c
(u16vector-length vec) → exact-nonnegative-integer?
  vec : u16vector?
(u16vector-ref vec k) → number?
  vec : u16vector?
  k : exact-nonnegative-integer?
(u16vector-set! vec k val) → void?
  vec : u16vector?
  k : exact-nonnegative-integer?
  val : number?
(list->u16vector lst) → u16vector?
  lst : (listof number?)
```

```
(u16vector->list vec) → (listof number?)
  vec : u16vector?
```

Like make-vector, etc., but for _uint16 elements.

```
(_u16vector mode maybe-len)
_u16vector
```

Like _cvector, but for vectors of _uint16 elements.

```
(make-s32vector len) → s32vector?
  len : exact-nonnegative-integer?
(s32vector val ...) → s32vector?
  val : number?
(s32vector? v) → boolean?
  v : any/c
(s32vector-length vec) → exact-nonnegative-integer?
  vec : s32vector?
(s32vector-ref vec k) → number?
  vec : s32vector?
  k : exact-nonnegative-integer?
(s32vector-set! vec k val) → void?
  vec : s32vector?
  k : exact-nonnegative-integer?
  val : number?
(list->s32vector lst) → s32vector?
  lst : (listof number?)
(s32vector->list vec) → (listof number?)
  vec : s32vector?
```

Like make-vector, etc., but for _int32 elements.

```
(_s32vector mode maybe-len)
_s32vector
```

Like _cvector, but for vectors of _int32 elements.

```
(make-u32vector len) → u32vector?
  len : exact-nonnegative-integer?
(u32vector val ...) → u32vector?
  val : number?
(u32vector? v) → boolean?
  v : any/c
(u32vector-length vec) → exact-nonnegative-integer?
```

```
  vec : u32vector?
(u32vector-ref vec k) → number?
  vec : u32vector?
  k : exact-nonnegative-integer?
(u32vector-set! vec k val) → void?
  vec : u32vector?
  k : exact-nonnegative-integer?
  val : number?
(list->u32vector lst) → u32vector?
  lst : (listof number?)
(u32vector->list vec) → (listof number?)
  vec : u32vector?
```

Like `make-vector`, etc., but for `_uint32` elements.

```
(_u32vector mode maybe-len)
_u32vector
```

Like `_cvector`, but for vectors of `_uint32` elements.

```
(make-s64vector len) → s64vector?
  len : exact-nonnegative-integer?
(s64vector val ...) → s64vector?
  val : number?
(s64vector? v) → boolean?
  v : any/c
(s64vector-length vec) → exact-nonnegative-integer?
  vec : s64vector?
(s64vector-ref vec k) → number?
  vec : s64vector?
  k : exact-nonnegative-integer?
(s64vector-set! vec k val) → void?
  vec : s64vector?
  k : exact-nonnegative-integer?
  val : number?
(list->s64vector lst) → s64vector?
  lst : (listof number?)
(s64vector->list vec) → (listof number?)
  vec : s64vector?
```

Like `make-vector`, etc., but for `_int64` elements.

```
(_s64vector mode maybe-len)
_s64vector
```

Like _cvector, but for vectors of _int64 elements.

```
(make-u64vector len) → u64vector?
  len : exact-nonnegative-integer?
(u64vector val ...) → u64vector?
  val : number?
(u64vector? v) → boolean?
  v : any/c
(u64vector-length vec) → exact-nonnegative-integer?
  vec : u64vector?
(u64vector-ref vec k) → number?
  vec : u64vector?
  k : exact-nonnegative-integer?
(u64vector-set! vec k val) → void?
  vec : u64vector?
  k : exact-nonnegative-integer?
  val : number?
(list->u64vector lst) → u64vector?
  lst : (listof number?)
(u64vector->list vec) → (listof number?)
  vec : u64vector?
```

Like make-vector, etc., but for _uint64 elements.

```
(_u64vector mode maybe-len)
_u64vector
```

Like _cvector, but for vectors of _uint64 elements.

```
(make-f32vector len) → f32vector?
  len : exact-nonnegative-integer?
(f32vector val ...) → f32vector?
  val : number?
(f32vector? v) → boolean?
  v : any/c
(f32vector-length vec) → exact-nonnegative-integer?
  vec : f32vector?
(f32vector-ref vec k) → number?
  vec : f32vector?
  k : exact-nonnegative-integer?
(f32vector-set! vec k val) → void?
  vec : f32vector?
  k : exact-nonnegative-integer?
  val : number?
```

```
(list->f32vector lst) → f32vector?
  lst : (listof number?)
(f32vector->list vec) → (listof number?)
  vec : f32vector?
```

Like `make-vector`, etc., but for `_float` elements.

```
(_f32vector mode maybe-len)
_f32vector
```

Like `_cvector`, but for vectors of `_float` elements.

```
(make-f64vector len) → f64vector?
  len : exact-nonnegative-integer?
(f64vector val ...) → f64vector?
  val : number?
(f64vector? v) → boolean?
  v : any/c
(f64vector-length vec) → exact-nonnegative-integer?
  vec : f64vector?
(f64vector-ref vec k) → number?
  vec : f64vector?
  k : exact-nonnegative-integer?
(f64vector-set! vec k val) → void?
  vec : f64vector?
  k : exact-nonnegative-integer?
  val : number?
(list->f64vector lst) → f64vector?
  lst : (listof number?)
(f64vector->list vec) → (listof number?)
  vec : f64vector?
```

Like `make-vector`, etc., but for `_double*` elements.

```
(_f64vector mode maybe-len)
_f64vector
```

Like `_cvector`, but for vectors of `_double*` elements.

7 Unexported Primitive Functions

Parts of the `scheme/foreign` library are implemented by the MzScheme built-in `'#%foreign` module. The `'#%foreign` module is not intended for direct use, but it exports the following procedures. If you find any of these useful, please let us know.

```
(ffi-obj objname lib) → any
  objname : (or/c string? bytes? symbol?)
  lib : (or/c ffi-lib? path-string? #f)
```

Pulls out a foreign object from a library, returning a Scheme value that can be used as a pointer. If a name is provided instead of a foreign-library value, `ffi-lib` is used to create a library object.

```
(ffi-obj? x) → boolean?
  x : any/c
(ffi-obj-lib obj) → ffi-lib?
  obj : ffi-obj?
(ffi-obj-name obj) → string?
  obj : ffi-obj?
```

A predicate for objects returned by `ffi-obj`, and accessor functions that return its corresponding library object and name. These values can also be used as C pointer objects.

```
(ctype-basetype type) → (or/c ctype? #f)
  type : ctype?
(ctype-scheme->c type) → procedure?
  type : ctype?
(ctype-c->scheme type) → procedure?
  type : ctype?
```

Accessors for the components of a C type object, made by `make-ctype`. The `ctype-basetype` selector returns a symbol for primitive types that names the type, a list of ctypes for cstructs, and another ctype for user-defined ctypes.

```
(ffi-call ptr in-types out-type [abi]) → any
  ptr : any/c
  in-types : (listof ctype?)
  out-type : ctype?
  abi : (or/c symbol/c #f) = #f
```

The primitive mechanism that creates Scheme "callout" values. The given *ptr* (any pointer value, including `ffi-obj` values) is wrapped in a Scheme-callable primitive function that

uses the types to specify how values are marshaled.

The optional *abi* argument determines the foreign ABI that is used. `#f` or `'default` will use a platform-dependent default; other possible values are `'stdcall` and `'sysv` (the latter corresponds to "cdecl"). This is especially important on Windows, where most system functions are `'stdcall`, which is not the default.

```
(ffi-callback  proc
               in-types
               out-type
              [abi
               atomic?]) → ffi-callback?
  proc : any/c
  in-types : any/c
  out-type : any/c
  abi : (or/c symbol/c #f) = #f
  atomic? : any/c = #f
```

The symmetric counterpart of `ffi-call`. It receives a Scheme procedure and creates a callback object, which can also be used as a pointer. This object can be used as a C-callable function, which invokes *proc* using the types to specify how values are marshaled.

```
(ffi-callback? x) → boolean?
  x : any/c
```

A predicate for callback values that are created by `ffi-callback`.

8 Macros for Unsafety

`(unsafe!)`

Makes most of the bindings documented in this module available. See §1 “Overview” for information on why this declaration is required.

```
(provide* provide-star-spec ...)

provide-star-spec = (unsafe id)
                  | (unsafe (rename-out [id external-id]))
                  | provide-spec
```

Like `provide`, but *id*s under `unsafe` are not actually provided. Instead, they are collected for introduction into an importing module via a macro created by `define-unsafer`.

Providing users with unsafe operations without using this facility should be considered a bug in your code.

`(define-unsafer id)`

Cooperates with `provide*` to define *id* as a `unsafe!`-like form that introduces definitions for each binding provided as `unsafe`. The `define-unsafer` form must occur after all the `provide*` forms to which it refers.

Index