PLT Foreign Interface Manual

PLT (scheme@plt-scheme.org)

301
Released December 2006

Copyright notice

Contents

1. Introduction

MzScheme has includes functionality that makes it possible to interact with foreign data and foreign function, available through the **foreign.ss** module in MzLib. Traditional Scheme code is usually safe: if the process crashes the blame is always on C code. Using C extensions makes it possible to extend Scheme — the added code can lead to crashes, but the blame for crashes is still at the C side.

The foreign capabilities of MzScheme makes it possible to use Scheme in situation that would otherwise require writing a C extension, so Scheme code can be used to substitute C. Doing this is, obviously, as unsafe as writing C — so a crash can be now blamed on either C or Scheme code which interacts with the foreign world. To make things a bit safer, the **foreign.ss** module will not provide you with any of the unsafe operations by default. Instead, you get an '`unsafe!`' macro, which you can use to make the unsafe operations available[1]. 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. By this we constrain the blame for crashes to either C code or such unsafe Scheme code.

In rare cases you might want to provide an unsafe interface, for example, if you write a module that provides more foreign-interaction functionality. In these cases you should use the '`provide*`' macro with '`unsafe`' bindings, and the '`define-unsafer`' to provide an `unsafe!`-like macro that will make these bindings available. See the **foreign.ss** source for details. Providing users with unsafe operations without this facility is considered a bug in your code.

In addition to this manual, *Inside PLT MzScheme* can be used to learn about the internals of MzScheme. Note that using `ffi-lib` with `#f` instead of a file name will use the running process as a library, making all C-level functions (including the standard library) available.

[1] For additional safety, the `unsafe!` is protected (see §9.4 in *PLT MzScheme: Language Manual*).

2. Foreign Interface

This is a description of the foreign interface. The interface has some parts implemented in C (**plt/src/foreign/foreign.c**) which is available as a built-in `#%foreign` module. This module is not intended for general use as is, and further documentation can be found in the source. The relevant functionality is provided via the `foreign` module in MzLib (**foreign.ss**).

For examples of common usage patterns, see the defined interfaces in the **ffi** collection.

2.1 Basic C Type Functions

C types are the main concept of the foreign interface (C is just used for conventions). They can roughly be divided into primitive types and user-defined types. The foreign interface deals with primitive types internally, converting them to/from C types. A user type is one that is made on top of existing primitive types (or other user types), by supplying an existing type, and conversion functions that go from Scheme to that type and from that type to Scheme ('Scheme' is misleading here: the translation is from some Scheme objects to other objects that the system already knows how to translate).

`(make-ctype` *`ctype scheme-to-C-proc C-to-scheme-proc`*`)` PROCEDURE

Create a new C type object, with the given conversions functions. The conversion functions can be `#f` meaning that there is no conversion for the corresponding direction. If both functions are `#f`, *`ctype`* is returned.

`(ctype?` *`ctype`*`)` PROCEDURE

A predicate for C type objects (both primitive and user-defined).

`(ctype-sizeof` *`ctype`*`)` PROCEDURE
`(ctype-alignof` *`ctype`*`)` PROCEDURE

Functions that return the platform-dependent size and alignment of a given *`ctype`*.

`(compiler-sizeof` *`symbols`*`)` PROCEDURE

This procedure expects a description of a C type, which is a symbol that names a type. Possible values are `'int`, `'char`, `'short`, `'long`, `'*`, `'void`, `'float`, `'double`. It uses the C `sizeof` operator to return the size of the named C type directly. It should be used to gather information about the current platform, for example, it is used to define alias types to ones with know size (e.g., binding `_int` to `_int32`).

2.2 Simple Types

2.2.1 Numeric Types

There are basic integer types at various sizes. These are: `_int8`, `_sint8`, `_uint8`, `_int16`, `_sint16`, `_uint16`, `_int32`, `_sint32`, `_uint32`, `_int64`, `_sint64`, and `_uint64`. The 's' or 'u' prefix specifies a signed or an unsigned integer respectively; the ones with no prefix are signed.

In addition, there are several type 'aliases' (extra bindings for some of the above types):

- `_byte`, `_ubyte` and `_sbyte`: aliases for `_uint8` and `_sint8` (`_byte` is unsigned),
- `_word`, `_uword` and `_sword`: aliases for `_uint16` and `_sint16` (`_word` is unsigned),
- `_short`, `_ushort` and `_sshort`: aliases for the integer type that correspond to the platform's `short` type (`_short` is signed),
- `_int`, `_uint` and `_sint`: aliases for the integer type that correspond to the platform's `int` type (`_int` is signed),
- `_long`, `_ulong` and `_slong`: aliases for the integer type that correspond to the platform's `long` type (`_long` is signed),

In cases where speed matters, and you know that the integer is small enough, you can use the `_fixnum` and `_ufixnum`, which are similar to `_long` and `_ulong` but assume that the quantities fit in MzScheme's immediate integers (not bignums). If you need this but you want to be sure that the C level integer is an integer (32 bit) size, then use `_fixint` and `_ufixint`.

Finally, there are two floating point types, `_float` and `_double` for the corresponding C types; and `_double*` that implicitly coerces any non-complex number to a C double.

2.2.2 String Types

2.2.2.1 Primitive String Types

There are several built-in primitive string types.

`_bytes` C Type

This is a type for Scheme byte-string, which corresponds to C's `char*` type. In addition to translating byte strings, `#f` corresponds to the `NULL` pointer. Note that this is also a custom-type macro, see section 2.8.1 below.

`_string/ucs-4` C Type

A type for MzScheme's native Unicode strings, in UCS-4 format. These correspond to the C `mzchar*` type used by MzScheme.

`_string/utf-16` C Type

Unicode strings in UTF-16 format.

`_path` C Type

Simple `char*` strings, corresponding to MzScheme's path strings.

`_symbol` C TYPE

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

2.2.2.2 ADDITIONAL STRING TYPES

In addition to the above string types, there are several bytes-based types that are provided. `_string/utf-8`, `_string/locale`, and `_string/latin-1` are defined using the corresponding built-in conversion procedures from strings (they do treat `#f` as `NULL`). `_string*/utf-8`, `_string*/locale`, and `_string*/latin-1` are similar but accept a wider range of values: byte strings are passed as is, and paths are converted using `path->bytes`.

2.2.2.3 THE `_string` TYPE

The `_string/ucs-4` type is rarely useful when interacting with foreign code, but using `_bytes` is somewhat unnatural since it forces users to use byte strings. The solution is the `_string` 'type'.

`_string` C TYPE

This is actually a syntax, which expands to a usage of the `default-_string-type` parameter. Note that the parameter is used when this identifier is evaluated, meaning that the parameter should be set before interface definition are being made (which is when the types are used).

(It is not difficult to create a type that uses a customizable conversion, but this means that all usages of this type will always use the same conversion, and that the result will take one more step when executing.)

`(default-_string-type` [*ctype*]`)` PROCEDURE

This is a parameter that determines what is the current `_string` type. It is initially set to `_string/*utf-8`. If you need to change it, make sure you do so *before* interfaces are defined.

2.2.2.4 UTILITY STRING TYPES

`_file` C TYPE

This is just like `_path`, but when values go from Scheme to C, `expand-path` is used on the given value. As an output value, it is identical to `_path`.

`_string/eof` C TYPE

This is similar to the `_string` type (it is a syntax for the same reason), except that a foreign return value of `NULL` (`#f`) is translated to a Scheme `eof` value.

2.2.3 Pointer Types

`_pointer` C TYPE

This type corresponds to Scheme 'C pointer' objects. These pointers can have an arbitrary Scheme object attached to them as a type tag. The tag is ignored by built-in functionality, it is intended to be used by interfaces. See section 2.6 for creating pointer types that use these tags for safety.

`_scheme` C TYPE

This type can be used with any Scheme object, it corresponds to MzScheme's `Scheme_Object*` type. It is useful only for libraries that are aware of MzScheme's objects.

`_fpointer` C TYPE

This is similar to `_pointer`, except that it should be used with function pointers. Using these pointers avoids one dereferencing, which is the proper way of dealing with function pointers. This type should be used only in rare situations where you need to pass a foreign function pointer to a foreign function — Using a `_cprocedure` type is possible for such situations, but extremely inefficient as every call will go through Scheme. Otherwise, `_cprocedure` should be used (it is based on `_fpointer`).

2.2.4 Type Constructors

Since types are first-class values, there are several type constructors that build type objects. These are just the simple ones, more constructors are described below. Note that they are listed as procedures, but implemented as syntax, so it is possible to use a name where the syntactical context implies one (they can also be used as values, but error messages will not have a meaningful name in this case).

`(_enum` *`symbols`* [*`basetype`*]`)` PROCEDURE

This procedure takes in a list of symbols and generates an enumeration type. The enumeration maps between the given *`symbols`* and integers, counting from 0. The list of *`symbols`* can also set the values of symbols, if a symbol is followed by a '`=`' symbol and an integer, not in nested form to make it easy to copy, paste & modify C code. For example, the list `'(x y = 10 z)` maps `'x` to 0, `'y` to 10, and `'z` to 11.

The optional *`basetype`* argument can specify the base type to use, defaulting to `_ufixint`.

`(_bitmask` *`symbols`* [*`basetype`*]`)` PROCEDURE

This is similar to `_enum`, but the resulting mapping translates a list of symbols to a number and back, using a logical or. A single symbol can be given as an input to make things a little more convenient. The default *`basetype`* for this is `_uint`, since high bits are often used for flags.

2.2.5 Misc Types

`_bool` C TYPE

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

`_void` C TYPE

This type indicates a Scheme void return value, and it cannot be used to translate values to C (i.e., this type cannot be used for function inputs).

2.3 Pointer Functions

`(cpointer? x)` PROCEDURE

A predicate for pointer values: returns `#t` for C pointer objects as well as other values that can be used as pointers — `#f` (used as a NULL pointer), byte strings (used as memory blocks), and some additional internal objects (`ffi-obj`s and callbacks, see section 2.10). `#f` for other values.

`(ptr-ref cptr ctype [['abs] offset])` PROCEDURE

This function returns the object referenced by *`cptr`*, using the given *`ctype`*.

`(ptr-set! cptr ctype [['abs] offset] value)` PROCEDURE

This function stores the *`value`* in the memory *`cptr`* points to, using the given *`ctype`* for the conversion. Returns void.

`offset` defaults to 0 (which is the only value that should be used with `ffi-obj` objects, see section 2.10). If an *`offset`* index is given, the value is stored at that location, considering the pointer as a vector of *`ctype`*s — so the actual address is the pointer plus the size of *`ctype`* multiplied by *`offset`*. In addition, a *`'abs`* flag can be used to use the *`offset`* as counting bytes rather then increments of the specified *`ctype`*.

Caution: the `ptr-ref` and `ptr-set!` operations do not keep any meta-information on how pointers are used. It is the programmers 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, there is no way to detect when offsets beyond the size of the block are used, so segmentation faults are easy to get.

`(ptr-equal? cptr1 cptr2)` PROCEDURE

Compares the values of the two pointers. (Note that two different Scheme pointer objects can contain the same pointer.)

`(cpointer-tag cptr)` PROCEDURE

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

`(set-cpointer-tag! cptr tag)` PROCEDURE

Sets the tag of the given *`cptr`*. *`tag`* can be any arbitrary value, it is intended to be useful for users, and otherwise it is ignored. When a cpointer value is printed, its tag is shown if it's a symbol, a byte string, a string, or when it's a pair holding one of these in its car then that is shown. The reason for printing the car of a pair tag is to make it possible to use tags that contain more information, e.g., as used by `cpointer-has-tag?` and `cpointer-push-tag!`.

2.3.1 Memory Management

(malloc *bytes-or-type* [*type-or-bytes*] [*cptr*] [*mode*] [*'fail-ok*]) PROCEDURE

This function allocates a memory block of a specified size. Returns a `cpointer` to that location in memory. The four arguments can appear in any order since they are all different types of Scheme objects, at the least, a size specification is required:

- 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 contents is copied to the new block, it is expected to be able to do so.
- A symbol *mode* argument can be given, which specifies what allocation function to use. It should be one of `'nonatomic` (uses `scheme_malloc`), `'atomic` (`scheme_malloc_atomic`), `'stubborn` (`scheme_malloc_stubborn`), `'uncollectable` (`scheme_malloc_uncollectable`), `'eternal` (`scheme_malloc_eternal`), 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 will be used when the type is any pointer-based type, otherwise, an `'atomic` allocation is used. Note that raw allocations is sometimes needed when dealing with memory management issues (usually with the precise GC).

(free *cpointer*) PROCEDURE

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 *cpointer*) PROCEDURE

Uses `scheme_end_stubborn_change` on the given stubborn-allocated pointer (see the MzScheme documentation).

(register-finalizer *obj finalizer-proc*) PROCEDURE

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 §13.3 in *PLT MzScheme: Language Manual*); it is invoked when *obj* is about to be collected. (This is done by a thread that is in charge of triggering these will executors.)

This is 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 have to be careful to not register finalizers for two cpointers that point to the same address.

(make-sized-byte-string *cptr length*) PROCEDURE

This function returns a byte string made out 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.

2.4 Safe C Vectors

`(make-cvector` *`ctype length`*`)` PROCEDURE

Creates a C vector using the given *`ctype`* and *`length`*. This will allocate a memory block for the vector.

`(cvector` *`ctype vals ⋯`*`)` PROCEDURE

Creates a C vector using the given *`ctype`*, initialized to the given list of values.

`(cvector?` *`x`*`)` PROCEDURE

Predicate for C vectors.

`(cvector-length` *`cvec`*`)` PROCEDURE

Returns the length of a C vector.

`(cvector-type` *`cvec`*`)` PROCEDURE

Returns the C type object of a C vector.

`(cvector-ref` *`cvec idx`*`)` PROCEDURE

References the *`idx`*th element of the *`cvec`* C vector. The result will have the type that the C vector uses.

`(cvector-set!` *`cvec idx val`*`)` PROCEDURE

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

`(cvector->list` *`cvec`*`)` PROCEDURE

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

`(list->cvector` *`list ctype`*`)` PROCEDURE

Converts the list *`list`* to a C vector of the given *`ctype`*.

`(make-cvector*` *`cptr ctype length`*`)` PROCEDURE

This 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 *`ctype`* and *`length`* are known.

`_cvector` C TYPE

This can be used as an input type for C vectors, which uses the pointer to the memory block. Also, see the `_cvector` custom type in section 2.8.1.

2.4.1 SRFI-4 Vectors

SRFI-4 vectors are similar to the above C vector, except it defines different types of vectors, each with a hard-wired type.

An internal '`make-srfi-4`' macro defines and provides 8 functions for each *`TAG`* type — a 'make-*`TAG`*vector' constructor, a '*`TAG`*vector' constructor (uses its arguments), a '*`TAG`*`vector?`' predicate, a '*`TAG`*`vector-length`' function, a '*`TAG`*`vector-ref`' accessor and a '*`TAG`*`vector-set!`' setter, and conversions to and from a list, '*`TAG`*`vector->list`' and '`list->`*`TAG`*`vector`'. The functions are the same as the corresponding `cvector` functions above, except that there is no type argument.

In addition, there is a `_`*`TAG`*`vector` type similar to `_cvector` that can be used when interfacing foreign functions. Just like `_cvector`, `_`*`TAG`*`vector` can be used both as a simple type to pass a pointer value to foreign code, or as a custom type, expecting a mode flag for input, output, or input-output, where output-mode requires specifying the size of the result.

These following homogeneous vector types are defined, for a total of 80 functions.

- `s8vector` using `_int8`,
- `u8vector` using `_uint8`,
- `s16vector` using `_int16`,
- `u16vector` using `_uint16`,
- `s32vector` using `_int32`,
- `u32vector` using `_uint32`,
- `s64vector` using `_int64`,
- `u64vector` using `_uint64`,
- `f32vector` using `_float`,
- `f64vector` using `_double*`.

2.5 Foreign Libraries

`(ffi-lib` *`path`* [*`version`*]`)` PROCEDURE

Opens a foreign library in an OS-specific way (using `LoadLibrary` on Windows, and `dlopen` on Unix), and returns an ffi-lib object. The path is not expected to contain the library suffix, which is added according to the current platform. If this fails, several other filename variations are tried — retrying without an automatically added suffix and using a full path of a file if it exists (`dlopen` always searches its path, unless the path is absolute).

An optional *`version`* string can be supplied, which is appended to the name.

`(ffi-lib?` *`x`*`)` PROCEDURE

A predicate for `ffi-lib` objects.

`(get-ffi-obj` *`objname lib type`* [*`failure-thunk`*]`)` PROCEDURE

Looks for the given object name *`objname`* (a string, a byte string, or a symbol), in the given *`lib`* library (a string, a library object, or `#f`). Specifying a string for the library will look for the named library (using *`ffi-lib`*), and using

`#f` will open up the executable (the MzScheme or MrEd process) as a library — using this it is possible to use internal (see *Inside PLT MzScheme*) functionality directly from Scheme. If the object is found, it is converted to Scheme using the given type. This is most often used with function types (see section 2.8).

If the object is not found, *`failure-thunk`* is used to produce a return value, or if it is not provided an exception is raised. This can be used to find a foreign function, or provide an error stub if it is not there, for example:

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

(Reminder: *`get-ffi-obj`* is an unsafe procedure, see the beginning of Chapter 1 for details.)

`(set-ffi-obj!` *`objname lib type new`*`)` PROCEDURE

Looks for the *`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 ctype`*`)` PROCEDURE

Returns a parameter-like procedure that can either reference the specified foreign value, or set it. This is useful in case Scheme code and library code interact through a library value. It can be used with any type, but it is not recommended to use this for foreign functions since each reference through this will construct the low-level interface before the actual call.

`(define-c` *`var lib type`*`)` SYNTAX

This syntax uses `make-c-parameter` above: it defines a *`var`* syntax that behaves like a Scheme binding, referencing and setting it is achieved through such a C parameter (so the same comments apply). The *`var`* part is used both for the Scheme binding and for the foreign object's name.

`(ffi-obj-ref` *`objname lib`*`)` FAILURE-THUNK procedure

Returns a pointer object for the required foreign object. This 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).

2.6 Tagged C Pointer Types

`(cpointer-has-tag?` *`cptr tag`*`)` PROCEDURE

`(cpointer-push-tag!` *`cptr tag`*`)` PROCEDURE

These two functions treat pointer tags as lists of tags. As described in Section 2.3, 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).

`cpointer-has-tag?` checks if the given *`cptr`* has the *`tag`* — a pointer has a tag *t* when its tag is either `eq?` to *t* or a list that contains (`memq`) *t*.

`cpointer-push-tag!` 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, (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).)

`(_cpointer` *`tag`* [*`ptr-type`* [*`scheme-to-C-proc C-to-scheme-proc`*]]`)` PROCEDURE

`(_cpointer/null` *`tag`* [*`ptr-type`* [*`scheme-to-C-proc C-to-scheme-proc`*]]`)` PROCEDURE

These functions constructs 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`. (See `set-cpointer-tag!` and `cpointer-tag` in section 2.3 for more details.)

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 un-fake-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` *`_name`* [*`ptr-type`* [*`scheme-to-C-proc C-to-scheme-proc`*]]`)` SYNTAX

A macro version of `_cpointer` and `_cpointer/null` above, using the defined name for a tag string, and defining a predicate too. The name should look like '`_foo`', the predicate will be '`foo?`', and the tag will be `"foo"`. In addition, '`foo-tag`' will be bound to the tag. The optional arguments are the same as those of `_cpointer`. '`_foo`' will be bound to the `_cpointer` type, and '`_foo/null`' to the `_cpointer/null` type.

2.7 C Struct Types

`(make-cstruct-type` *`ctypes`*`)` PROCEDURE

This is the primitive type constructor for creating new C struct types. These types are actually new primitive types — they don't have any 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 *`ctypes`* list. There is no other primitive support, the following wrap this in a more functional way.

`(_list-struct` *`ctypes`* $\cdots$[1]`)` PROCEDURE

This is a type constructor that builds a struct type using the above `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 needs to be allocated, and this type immediately allocates and uses a list from the allocated space, which means that using it is inefficient — use `define-cstruct` below for a more efficient approach.

`(define-cstruct _name ((fieldname ctype) ···))` SYNTAX

This macro defines a new C struct type, but unlike '`_list-struct`', the resulting type deals with C structs in binary form rather than marshal them to Scheme values. It uses a `define-struct`-like approach, 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 name must have a form of '`_foo`'. The form and the generated bindings are intentionally similar to `define-struct` only with type specification for the slots — the identifiers that will be bound as a result are:

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

Objects of this new type are actually cpointers, with a type tag that is a list that contains `"foo"`. 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 `_foo-pointer`. The `_foo` type should not be used when a pointer is expected — it will cause the struct to be copied rather than use the pointer value, leading to memory corruption.

If the first slot is itself a cstruct type, its tag will be used in addition to the new tag. This 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 slot 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 Section 2.7.1 for an example.

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 moving GC will invalidate the pointer's value). This means that only non-pointer values, and pointers to memory that is outside the GC's control can be used.

`(define-cstruct (_name _super) ((fieldname ctype) ···))` SYNTAX

This alternative form of `define-cstruct`, is shorthand for using an initial slot named `super` using *`_super`* as its type. Remember that the new struct will use *`_super`*'s tag in addition to its own tag, meaning that instances of *`_name`* can be used as instances of *`_super`*. Aside from the syntactic sugar, the constructor function will be different when this syntax is used: instead of expecting a first argument which is an instance of *`_super`*, it will expect arguments for each of *`_super`*'s slots in addition for the new slots. This is, again, in analogy to using a super-struct with `define-struct`.

2.7.1 C Struct Examples

A few examples will help understanding how to use structs. Assuming 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;
}
```

First, 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-structs 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))
  ; -> (#<cpointer:A> 1 2)
```

'gety' is also simple to use from Scheme:

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

2.8 Function Types

(_cprocedure *input-types output-type* [*wrapper-proc*]) PROCEDURE

This is a procedure type constructor: it creates a new function type, specified by the given *input-types* list and *output-type*. Usually, the _fun syntax (described below) should be used instead, since it can deal with a wide range of complicated cases.

The resulting type can be used to reference foreign functions (usually ffi-objs, 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.

This type 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, specifically, its lexical context is properly preserved.

The optional *wrapper-proc*, 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. This is provided as 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.

(_fun [*args ::*] *input-type* ⋯ -> *output-type* [-> *output-expr*])

Creates a new function type. This is a convenient syntax for the _cprocedure type constructor, that can handle

complicated cases of argument handling. In its simplest form, only the *input-type*s and the *output-type* are specified and each one 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. First, the full form of each of the types can include an optional label and an expression:

```
type-spec is one of
  type
  (label : type)
  (type = expr)
  (label : type = expr)
```

If an expression 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. 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 will get the string first.

2.8.1 Custom Function Types

The behavior of the _fun type can be customized via custom function types. These 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 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 list that looks like: '`(key: val ...)`', where all of the '`key:`'s are 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.

All of the special custom types that are described here are defined this way.

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 — it means that the type has specific interaction with a function call.

`(define-fun-syntax` *`identifier transformer`*`)` SYNTAX

The results of expanding custom type macros is taken apart by the `_fun` macro, which will lead to code certificate problems. To solve this, do use `define-fun-syntax` instead of `define-syntax`. It is used in the same way, but will avoid such problems.

`_?` CUSTOM C TYPE

This is not a conventional C type, it is a marker for expressions that should not be sent to the ffi 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`*`)` CUSTOM C TYPE

This is for C pointers, where *`mode`* indicates input or output pointers (or both). *`mode`* can be one of the following:

- '`i`', indicating an *input* pointer argument: the wrapper will arrange 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`', indicating 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 will get the Scheme value, allocate and set a pointer using this value, and reference the value after the call. The '`_ptr`' 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 — 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))
```

will create 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` *`type`*`)` CUSTOM C TYPE

This is 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* [*len*]`)` CUSTOM C TYPE

Similar to `_ptr`, except that it is used for converting lists to/from C vectors. The optional *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 any case it can refer to a previous binding for the length of the list which the C function will most likely require.

`_vector` CUSTOM C TYPE

Same as `_list`, except that it uses Scheme vectors instead of lists.

`_bytes` CUSTOM C TYPE

`_bytes` can be used by itself as a simple type that uses a byte string as a C pointer. Alternatively, it can be used as a '`(_bytes o` *len*`)`' 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's there for consistency with the above macros).

`(_cvector` *Custom C Type*`)` SYNTAX

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 — this is described in section 2.4 above. The syntax specified here is an alternative that makes it behave similarly to the `_list` and `_vector` custom types, except that this is more efficient since no Scheme list or vector are needed. (It can be used with all three modes.)

2.9 Miscellaneous Support

`(regexp-replaces` *name substs*`)` PROCEDURE

This is 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 first *name* argument can be any value that will be used to name the foreign object — a symbol, a string, a byte string etc. This 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 "`$`" where `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` *list ctype*`)` PROCEDURE

This function allocates a memory block of an appropriate size, and initializes it using values from *list* and the given *ctype*. The *list* must hold values that can all be converted to C values according to the given *ctype*.

`(cblock->list` *`cblock ctype length`*`)` PROCEDURE

This function converts C pointers *`cblock`* to vectors of *`ctype`*, to Scheme lists. The arguments are the same as in the `list->cblock`. *`length`* must be specified because there is no way to know where the block ends.

`(vector->cblock` *`vector ctype`*`)` PROCEDURE

Same as the `list->cblock` function, only for Scheme vectors.

`(cblock->vector` *`cblock ctype length`*`)` PROCEDURE

Same as the `cblock->vector` function, only for Scheme vectors.

2.10 Functions in the C part

These functions are provided by the internal `#%foreign`, but not re-provided from `foreign`. They are listed with brief explanations. If you find any of these useful, please let us know.

`(ffi-obj` *`objname ffi-lib-or-libname`*`)` PROCEDURE

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 library object, `ffi-lib` is used to create a library object.

`(ffi-obj?` *`x`*`)` PROCEDURE
`(ffi-obj-lib` *`ffi-obj`*`)` PROCEDURE
`(ffi-obj-name` *`ffi-obj`*`)` PROCEDURE

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` *`ctype`*`)` PROCEDURE
`(ctype-scheme->c` *`ctype`*`)` PROCEDURE
`(ctype-c->scheme` *`ctype`*`)` PROCEDURE

Accessors for the components of a C type object, made by `make-ctype`. `ctype-basetype` returns `#f` for primitive types (including cstruct types).

`(ffi-call` *`ptr in-types out-type`*`)` PROCEDURE

This is 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.

`(ffi-callback` *`proc in-types out-type`*`)` PROCEDURE

This operation is 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 will invoke *`proc`* using the types to specify how values are marshaled.

`(ffi-callback? x)` PROCEDURE

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

Index