The Racket Foreign Interface

Version 5.2.1

Eli Barzilay

February 2, 2012

```
(require ffi/unsafe)
```

The `ffi/unsafe` library enables the direct use of C-based APIs within Racket programs—without writing any new C code. From the Racket 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*.

Contents

1 Overview

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

Since using the FFI entails many safety concerns that Racket programmers can normally ignore, the library name includes `unsafe`. Importing the library 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. If your library provides an unsafe interface, then it should have `unsafe` in its name, too.

For more information on the motivation and design of the Racket FFI, see [Barzilay04].

1.1 Libraries, C Types, and Objects

To use the FFI, you must have in mind

- a particular library from which you want to access a function or value,
- a particular symbol exported by the file, and
- the C-level type (typically a function type) of the exported symbol.

The library corresponds to a file with a suffix such as `".dll"`, `".so"`, or `".dylib"` (depending on the platform), or it might be a library within a `".framework"` directory on Mac OS X.

Knowing the library's name and/or path is often the trickiest part of using the FFI. Sometimes, when using a library name without a path prefix or file suffix, the library file can be located automatically, especially on Unix. See `ffi-lib` for advice.

The `ffi-lib` function gets a handle to a library. To extract exports of the library, it's simplest to use `define-ffi-definer` from the `ffi/unsafe/define` library:

```
#lang racket/base
(require ffi/unsafe
         ffi/unsafe/define)

(define-ffi-definer define-curses "libcurses")
```

This `define-ffi-definer` declaration introduces a `define-curses` form for binding a Racket name to a value extracted from `"libcurses"`—which might be located at `"/usr/lib/libcurses.so"`, depending on the platform.

To use `define-curses`, we need the names and C types of functions from `"libcurses"`. We'll start by using the following functions:

```
WINDOW* initscr(void);
int waddstr(WINDOW *win, char *str);
int wrefresh(WINDOW *win);
int endwin(void);
```

We make these functions callable from Racket as follows:

```
(define _WINDOW-pointer (_cpointer 'WINDOW))

(define-curses initscr (_fun -> _WINDOW-pointer))
(define-curses waddstr (_fun _WINDOW-pointer _string -> _int))
(define-curses wrefresh (_fun _WINDOW-pointer -> _int))
(define-curses endwin (_fun -> _int))
```

By convention, an underscore prefix indicates a representation of a C type (such as `_int`) or a constructor of such representations (such as `_cpointer`).

The definition of `_WINDOW-pointer` creates a Racket value that reflects a C type via `_cpointer`, which creates a type representation for a pointer type—usually one that is opaque. The `'WINDOW` argument could have been any value, but by convention, we use a symbol matching the C base type.

Each `define-curses` form uses the given identifier as both the name of the library export and the Racket identifier to bind. The `(_fun ... -> ...)` part of each definition describes the C type of the exported function, since the library file does not encode that information for its exports. The types listed to the left of `->` are the argument types, while the type to the right of `->` is the result type. The pre-defined `_int` type naturally corresponds to the `int` C type, while `_string` corresponds to the `char*` type when it is intended as a string to read.

An optional `#:c-id` clause for `define-curses` can specify a name for the library export that is different from the Racket identifier to bind.

At this point, `initscr`, `waddstr`, `wrefresh`, and `endwin` are normal Racket bindings to Racket functions (that happen to call C functions), and so they can be exported from the defining module or called directly:

```
(define win (initscr))
(void (waddstr win "Hello"))
(void (wrefresh win))
(sleep 1)
(void (endwin))
```

1.2 Function-Type Bells and Whistles

Our initial use of functions like `waddstr` is sloppy, because we ignore return codes. C functions often return error codes, and checking them is a pain. A better approach is to build the check into the `waddstr` binding and raise an exception when the code is non-zero.

The `_fun` function-type constructor includes many options to help convert C functions to nicer Racket functions. We can use some of those features to convert return codes into either `#<void>` or an exception:

```
(define (check v who)
  (unless (zero? v)
    (error who "failed: ~a" v)))

(define-curses initscr (_fun -> _WINDOW-pointer))
(define-curses waddstr (_fun _WINDOW-pointer _string -> (r : _int)
                             -> (check r 'waddstr)))
(define-curses wrefresh (_fun _WINDOW-pointer -> (r : _int)
                              -> (check r 'wrefresh)))
(define-curses endwin (_fun -> (r : _int)
                            -> (check r 'endwin)))
```

Using `(r : _int)` as a result type gives the local name `r` to the C function's result. This name is then used in the result post-processing expression that is specified after a second `->` in the `_fun` form.

1.3 By-Reference Arguments

To get mouse events from `"libcurses"`, we must explicitly enable them through the `mousemask` function:

```
typedef unsigned long mmask_t;
#define BUTTON1_CLICKED 004L

mmask_t mousemask(mmask_t newmask, mmask_t *oldmask);
```

Setting `BUTTON1_CLICKED` in the mask enables button-click events. At the same time, `mousemask` returns the current mask by installing it into the pointer provided as its second argument.

Since these kinds of call-by-reference interfaces are common in C, `_fun` cooperates with a `_ptr` form to automatically allocate space for a by-reference argument and extract the value put there by the C function. Give the extracted value name to use in the post-processing expression. The post-processing expression can combine the by-reference result with the

function's direct result (which, in this case, reports a subset of the given mask that is actually supported).

```
(define _mmask_t _ulong)
(define-curses mousemask (_fun _mmask_t (o : (_ptr o _mmask_t))
                               -> (r : _mmask_t)
                               -> (values o r)))
(define BUTTON1_CLICKED 4)

(define-values (old supported) (mousemask BUTTON1_CLICKED))
```

1.4 C Structs

Assuming that mouse events are supported, the `"libcurses"` library reports them via `getmouse`, which accepts a pointer to a `MEVENT` struct to fill with mouse-event information:

```
typedef struct {
   short id;
   int x, y, z;
   mmask_t bstate;
} MEVENT;

int getmouse(MEVENT *event);
```

To work with `MEVENT` values, we use `define-cstruct`:

```
(define-cstruct _MEVENT ([id _short]
                         [x _int]
                         [y _int]
                         [z _int]
                         [bstate _mmask_t]))
```

This definition binds many names in the same way that `define-struct` binds many names: `_MEVENT` is a C type representing the struct type, `_MEVENT-pointer` is a C type representing a pointer to a `_MEVENT`, `make-MEVENT` constructs a `_MEVENT` value, `MEVENT-x` extracts the `x` fields from an `_MEVENT` value, and so on.

With this C struct declaration, we can define the function type for `getmouse`. The simplest approach is to define `getmouse` to accept an `_MEVENT-pointer`, and then explicitly allocate the `_MEVENT` value before calling `getmouse`:

```
(define-curses getmouse (_fun _MEVENT-pointer -> _int))

(define m (make-MEVENT 0 0 0 0 0))
(when (zero? (getmouse m))
```

```
  ; use m...
  ....)
```

For a more Racket-like function, use `(_ptr o _MEVENT)` and a post-processing expression:

```
(define-curses getmouse (_fun (m : (_ptr o _MEVENT))
                              -> (r : _int)
                              -> (and (zero? r) m)))

(waddstr win (format "click me fast..."))
(wrefresh win)
(sleep 1)

(define m (getmouse))
(when m
  (waddstr win (format "at ~a,~a"
                       (MEVENT-x m)
                       (MEVENT-y m)))
  (wrefresh win)
  (sleep 1))

(endwin)
```

The difference between `_MEVENT-pointer` and `_MEVENT` is crucial. Using `(_ptr o _MEVENT-pointer)` would allocate only enough space for a pointer to an `MEVENT` struct, which is not enough space for an `MEVENT` struct.

1.5 Pointers and Manual Allocation

To get text from the user instead of a mouse click, `"libcurses"` provides `wgetnstr`:

```
int wgetnstr(WINDOW *win, char *str, int n);
```

While the `char*` argument to `waddstr` is treated as a nul-terminated string, the `char*` argument to `wgetnstr` is treated as a buffer whose size is indicated by the final `int` argument. The C type `_string` does not work for such buffers.

One way to approach this function from Racket is to describe the arguments in their rawest form, using plain `_pointer` for the second argument to `wgetnstr`:

```
(define-curses wgetnstr (_fun _WINDOW-pointer _pointer _int -> _int))
```

To call this raw version of `wgetnstr`, allocate memory, zero it, and pass the size minus one (to leave room a nul terminator) to `wgetnstr`:

```
(define SIZE 256)
(define buffer (malloc 'raw SIZE))
(memset buffer 0 SIZE)

(void (wgetnstr win buffer (sub1 SIZE)))
```

When `wgetnstr` returns, it has written bytes to `buffer`. At that point, we can use `cast` to convert the value from a raw pointer to a string:

```
(cast buffer _pointer _string)
```

Conversion via the `_string` type causes the data refereced by the original pointer to be copied (and UTF-8 decoded), so the memory referenced by `buffer` is no longer needed. Memory allocated with `(malloc 'raw ...)` must be released with `free`:

```
(free buffer)
```

1.6 Pointers and GC-Managed Allocation

Instead of allocating `buffer` with `(malloc 'raw ...)`, we could have allocated it with `(malloc 'atomic ...)`:

```
(define buffer (malloc 'atomic SIZE))
```

Memory allocated with `'atomic` is managed by the garbage collector, so `free` is neither necessary nor allowed when the memory referenced by `buffer` is no longer needed. Instead, when `buffer` becomes inaccessible, the allocated memory will be reclaimed automatically.

Allowing the garbage collector (GC) to manage memory is usually preferable. It's easy to forget to call `free`, and exceptions or thread termination can easily skip a `free`.

At the same time, using GC-managed memory adds a different burden on the programmer: data managed by the GC may be moved to a new address as the GC compacts allocated objects to avoid fragmentation. C functions, meanwhile, expect to receive pointers to objects that will stay put.

Fortunately, unless a C function calls back into the Racket run-time system (perhaps through a function that is provided as an argument), no garbage collection will happen between the time that a C function is called and the time that the function returns.

Let's look a few possibilities related to allocation and pointers:

- Ok:

```
(define p (malloc 'atomic SIZE))
(wgetnstr win p (sub1 SIZE))
```

Although the data allocated by `malloc` can move around, `p` will always point to it, and no garbage collection will happen between the time that the address is extracted form `p` to pass to `wgetnstr` and the time that `wgetnstr` returns.

- Bad:

```
(define p (malloc 'atomic SIZE))
(define i (cast p _pointer _intptr))
(wgetnstr win (cast i _intptr _pointer) (sub1 SIZE))
```

The data referenced by `p` can move after the address is converted to an integer, in which case `i` cast back to a pointer will be the wrong address.

Obviously, casting a pointer to an integer is generally a bad idea, but the cast simulates another possibility, which is passing the pointer to a C function that retains the pointer in its own private store for later use. Such private storage is invisible to the Racket GC, so it has the same effect as casting the pointer to an integer.

- Ok:

```
(define p (malloc 'atomic SIZE))
(define p2 (ptr-add p 4))
(wgetnstr win p2 (- SIZE 5))
```

The pointer `p2` retains the original reference and only adds the `4` at the last minute before calling `wgetnstr` (i.e., after the point that garbage collection is allowed).

- Ok:

```
(define p (malloc 'atomic-interior SIZE))
(define i (cast p _pointer _intptr))
(wgetnstr win (cast i _intptr _pointer) (sub1 SIZE))
```

This is ok assuming that `p` itself stays accessible, so that the data it references isn't reclaimed. Allocating with `'atomic-interior` puts data at a particular address and keeps it there. A garbage collection will not change the address in `p`, and so `i` (cast back to a pointer) will always refer to the data.

Keep in mind that C struct constructors like `make-MEVENT` are effectively the same as `(malloc 'atomic ...)`; the result values can move in memory during a garbage collection. The same is true of byte strings allocated with `make-bytes`, which (as a convenience) can be used directly as a pointer value (unlike character strings, which are always copied for UTF-8 encoding or decoding).

For more information about memory management and garbage collection, see §3 "Memory Allocation" in *Inside: Racket C API*.

1.7 Reliable Release of Resources

Using GC-managed memory saves you from manual `free`s for plain memory blocks, but C libraries often allocate resources and require a matching call to a function that releases the resources. For example, `"libcurses"` supports windows on the screen that are created with `newwin` and released with `delwin`:

```
WINDOW *newwin(int lines, int ncols, int y, int x);
int delwin(WINDOW *win);
```

In a sufficiently complex program, ensuring that every `newwin` is paired with `delwin` can be challenging, especially if the functions are wrapped by otherwise safe functions that are provided from a library. A library that is intended to be safe for use in a sandbox, say, must protect against resource leaks within the Racket process as a whole when a sandboxed program misbehaves or is terminated.

The `ffi/unsafe/alloc` library provides functions to connect resource-allocating functions and resource-releasing functions. The library then arranges for finalization to release a resource if it becomes inaccessible (according to the GC) before it is explicitly released. At the same time, the library handles tricky atomicity requirements to ensure that the finalization is properly registered and never run multiple times.

Using `ffi/unsafe/alloc`, the `newwin` and `delwin` functions can be imported with `allocator` and `deallocator` wrappers, respectively:

```
(require ffi/unsafe/alloc)

(define-curses delwin (_fun _WINDOW-pointer -> _int)
  #:wrap (deallocator))

(define-curses newwin (_fun _int _int _int _int
                            -> _WINDOW-pointer)
  #:wrap (allocator delwin))
```

A `deallocator` wrapper makes a function cancel any existing finalizer for the function's argument. An `allocator` wrapper refers to the deallocator, so that the deallocator can be run if necessary by a finalizer.

1.8 More Examples

For more examples of common FFI patterns, see the defined interfaces in the `"ffi/examples"` collection. See also [Barzilay04].

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 a *foreign-library value*, `#f` otherwise.

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

Returns a foreign-library value or the result of *`fail`*. Normally,

- *`path`* is a path without a version or suffix (i.e., without `".dll"`, `".so"`, or `".dylib"`); and
- *`version`* is a list of versions to try in order with `#f` (i.e., no version) as the last element of the list; for example, `'("2" #f)` indicates version 2 with a fallback to a versionless library.

A string or `#f` *`version`* is equivalent to a list containing just the string or `#f`, and an empty string (by itself or in a list) is equivalent to `#f`.

Beware of relying on versionless library names. On some platforms, versionless library names are provided only by development packages. At the same time, other platforms may require a versionless fallback. A list of version strings followed by `#f` is typically best for *`version`*.

Assuming that *`path`* is not `#f`, the result from `ffi-lib` represents the library found by the following search process:

- If *`path`* is not an absolute path, look in each directory reported by *`get-lib-dirs`*. In each directory, try *`path`* with the first version in *`version`*, adding a suitable suffix if *`path`* does not already end in the suffix, then try the second version in *`version`*, etc. (If *`version`* is an empty list, no paths are tried in this step.)

- Try the same filenames again, but without converting the path to an absolute path, which allows the operating system to use its own search paths. (If *version* is an empty list, no paths are tried in this step.)
- Try *path* without adding any version or suffix, and without converting to an absolute path.
- Try the version-adjusted filenames again, but relative to the current directory. (If *version* is an empty list, no paths are tried in this step.)
- Try *path* without adding any version or suffix, but converted to an absolute path relative to the current directory.

If none of the paths succeed and *fail* is a function, then *fail* is called in tail position. If *fail* is `#f`, an error is reported from trying the first path from the second bullet above or (if *version* is an empty list) from the third bullet above. A library file may exist but fail to load for some reason; the eventual error message will unfortunately name the fallback from the second or third bullet, since some operating systems offer no way to determine why a given library path failed.

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: Racket C API*). The *version* argument is ignored when *path* is `#f`.

Due to the way the operating system performs dynamic binding, loaded libraries are associated with Racket (or DrRacket) for the duration of the process. Re-evaluating `ffi-lib` (or hitting the Run button in DrRacket) will not force a re-load of the corresponding library.

```
(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 it is converted to one by calling `ffi-lib`. If *objname* is found in *lib*, it is converted to Racket 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 Racket 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 Racket 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 Racket binding, but *id* is actually redirected through a parameter-like procedure created by `make-c-parameter`. The *id* is used both for the Racket 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 Racket 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. Finally, the result can be a vector of size 2 containing an element representation followed by an exact-integer count.

```
(compiler-sizeof sym) → exact-nonnegative-integer?
  sym : (or/c symbol? (listof symbol?))
```

Possible values for *sym* are `'int`, `'char`, `'short`, `'long`, `'*`, `'void`, `'float`, `'double`, or lists of symbols, such as `'(long long)`. The result is the size of the corresponding 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?
_llong : ctype?
_sllong : ctype?
_ullong : ctype?
_intptr : ctype?
_sintptr : ctype?
_uintptr : 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. Similarly, the `_intptr` 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 `_intptr` and `_uintptr` but assume that the quantities fit in Racket's immediate integers (i.e., not bignums).

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

Similar to `_fixnum`/`_ufixnum`, but based on `_int`/`_uint` instead of `_intptr`/`_uintptr`, and 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. Both single- and double-precision Racket numbers are accepted for conversion via both `_float` and `_double`, while both `_float` and `_double` coerce C values to double-precision Racket numbers. The type `_double*` coerces any Racket 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 Racket `#<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 Racket's representation without encodings.

The form `_bytes` form can be used type for Racket 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 Racket's native Unicode strings, which are in UCS-4 format. These correspond to the C `mzchar*` type used by Racket. 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 Racket's paths. As usual, the types treat `#f` as `NULL` and vice versa.

Beware that changing the current directory via `current-directory` does not change the OS-level current directory as seen by foreign library functions. Paths normally should be converted to absolute form using `path->complete-path` (which uses the `current-directory` parameter) before passing them to a foreign function.

```
_symbol : ctype?
```

Simple `char*` strings as Racket 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 Racket 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: Racket byte strings are allowed and passed as is, and Racket 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 Racket 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 Racket 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 Racket 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 Racket eof value.

```
_string/eof : ctype?
```

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

3.5 Pointer Types

```
_pointer : ctype?
```

Corresponds to Racket "C pointer" objects. These pointers can have an arbitrary Racket object attached as a type tag. The tag is ignored by built-in functionality; it is intended to be used by interfaces. See §5.3 "Tagged C Pointer Types" for creating pointer types that use these tags for safety. A `#f` value is converted to `NULL` and vice versa.

The address referenced by a `_pointer` value must not refer to memory managed by the garbage collector (unless the address corresponds to a value that supports interior pointers and that is otherwise referenced to preserve the value from garbage collection). The reference is not traced or updated by the garbage collector.

The `equal?` predicate equates C pointers (including pointers for `_gcpointer` and possibly containing an offset) when they refer to the same address.

```
_gcpointer : ctype?
```

Like `_pointer`, but for a value that can refer to memory managed by the garbage collector.

Although a `_gcpointer` can reference to memory that is not managed by the garbage collector, beware of using an address that might eventually become managed by the garbage collector. For example, if a reference is created by `malloc` with `'raw` and released by `free`, then the `free` may allow the memory formerly occupied by the reference to be used later by the garbage collector.

```
_racket : ctype?
_scheme : ctype?
```

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

```
(_or-null ctype) → ctype?
  ctype : ctype?
```

Creates a type that is like *ctype*, but `#f` is converted to `NULL` and vice versa. The given *ctype* must have the same C representation as `_pointer`, `_gcpointer`, or `_fpointer`.

```
(_gcable ctype) → ctype?
  ctype : ctype?
```

Creates a type that is like *ctype*, but whose base representation is like `_gcpointer` instead of `_pointer`. The given *ctype* must have a base representation like `_pointer` or `_gcpointer` (and in the later case, the result is the *ctype*).

3.6 Function Types

```
(_cprocedure  input-types
              output-type
             [#:abi abi
              #:atomic? atomic?
              #:async-apply async-apply
              #:in-original-place? in-original-place?
              #:save-errno save-errno
              #:wrapper wrapper
              #:keep keep])                          → any
  input-types : (list ctype?)
  output-type : ctype?
  abi : (or/c #f 'default 'stdcall 'sysv) = #f
  atomic? : any/c = #f
  async-apply : (or/c #f ((-> any) . -> . any)) = #f
  in-original-place? : any/c = #f
  save-errno : (or/c #f 'posix 'windows) = #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 Racket 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 Racket procedures to foreign functions, which will generate a foreign function pointer that calls the given Racket procedure when it is used. There are no restrictions on the Racket procedure; in particular, its lexical context is properly preserved.

The optional *abi* keyword argument determines the foreign ABI that is used. Supplying `#f` or `'default` indicates the platform-dependent default. The other possible values—`'stdcall` and `'sysv` (i.e., "cdecl")—are currently supported only for 32-bit Windows; using them on other platforms raises an exception. See also `ffi/winapi`.

If *atomic?* is true, then when a Racket procedure is given this procedure type and called from foreign code, then the Racket process is put into atomic mode while evaluating the Racket procedure body. In atomic mode, other Racket threads do not run, so the Racket code must not call any function that potentially blocks on synchronization with other threads, or else it may lead to deadlock. In addition, the Racket 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.

If an *async-apply* procedure is provided, then a Racket procedure with the generated procedure type can be applied in a foreign thread (i.e., an OS-level thread other than the one used to run Racket). The call in the foreign thread is transferred to the OS-level thread that runs Racket, but the Racket-level thread (in the sense of `thread`) is unspecified; the job of the provided *async-apply* procedure is to arrange for the callback procedure to be run in a suitable Racket thread. The given *async-apply* procedure is applied to a thunk that encapsulates the specific callback invocation, and the foreign OS-level thread blocks until the thunk is called and completes; the thunk must be called exactly once, and the callback invocation must return normally. The given *async-apply* procedure itself is called in atomic mode (see *atomic?* above). If the callback is known to complete quickly, requires no synchronization, and works independent of the Racket thread in which it runs, then it is safe for the given *async-apply* procedure to apply the thunk directly. Otherwise, the given *async-apply* procedure must arrange for the thunk to be applied in a suitable Racket thread sometime after the given *async-apply* procedure itself returns; if the thunk raises an exception or synchronizes within an unsuitable Racket-level thread, it can deadlock or otherwise damage the Racket process. Foreign-thread detection to trigger *async-apply* works only when Racket is compiled with OS-level thread support, which is the default for many platforms. If a callback with an *async-apply* is called from foreign code in the same OS-level thread that runs Racket, then the *async-apply* wrapper is not used.

The *atomic?* and *async-apply* arguments affect callbacks into Racket, while *in-original-place?* affects calls from Racket into foreign code.

If *in-original-place?* is true, then when a foreign procedure with the generated type is called in any Racket place, the procedure is called from the original Racket place. Use this mode for a foreign function that is not thread-safe at the C level, which means that it is not place-safe at the Racket level. Callbacks from place-unsafe code back into Racket at a non-original place typically will not work, since the place of the Racket code may have a different allocator than the original place.

If *save-errno* is `'posix`, then the value of `errno` is saved (specific to the current thread) immediately after a foreign function returns. The saved value is accessible through `saved-errno`. If *save-errno* is `'windows`, then the value of `GetLastError()` is saved for later use via `saved-errno`; the `'windows` option is available only on Windows (on other platforms `saved-errno` will return 0). If *save-errno* is `#f`, no error value is saved automat-

ically. The error-recording support provided by *save-errno* is needed because the Racket runtime system may otherwise preempt the current Racket thread and itself call functions that set error values.

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 Racket procedure, and possibly changes the result values too.

Sending Racket 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 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 Racket 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 Racket 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
              | #:save-errno save-errno-expr
              | #:keep keep-expr
              | #:atomic? atomic?-expr
              | #:async-apply async-apply-expr
              | #:in-original-place? in-original-place?-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.

For instance,

```
(_fun _int _string -> _int)
```

specifies a function that receives an integer and a string, and returns an integer.

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.
- `keywords:` specifies keyword/value expressions that will be used with the surrounding `_fun` form. (Note: the keyword/value sequence follows `keywords:`, not parenthesized.)

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 as well as a syntax transformer (i.e, macro). The type is expanded by applying the procedure produced by *transformer-expr* to a use of the custom function type.

For instance, the following defines a new type that automatically coerces the input number to an inexact form which is compatible with the `_float` type.

```
(define-fun-syntax _float*
  (syntax-id-rules (_float*)
    [(_float*) (type: _float pre: (x => (+ 0.0 x)))]))

(_fun _float* -> _bool)
```

```
_?
```

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 Racket 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 Racket 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).

3.7 C Struct Types

```
(make-cstruct-type types [abi alignment]) → ctype?
  types : (listof ctype?)
  abi : (or/c #f 'default 'stdcall 'sysv) = #f
  alignment : (or/c #f 1 2 4 8 16) = #f
```

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 Racket 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.

If *alignment* is #f, then the natural alignment of each type in *types* is used for its alignment within the struct type. Otherwise, *alignment* is used for all struct type members.

```
(_list-struct [#:alignment alignment]
              type ...+)              → ctype?
  alignment : (or/c #f 1 2 4 8 16) = #f
  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] ...) alignment)

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

alignment =
          | #:alignment alignment-expr
```

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 Racket 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 object that is used with instances. The tag object may be the symbol form of *id* or a list of symbols containing the *id* symbol and other symbols, such as the *super-id* symbol.
- make-*id* : a constructor, which expects an argument for each type.
- *id*-*field-id* : an accessor function for each *field-id*; if the field has a cstruct type, then the result of the accessor is a pointer to the field within the enclosing structure, rather than a copy of the field.

- `set-`*`id`*`-field-`*`id`*`!` : a mutator function for each *`field-id`*.
- *`id`*: structure-type information compatible with `struct-out` or `match` (but not `define-struct`); currently, this information is correct only when no *`super-id`* is specified.
- *`id`*`->list` : a function that converts a struct into a list of values.

Objects of the new type are actually C pointers, with a type tag that is the symbol form of *`id`* or a list that contains the symbol 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 Racket; 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. Thus, struct fields can hold only non-pointer values, pointers to memory outside the GC's control, and otherwise-reachable pointers to immobile GC-managed values (such as those allocated with `malloc` and `'internal` or `'internal-atomic`).

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-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))
; 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 Racket 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 and the new ones:

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

3.8 C Array Types

```
(make-array-type type count) → ctype?
  type : ctype?
  count : exact-nonnegative-integer?
```

The primitive type constructor for creating new C array types. Like C struct types, array types are new primitive types with no conversion functions associated. When used as a function argument or return type, array types behave like pointer types; otherwise, array types behave like struct types (i.e., a struct with as many fields as the array has elements), particularly when used for a field within a struct type.

Since an array is treated like a struct, `cast`ing a pointer type to an array type does not work. Instead, use `ptr-ref` with a pointer, an array type constructed with `_array`, and index `0` to convert a pointer to a Racket representation that works with `array-ref` and `array-set!`.

```
(_array type count ...+) → ctype?
  type : ctype?
  count : exact-nonnegative-integer?
```

Creates an array type whose Racket representation is an array that works with `array-ref` and `array-set!`. The array is not copied; the Racket representation is backed by the underlying C representation.

Supply multiple *count*s for a multidimensional array. Since C uses row-major order for arrays, `(_array t n m)` is equivalent to `(_array (_array t m) n)`, which is different from an array of pointers to arrays.

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

Returns `#t` if *v* is a Racket representation of a C value via `_array`, `#f` otherwise.

```
(array-ref a i ...+) → any/c
  a : array?
  i : exact-nonnegative-integer?
```

Extracts an element from an array. Use multiple *i* indices for a multidimensional array access; using fewer indices than the array dimension produces a sub-array.

```
(array-set! a i ...+ v) → void?
  a : array?
  i : exact-nonnegative-integer?
  v : any/c
```

Sets an element in an array. Use multiple *i* indices for a multidimensional array update; using fewer indices than the array dimension sets a sub-array (i.e., *v* must be an array of the same size as the sub-array and *v* is copied into the sub-array).

```
(array-ptr a) → cpointer?
  a : array?
```

Extracts the pointer for an array's storage.

```
(_array/list type count ...+) → ctype?
  type : ctype?
  count : exact-nonnegative-integer?
```

Like `_array`, but the Racket representation is a list (or list of lists for a multidimensional array) of elements copied to and from an underlying C array.

```
(_array/vector type count ...+) → ctype?
  type : ctype?
  count : exact-nonnegative-integer?
```

Like _array, but the Racket representation is a vector (or vector of vectors for a multidimensional array) of elements copied to and from an underlying C array.

3.9 C Union Types

```
(make-union-type type ...+) → ctype?
  type : ctype?
```

The primitive type constructor for creating new C union types. Like C struct types, union types are new primitive types with no conversion functions associated. Unions are always treated like structs.

```
(_union type ...+) → ctype?
  type : ctype?
```

Creates a union type whose Racket representation is a union that works with union-ref and union-set!. The union is not copied; the Racket representation is backed by the underlying C representation.

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

Returns #t if v is a Racket representation of a C value via _union, #f otherwise.

```
(union-ref u i) → any/c
  u : union?
  i : exact-nonnegative-integer?
```

Extracts a variant from a union.

```
(union-set! u i v) → void?
  u : union?
  i : exact-nonnegative-integer?
  v : any/c
```

Sets a variant in a union..

```
(union-ptr u) → cpointer?
  u : array?
```

Extracts the pointer for a union's storage.

3.10 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 #:unknown unknown]) → ctype?
  symbols : list?
  basetype : ctype? = _ufixint
  unknown : any/c = (lambda (x) (error ....))
```

Takes a list of symbols and generates an enumeration type. The enumeration maps between a symbol in the given *symbols* list and corresponding 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.

The *unknown* argument specifies the result of converting an unknown integer from the foreign side: it can be a one-argument function to be applied on the integer, or a value to return instead. The default is to throw an exception.

```
(_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-ior`. 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 Racket pointer objects can contain the same pointer.

If the values are both C pointers—as opposed to `#f`, a byte string, `ffi-obj`, or callback—this comparison is the same as `equal?`.

```
(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 Pointer Dereferencing

```
(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). When a *type* argument is present, the result is that of a call to memset with no *type* argument and the *count* multiplied by the size associated with the *type*.

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

Returns the Racket 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 Memory Management

For general information on C-level memory management with Racket, see *Inside: Racket 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
        (one-of/c 'nonatomic 'stubborn 'uncollectable
  mode :          'eternal 'interior 'atomic-interior = absent
                  '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 Racket 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 Racket'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 a `_gcpointer`- or `_scheme`-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 Racket 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) Racket 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 Racket (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 Racket object—even ones that have nothing to do with foreign code. Note, however, that the finalizer is registered for the *Racket* 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 Racket 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 Racket 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 Racket 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 Racket byte string, because the offset pointer would be to the middle of a collectable object (which is not allowed).

5 Derived Utilities

5.1 Safe Homogenous Vectors

```
(require ffi/vector)
```

Homogenous vectors are similar to C vectors (see §5.2 “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; for example, `make-u8vector` is an alias for `make-bytes`.

```
(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?
(u8vector->cpointer vec) → cpointer?
  vec : u8vector?
```

Like `_cvector`, but for vectors of `_byte` elements. These are aliases for `byte` operations, where `u8vector->cpointer` is the identity function.

```
(_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?
(s8vector->cpointer vec) → cpointer?
  vec : s8vector?
```

Like `make-vector`, etc., but for `_int8` elements. The `s8vector->cpointer` function extracts a plain pointer to the underlying array.

```
(_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?
(s16vector->cpointer vec) → cpointer?
  vec : s16vector?
```

Like `make-vector`, etc., but for `_int16` elements. The `s16vector->cpointer` function extracts a plain pointer to the underlying array.

```
(_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?
(u16vector->cpointer vec) → cpointer?
  vec : u16vector?
```

Like `make-vector`, etc., but for `_uint16` elements. The `u16vector->cpointer` function extracts a plain pointer to the underlying array.

```
(_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?
(s32vector->cpointer vec) → cpointer?
  vec : s32vector?
```

Like `make-vector`, etc., but for `_int32` elements. The `s32vector->cpointer` function extracts a plain pointer to the underlying array.

```
(_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?
(u32vector->cpointer vec) → cpointer?
  vec : u32vector?
```

Like `make-vector`, etc., but for `_uint32` elements. The `u32vector->cpointer` function extracts a plain pointer to the underlying array.

```
(_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?
(s64vector->cpointer vec) → cpointer?
  vec : s64vector?
```

Like `make-vector`, etc., but for `_int64` elements. The `s64vector->cpointer` function extracts a plain pointer to the underlying array.

```
(_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?
(u64vector->cpointer vec) → cpointer?
  vec : u64vector?
```

Like make-vector, etc., but for _uint64 elements. The u64vector->cpointer function extracts a plain pointer to the underlying array.

```
(_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?
(f32vector->cpointer vec) → cpointer?
  vec : f32vector?
```

Like `make-vector`, etc., but for `_float` elements. The `f32vector->cpointer` function extracts a plain pointer to the underlying array.

```
(_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?
(f64vector->cpointer vec) → cpointer?
  vec : f64vector?
```

Like `make-vector`, etc., but for `_double*` elements. The `f64vector->cpointer` function extracts a plain pointer to the underlying array.

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

Like _cvector, but for vectors of _double* elements.

5.2 Safe C Vectors

```
(require ffi/cvector)
(require ffi/unsafe/cvector)
```

The ffi/unsafe/cvector library exports the bindings of this section. The ffi/cvector library exports the same bindings, except for the unsafe make-cvector* operation.

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

```
(_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 Racket side. The longer form behaves similarly to the _list and _vector custom types, except that _cvector is more efficient; no Racket list or vector is needed.

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

Allocates a C vector using the given type and length. The resulting vector is not guaranteed to contain any particular values.

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

Creates a C vector of the given type, initialized to the given list of vals.

```
(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*.

```
(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.

5.3 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 Racket, 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.

Although any value can be used as a tag, by convention the symbol form of a type name—without a leading underscore—is used as the tag. For example, a pointer type _animal would normally use 'animal as the key.

Pointer tags are checked with cpointer-has-tag? and changed with cpointer-push-tag!, which means that other tags are preserved on an existing pointer value. 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 is displayed.

The _cpointer/null function 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 Racket, so they are not tagged.

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

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

The optional expressions produce 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 symbol form of id.

```
(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 Racket 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).

5.4 Defining Bindings

```
(require ffi/unsafe/define)
```

```
(define-ffi-definer define-id ffi-lib-expr
  option ...)

option = #:provide provide-id
       | #:define core-define-id
       | #:default-make-fail default-make-fail-expr
```

Binds define-id as a definition form to extract bindings from the library produced by ffi-lib-expr. The syntax of define-id is

```
(define-id id type-expr
  bind-option ...)

bind-option = #:c-id c-id
            | #:wrap wrap-expr
            | #:make-fail make-fail-expr
            | #:fail fail-expr
```

A *define-id* form binds id by extracting a binding with the name c-id from the library produced by *ffi-lib-expr*, where c-id defaults to id. The other options support further wrapping and configuration:

- Before the extracted result is bound as *id*, it is passed to the result of *wrap-expr*, which defaults to values. Expressions such as (allocator *delete*) or (deallocator) are useful as *wrap-expr*s.
- The #:make-fail and #:fail options are mutually exclusive; if *make-fail-expr* is provided, it is applied to '*id* to obtain the last argument to get-ffi-obj; if *fail-expr* is provided, it is supplied directly as the last argument to get-ffi-obj. The make-not-available function is useful as *make-fail-expr* to cause a use of *id* to report an error when it is applied if *c-id* was not found in the foreign library.

If *provide-id* is provided to define-ffi-definer, then *define-id* also provides its binding using *provide-id*. The provide-protected form is usually a good choice for *provide-id*.

If *core-define-id* is provided to define-ffi-definer, then code-define-id is used in place of define in the expansion of *define-id* for each binding.

If *default-make-fail-expr* is provided to define-ffi-definer, it serves as the default #:make-fail value for *define-id*.

For example,

```
(define-ffi-definer define-gtk gtk-lib)
```

binds define-gtk to extract FFI bindings from gtk-lib, so that gtk_rc_parse could be bound as

```
(define-gtk gtk_rc_parse (_fun _path -> _void))
```

If gtk_rc_parse is not found, then define-gtk reports an error immediately. If define-gtk is instead defined with

```
(define-ffi-definer define-gtk gtk-lib
   #:default-make-fail make-not-available)
```

then if `gtk_rc_parse` is not found in `gtk-lib`, an error is reported only when `gtk_rc_parse` is called.

```
(make-not-available name) → (#:rest list? -> any/c)
  name : symbol?
```

Returns a procedure that takes any number of arguments and reports an error message from *name*. This function is intended for using with `#:make-fail` or `#:default-make-fail` in `define-ffi-definer`

```
(provide-protected provide-spec ...)
```

Equivalent to `(provide (protect-out provide-spec ...))`. The `provide-protected` identifier is useful with `#:provide` in `define-ffi-definer`.

5.5 Allocation and Finalization

```
(require ffi/unsafe/alloc)
```

The `ffi/unsafe/alloc` library provides utilities for ensuring that values allocated through foreign functions are reliably deallocated.

```
((allocator dealloc) alloc) → procedure?
  dealloc : (any/c . -> . any)
  alloc : procedure?
```

Produces a procedure that behaves like *alloc*, but the result of *alloc* is given a finalizer that calls *dealloc* on the result if it is not otherwise freed through a deallocator (as designated with `deallocator`). In addition, *alloc* is called in atomic mode (see `start-atomic`); its result is received and registered in atomic mode, so that the result is reliably freed.

The *dealloc* procedure itself need not be specifically designated a deallocator (via `deallocator`). If a deallocator is called explicitly, it need not be the same as *dealloc*.

```
((deallocator [get-arg]) dealloc) → procedure?
  get-arg : (list? . -> . any/c) = car
  dealloc : procedure?
((releaser [get-arg]) dealloc) → procedure?
  get-arg : (list? . -> . any/c) = car
  dealloc : procedure?
```

Produces a procedure that behaves like *dealloc*. The *dealloc* procedure is called in atomic mode (see start-atomic), and the reference count on one of its arguments is decremented; if the reference count reaches zero, no finalizer associated by an allocator- or referencer-wrapped procedure is invoked when the value becomes inaccessible.

The optional *get-arg* procedure determines which of *dealloc*'s arguments correspond to the released object; *get-arg* receives a list of arguments passed to *dealloc*, so the default car selects the first one.

The releaser procedure is a synonym for deallocator.

```
((retainer release [get-arg]) retain) → procedure?
  release : (any/c . -> . any)
  get-arg : (list? . -> . any/c) = car
  retain : procedure?
```

Produces a procedure that behaves like *retain*. The procedure is called in atomic mode (see start-atomic), and the reference count on one of its arguments is incremented, with *release* recorded as the corresponding release procedure to be called by the finalizer on the retained object (unless some deallocator, as wrapped by deallocate, is explicitly called first).

The optional *get-arg* procedure determines which of *retain*'s arguments correspond to the retained object; *get-arg* receives a list of arguments passed to *retain*, so the default car selects the first one.

The *release* procedure itself need not be specifically designated a deallocator (via deallocator). If a deallocator is called explicitly, it need not be the same as *release*.

5.6 Atomic Execution

```
(require ffi/unsafe/atomic)
```

```
(start-atomic) → void?
(end-atomic) → void?
```

Disables and enables context switches and delivery of break exceptions at the level of Racket threads. Calls to start-atomic and end-atomic can be nested.

Using call-as-atomic is somewhat safer, in that call-as-atomic correctly catches exceptions and re-raises them after exiting atomic mode. For simple uses where exceptions need not be handled, however, start-atomic and end-atomic are faster.

```
(start-breakable-atomic) → void?
(end-breakable-atomic) → void?
```

Like start-atomic and end-atomic, but the delivery of break exceptions is not suspended. To ensure that a call to start-atomic is reliably paired with a call to end-atomic, use dynamic-wind pre- and post thunks or some other context where breaks are disabled. These variants are not faster than plan start-atomic and end-atomic.

```
(call-as-atomic thunk) → any
  thunk : (-> any)
```

Calls *thunk* in atomic mode. If *thunk* raises and exception, the exception is caught and re-raised after exiting atomic mode.

When call-as-atomic is used in the dynamic extent of call-as-atomic, then *thunk* is simply called directly (as a tail call).

```
(call-as-nonatomic thunk) → any
  thunk : (-> any)
```

Within the dynamic extent of a call-as-atomic, calls *thunk* in non-atomic mode. Beware that the current thread maybe suspended or terminated by other threads during *thunk*, in which case the call never returns.

When used not in the dynamic extent of call-as-atomic, call-as-nonatomic raises exn:fail:contract.

5.7 Speculatively Atomic Execution

```
(require ffi/unsafe/try-atomic)
```

The ffi/unsafe/try-atomic supports atomic execution that can be suspended and resumed in non-atomic mode if it takes too long or if some external event causes the attempt to be abandoned.

```
(call-as-nonatomic-retry-point thunk) → any
  thunk : (-> any)
```

Calls *thunk* in atomic mode (see start-atomic and end-atomic) while allowing *thunk* to use try-atomic. Any incomplete computations started with try-atomic are run non-atomically after *thunk* returns. The result of *thunk* is used as the result of call-as-nonatomic-retry-point.

```
(try-atomic  thunk
             default-val
            [#:should-give-up? give-up-proc
             #:keep-in-order? keep-in-order?]) → any
```

```
thunk : (-> any)
default-val : any/c
give-up-proc : (-> any/c) = run-200-milliseconds
keep-in-order? : any/c = #t
```

Within the dynamic extent of a `call-as-nonatomic-retry-point` call, attempts to run *thunk* in the existing atomic mode. The *give-up-proc* procedure is called periodically to determine whether atomic mode should be abandoned; the default *give-up-proc* returns true after 200 milliseconds. If atomic mode is abandoned, the computation is suspended, and *default-val* is returned, instead. The computation is resumed later by the enclosing `call-as-nonatomic-retry-point` call.

If *keep-in-order?* is true, then if `try-atomic` is called after an earlier computation was suspended for the same `call-as-nonatomic-retry-point` call, then *thunk* is immediately enqueued for completion by `call-as-nonatomic-retry-point` and *default-val* is returned.

The *give-up-proc* callback is polled only at points where the level of atomic-mode nesting (see `start-atomic`, `start-breakable-atomic`, and `call-as-atomic`) is the same as at the point of calling `try-atomic`.

If *thunk* aborts the current continuation using `(default-continuation-prompt-tag)`, the abort is suspended the resumed by the enclosing `call-as-nonatomic-retry-point`. Escapes to the context of the call to *thunk* using any other prompt tag or continuation are blocked (using `dynamic-wind`) and simply return `(void)` from *thunk*.

5.8 Objective-C FFI

```
(require ffi/unsafe/objc)
```

The `ffi/unsafe/objc` library builds on `ffi/unsafe` to support interaction with Objective-C.

The library supports Objective-C interaction in two layers. The upper layer provides syntactic forms for sending messages and deriving subclasses. The lower layer is a think wrapper on the Objective-C runtime library functions. Even the upper layer is unsafe and relatively low-level compared to normal Racket libraries, because argument and result types must be declared in terms of FFI C types (see §3.1 “Type Constructors”).

5.8.1 FFI Types and Constants

```
_id : ctype?
```

The type of an Objective-C object, an opaque pointer.

```
_Class : ctype?
```

The type of an Objective-C class, which is also an `_id`.

```
_Protocol : ctype?
```

The type of an Objective-C protocol, which is also an `_id`.

```
_SEL : ctype?
```

The type of an Objective-C selector, an opaque pointer.

```
_BOOL : ctype?
```

The Objective-C boolean type. Racket values are converted for C in the usual way: `#f` is false and any other value is true. C values are converted to Racket booleans.

```
YES : boolean?
```

Synonym for `#t`

```
NO : boolean?
```

Synonym for `#f`

5.8.2 Syntactic Forms and Procedures

```
(tell result-type obj-expr method-id)
(tell result-type obj-expr arg ...)

result-type =
            | #:type ctype-expr

        arg = method-id arg-expr
            | method-id #:type ctype-expr arg-expr
```

Sends a message to the Objective-C object produced by *obj-expr*. When a type is omitted for either the result or an argument, the type is assumed to be `_id`, otherwise it must be specified as an FFI C type (see §3.1 “Type Constructors”).

If a single *method-id* is provided with no arguments, then *method-id* must not end with `:`; otherwise, each *method-id* must end with `:`.

Examples:

```
> (tell NSString alloc)
#<cpointer:id>
> (tell (tell NSString alloc)
        initWithUTF8String: #:type _string "Hello")
#<cpointer:id>
```

```
(tellv obj-expr method-id)
(tellv obj-expr arg ...)
```

Like `tell`, but with a result type `_void`.

```
(import-class class-id ...)
```

Defines each *class-id* to the class (a value with FFI type `_Class`) that is registered with the string form of *class-id*. The registered class is obtained via `objc_lookUpClass`.

Example:

```
> (import-class NSString)
```

```
(import-protocol protocol-id ...)
```

Defines each *protocol-id* to the protocol (a value with FFI type `_Protocol`) that is registered with the string form of *protocol-id*. The registered class is obtained via `objc_getProtocol`.

Example:

```
> (import-protocol NSCoding)
```

```
(define-objc-class class-id superclass-expr
  maybe-mixins
  maybe-protocols
  [field-id ...]
  method)
```

```
   maybe-mixins =
                | #:mixins (mixin-expr ...)

maybe-protocols =
                | #:protocols (protocol-expr ...)

         method = (mode result-ctype-expr (method-id) body ...+)
                | (mode result-ctype-expr (arg ...+) body ...+)

           mode = +
                | -
                | +a
                | -a

            arg = method-id [ctype-expr arg-id]
```

Defines *class-id* as a new, registered Objective-C class (of FFI type `_Class`). The *superclass-expr* should produce an Objective-C class or `#f` for the superclass. An optional `#:mixins` clause can specify mixins defined with `define-objc-mixin`. An optional `#:protocols` clause can specify Objective-C protocols to be implemented by the class.

Each *field-id* is an instance field that holds a Racket value and that is initialized to `#f` when the object is allocated. The *field-id*s can be referenced and `set!` directly when the method *body*s. Outside the object, they can be referenced and set with `get-ivar` and `set-ivar!`.

Each *method* adds or overrides a method to the class (when *mode* is `-` or `-a`) to be called on instances, or it adds a method to the meta-class (when *mode* is `+` or `+a`) to be called on the class itself. All result and argument types must be declared using FFI C types (see §3.1 "Type Constructors"). When *mode* is `+a` or `-a`, the method is called in atomic mode (see `_cprocedure`).

If a *method* is declared with a single *method-id* and no arguments, then *method-id* must not end with `:`. Otherwise, each *method-id* must end with `:`.

If the special method `dealloc` is declared for mode `-`, it must not call the superclass method, because a `(super-tell dealloc)` is added to the end of the method automatically. In addition, before `(super-tell dealloc)`, space for each *field-id* within the instance is deallocated.

Example:

```
> (define-objc-class MyView NSView
    [bm] ; <- one field
    (- _racket (swapBitwmap: [_racket new-bm])
       (begin0 bm (set! bm new-bm)))
    (- _void (drawRect: [_NSRect exposed-rect])
```

```
      (super-tell drawRect: exposed-rect)
      (draw-bitmap-region bm exposed-rect))
    (- _void (dealloc)
      (when bm (done-with-bm bm))))
```

```
(define-objc-mixin (class-id superclass-id)
  maybe-mixins
  maybe-protocols
  [field-id ...]
  method)
```

Like `define-objc-class`, but defines a mixin to be combined with other method definitions through either `define-objc-class` or `define-objc-mixin`. The specified *field-id*s are not added by the mixin, but must be a subset of the *field-id*s declared for the class to which the methods are added.

```
self
```

When used within the body of a `define-objc-class` or `define-objc-mixin` method, refers to the object whose method was called. This form cannot be used outside of a `define-objc-class` or `define-objc-mixin` method.

```
(super-tell result-type method-id)
(super-tell result-type arg ...)
```

When used within the body of a `define-objc-class` or `define-objc-mixin` method, calls a superclass method. The *result-type* and *arg* sub-forms have the same syntax as in `tell`. This form cannot be used outside of a `define-objc-class` or `define-objc-mixin` method.

```
(get-ivar obj-expr field-id)
```

Extracts the Racket value of a field in a class created with `define-objc-class`.

```
(set-ivar! obj-expr field-id value-expr)
```

Sets the Racket value of a field in a class created with `define-objc-class`.

```
(selector method-id)
```

Returns a selector (of FFI type `_SEL`) for the string form of *method-id*.

Example:

```
> (tellv button setAction: #:type _SEL (selector terminate:))
```

```
(objc-is-a? obj cls) → boolean?
  obj : _id
  cls : _Class
```

Check whether *obj* is an instance of the Objective-C class *cls*.

5.8.3 Raw Runtime Functions

```
(objc_lookUpClass s) → (or/c _Class #f)
  s : string?
```

Finds a registered class by name.

```
(objc_getProtocol s) → (or/c _Protocol #f)
  s : string?
```

Finds a registered protocol by name.

```
(sel_registerName s) → _SEL
  s : string?
```

Interns a selector given its name in string form.

```
(objc_allocateClassPair cls s extra) → _Class
  cls : _Class
  s : string?
  extra : integer?
```

Allocates a new Objective-C class.

```
(objc_registerClassPair cls) → void?
  cls : _Class
```

Registers an Objective-C class.

```
(object_getClass obj) → _Class
  obj : _id
```

Returns the class of an object (or the meta-class of a class).

```
(class_addMethod cls
                 sel
                 imp
                 type
                 type-encoding) → boolean?
  cls : _Class
  sel : _SEL
  imp : procedure?
  type : ctype?
  type-encoding : string?
```

Adds a method to a class. The *type* argument must be a FFI C type (see §3.1 “Type Constructors”) that matches both *imp* and the not Objective-C type string *type-encoding*.

```
(class_addIvar cls
               name
               size
               log-alignment
               type-encoding) → boolean?
  cls : _Class
  name : string?
  size : exact-nonnegative-integer?
  log-alignment : exact-nonnegative-integer?
  type-encoding : string?
```

Adds an instance variable to an Objective-C class.

```
(object_getInstanceVariable obj name) → _Ivar any/c
  obj : _id
  name : string?
```

Gets the value of an instance variable whose type is `_pointer`.

```
(object_setInstanceVariable obj name val) → _Ivar
  obj : _id
  name : string?
  val : any/c
```

Sets the value of an instance variable whose type is `_pointer`.

```
_Ivar : ctype?
```

The type of an Objective-C instance variable, an opaque pointer.

```
((objc_msgSend/typed types) obj sel arg) → any/c
  types : (vector/c result-ctype arg-ctype ...)
  obj : _id
  sel : _SEL
  arg : any/c
```

Calls the Objective-C method on `_id` named by *sel*. The *types* vector must contain one more than the number of supplied *arg*s; the first FFI C type in `type` is used as the result type.

```
((objc_msgSendSuper/typed types)
                          super
                          sel
                          arg)   → any/c
  types : (vector/c result-ctype arg-ctype ...)
  super : _objc_super
  sel : _SEL
  arg : any/c
```

Like `objc_msgSend/typed`, but for a super call.

```
(make-objc_super id super) → _objc_super
  id : _id
  super : _Class
_objc_super : ctype?
```

Constructor and FFI C type use for super calls.

Contents

5.8.4 Legacy Library

```
(require ffi/objc)
```

The `ffi/objc` library is a deprecated entry point to `ffi/unsafe/objc`. It exports only safe operations directly, and unsafe operations are imported using `objc-unsafe!`.

```
(objc-unsafe!)
```

Analogous to `(unsafe!)`, makes unsafe bindings of `ffi/unsafe/objc` available in the importing module.

5.9 File Security-Guard Checks

```
(require ffi/file)
```

```
(security-guard-check-file who path perms) → void?
  who : symbol?
  path : path-string?
  perms : (listof (or/c 'read 'write 'execute 'delete 'exists))
```

Checks whether `(current-security-guard)` permits access to the file specified by *path* with the permissions *perms*. See `make-security-guard` for more information on *perms*.

The symbol *who* should be the name of the function on whose behalf the security check is performed; it is passed to the security guard to use in access-denied errors.

```
(_file/guard perms [who]) → ctype?
  perms : (listof (or/c 'read 'write 'execute 'delete 'exists))
  who : symbol? = '_file/guard
```

Like `_file` and `_path`, but conversion from Racket to C first completes the path using `path->complete-path` then cleanses it using `cleanse-path`, then checks that the current security guard grants access on the resulting complete path with *perms*. As an output value, identical to `_path`.

```
_file/r : ctype?
_file/rw : ctype?
```

Equivalent to `(_file/guard '(read))` and `(_file/guard '(read write))`, respectively.

5.10 Windows API Helpers

```
(require ffi/winapi)
```

```
win64? : boolean?
```

Indicates whether the current platform is 64-bit Windows: `#t` if so, `#f` otherwise.

```
winapi : (or/c 'stdcall 'default)
```

Suitable for use as an ABI specification for a Windows API function: `'stdcall` on 32-bit Windows, `'default` on 64-bit Windows or any other platform.

6 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 Racket 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 vec type) → any
  vec : vector?
  type : type?
```

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

```
(vector->cpointer vec) → cpointer?
  vec : vector?
```

Returns a pointer to an array of `_scheme` values, which is the internal representation of *vec*.

```
(flvector->cpointer flvec) → cpointer?
  flvec : flvector?
```

Returns a pointer to an array of `_double` values, which is the internal representation of *flvec*.

```
(saved-errno) → exact-integer?
```

Returns the value most recently saved (in the current thread) after a foreign call with a non-`#f` `#:save-errno` option (see `_fun` and `_cprocedure`).

```
(lookup-errno sym) → exact-integer?
  sym : (or/c 'EINTR 'EEXIST 'EAGAIN)
```

Returns a platform-specific value corresponding to a Posix `errno` symbol. The set of supported symbols is likely to expand in the future.

```
(cast v from-type to-type) → any/c
  v : any/c
  from-type : ctype?
  to-type : ctype?
```

Converts *v* from a value matching *from-type* to a value matching *to-type*, where `(ctype-sizeof` *from-type*`)` matches `(ctype-sizeof` *to-type*`)`.

The conversion is equivalent to

```
(let ([p (malloc from-type)])
  (ptr-set! p from-type v)
  (ptr-ref p to-type))
```

```
(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 Racket 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 Racket vectors.

7 Unexported Primitive Functions

Parts of the `ffi/unsafe` library are implemented by the Racket 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 Racket 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 Racket “callout” values. The given *ptr* (any pointer value, including `ffi-obj` values) is wrapped in a Racket-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 Racket 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`.

Bibliography

[Barzilay04] Eli Barzilay and Dmitry Orlovsky, “Foreign Interface for PLT Scheme,” Workshop on Scheme and Functional Programming, 2004.

Index