

Objective-C FFI

Version 4.2.3

December 1, 2009

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

The `ffi/objc` library builds on `scheme/foreign` 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 thin wrapper on the Objective-C runtime library functions. Even the upper layer is unsafe and relatively low-level compared to normal Scheme libraries, because argument and result types must be declared in terms of FFI C types (see §3.1 “Type Constructors”).

Important: Most of the bindings documented here are available only after an `(objc-unsafe!)` declaration in the importing module.

Contents

1	Using Unsafe Bindings	3
2	FFI Types and Constants	4
3	Syntactic Forms	5
4	Raw Runtime Functions	8

1 Using Unsafe Bindings

`(objc-unsafe!)`

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

2 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`.

`_SEL` : `ctype?`

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

`_BOOL` : `ctype?`

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

`YES` : `boolean?`

Synonym for `#t`

`NO` : `boolean?`

Synonym for `#f`

3 Syntactic Forms

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

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

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)
```

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

```
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.

Each *field-id* is an instance field that holds a Scheme value and that is initialized to `#f` when the object is allocated. The *field-ids* can be referenced and `set!` directly when the method *bodys*. 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
  (- _scheme (swapBitmap: [_scheme 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))))
```

`self`

When used within the body of a `define-objc-class` method, refers to the object whose

method was called. This form cannot be used outside of a `define-objc-class` method.

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

When used within the body of a `define-objc-class` 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` method.

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

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

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

Sets the Scheme 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:))
```

4 Raw Runtime Functions

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

Finds a registered class 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 `args`; 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.