Common Lisp ObjectiveC Interface

Documentation for the Common Lisp Objective C Interface, version 1.0.

Table of Contents

1 Introduction

1.1 What is CL-ObjC?

CL-ObjC is a Common Lisp library whose aim is to allow developers to use libraries written in the Objective C language, providing a lisp-like interface towards Objective C conventions.

Actually it is made of four different interfaces, i.e. four different style to use Objective C libraries in your Common Lisp program.

The first one provided by the package CL-OBJC gives to the users the ability to write application in a Lisp-like way with a functional interface in order to send message (e.g. call methods) to Objective C objects. E.g.:

```
(let ((new-string (invoke 'ns-string alloc)))
  (invoke new-string :init-with-utf8-string ``ciao''))

(define-objc-class bank-account ns-object
 ((amount :float)
  (currency currency-type)))

(define-objc-method withdraw ((self bank-account) (withdraw :float))
  (with-ivar-accessors bank-account
    (decf (amount self) withdraw)
    self))
```

The second one provided by OBJC-CLOS expose a CLOS object oriented interface to Objective C mapping every Objective C class/method to a CLOS class/method with names interned in a separate OBJC package. So the user can code in a CLOS fashion mixing the strength of both object systems. Functions to create bindings for custom Objective C frameworks are provided. E.g.

```
(let ((new-string (make-instance 'objc:ns-string)))
  (init-with-utf8-string? new-string ``ciao''))
```

The third one provided by the package OBJC-CFFI exposes the CFFI bindings to the user and allows to handle Objective C objects through Common Lisp objects via the CFFI foreign type translators mechanism. It is mainly useful when you have to work with C-like construct of Objective C, like function returning C struct value and so on.

The last one, you can find in the OBJC-READER package, provides a reader macro that allows user to mix ObjectiveC and Common Lisp syntax just to call Objective C methods. It is designed just to be used for the sake of convenience from the REPL.

This manual is a work in progress. If you have difficulty using or comprehending it please contact the developers with details.

1.2 Status

At the moment CL-ObjC has been tested mainly on x86-64 platform on Darwin with the last stable release of sbcl. CL-ObjC is known also to work on Allegro 8.1. As CL-ObjC uses CFFI to link with the Objective C runtime foreign functions, it should not be a problem to use it portably on other platforms, compilers or OSs.

1.3 Download and installation

You can get the current development version of CL-ObjC from the darcs repository on http://common-lisp.net/project/cl-objc/darcs/cl-objc/

CL-ObjC is packaged using the ASDF library, so once downloaded the code you should change the asdf:*central-registry* parameter to make asdf find the cl-objc.asd file, or you can make a symbolic link of it in a directory already in asdf:*central-registry*.

The first time you build CL-ObjC the asdf system will build the bindings for the standard frameworks present on your system. That operation can take a long time, until 5 minutes. Loot at the file generate-framework-bindings.lisp to configure this process.

To build CL-ObjC you need to get a recent (at least from February 2007) version of CFFI. You can get the development version from the darcs repository at http://common-lisp.net/project/cffi/darcs/cffi/.

In order to work CL-ObjC needs to find the libobjc library. By default it will search for it in /usr/lib. Change *foreign-library-directories* if you want to load libobjc from a different path.

To run tests bundled with CL-ObjC you need the FiveAM test unit software, then you can eval:

```
(asdf:oos 'asdf:load-op 'cl-objc)
(asdf:oos 'asdf:test-op 'cl-objc)
```

1.4 Authors

* Geoff Cant: initial author
* Luigi Panzeri: main developer

1.5 Acknowledgments

Thanks to the LispNyC (that approved and voted for this project) CL-ObjC was funded by Google Inc. in the Summer of Code of 2007. We thanks also Marco Baringer mentor of the project during the SoC. We thank also sbcl guys because part of this manual has been generated by the SB-TEXINFO software.

2 Tutorial

2.1 Quick start

The purpose of CL-ObjC is to allow to use Objective C with Lisp-like syntax. Actually we can do that with four equivalent way. For the impatient an ObjC call like:

```
[my_obj doSomethingCoolAt: now withThisParam 5]
```

can be equivalently be executed in CL-ObjC

* using the interface provided by the package CL-OBJC :

```
(invoke my-obj :do-something-cool-at now :with-this-param 5)
```

* using CLOS bindings

```
;; you can alloc new instances with (make-instance 'objc:ns-foo)

(objc:do-something-cool-at?-with-this-param? my-obj now 5)
```

* using the low level facility in the package OBJC-CFFI (rarely needed):

```
; specifying argument types
(typed-objc-msg-send (my-obj "doSomethingCoolAt:withThisParam")
     'objc-id now
     :int 5)
; without argument types
(untyped-objc-msg-send my-obj
                       "doSomethingCoolAt:withThisParam" now 5)
```

* using the reader macro in the package OBJC-READER :

```
[my-obj doSomethingCoolAt: now withThisParams 5]
```

So every message call can be specified in a typed or untyped way, i.e. expliciting the CFFI type of the arguments or not, or mixing the two approach. Actually if types are available the compiler can optimize the message call. CLOS bindings and the reader macro have to be enabled before use because they are disabled by default.

2.2 Loading Frameworks

Once loaded CL-ObjC we can use existing Objective C classes and methods loading frameworks with the macro `OBJC-CFFI:IMPORT-FRAMEWORK`:

```
(import-framework "Foundation")
```

If you want to enable the CLOS interface you need to set the `OBJC-CLOS:*AUTOMATIC-CLOS-BINDINGS-UPDATE*` to T before loading the framework.

See [Variable objc-clos:*automatic-clos-bindings-update*], page 15 and [Macro objc-cffi:import-framework], page 10 for details.

Please note that the first time you load CL-ObjC, bindings for standard Cocoa frameworks will be built. This process can take long time.

2.3 Type translations

Objective C types are translated directly into Common Lisp types. Primitive C types (`float`, `int`, `char *`) are translated by CFFI in the common way. Struct values as in CFFI are always managed by a pointer to their data. The specific Objective C types (like `id`) are translated automatically by foreign type translators into lisp types and viceversa when needed. Several CLOS classes are added to handle them and custom PRINT-OBJECT and DESCRIBE-OBJECT are provided in the OBJC-CFFI package. The following types translators are installed:

Objective C type	*Description*	*Lisp type*
`SEL`	selectors	`objc-selector`
`objc_class`	Class description	`objc-class`
`id`	Instance Object	`objc-object`
`objc_method`	Method Object	`objc-method`
`objc_method_list`	List of Methods	a list or an objc-method
`objc_ivar`	Instance variable	`objc-ivar`
`objc_ivar_list`	Instance variable lists	`objc-ivar-list`

You can also use normal string to specify classes or selectors in the OBJC-CFFI package and they will be translated to the type needed.

See Chapter 6 [CLOS Bindings], page 15 to read documentation about type translation with CLOS bindings.

2.4 Name translators

In the CL-OBJC package you don't specify class and selector by strings but with symbols. To see how a given ObjC or Lisp name will be translated by CL-ObjC, you can use the following functions:

```
OBJC-CFFI:OBJC-CLASS-NAME-TO-SYMBOL
OBJC-CFFI:SYMBOL-TO-OBJC-CLASS-NAME
CL-OBJC:OBJC-SELECTOR-TO-SYMBOLS
CL-OBJC:SYMBOLS-TO-OBJC-SELECTOR
```

So for example the class "NSObject" is translated to NS-OBJECT, the class "VeryImportant" to VERY-IMPORTANT and so on. You can add strings to the special variable CL-OBJC:*ACRONYMS* in order to not espand acronyms like URL into u-r-l.

A selector name can be translated in a list of symbols or a single symbol depending on the number of parameters. The following examples explain its behavior:

`[NSObject alloc]`
: (invoke 'ns-object alloc)

`[obj setX: 3 Y: 5]`
: (invoke obj :set-x 3 :y 5)

`[NSNumber numberWithInt: 3]`
: (invoke 'ns-number :number-with-int 3)

See Chapter 6 [CLOS Bindings], page 15 to read documentation about name translations with CLOS bindings.

2.5 Utilities

Several macros are present for the sake of convenience.

2.5.1 The SLET, SLET* macro

The CL-OBJC:SLET* and CL-OBJC:SLET* macro are used to handle data stored in structs. They bind variable to new allocated (or not if you provide a value) structs and provide lisp accessor to their fields.

```
(defun make-rect (x y width height)
  (destructuring-bind (x y width height)
   (mapcar #'float (list x y width height))
    (slet* ((rect ns-rect)
    (size ns-size (ns-rect-size rect))
    (point ns-point (ns-rect-origin rect)))
      (setf (ns-point-x point) x
    (ns-point-y point) y
    (ns-size-width size) width
    (ns-size-height size) height)
      rect))
```

See [Macro cl-objc:slet], page 13 for details.

2.5.2 WITH-OBJECT and OBJC-LET

As a common snippet Objective C code pattern is `[[MyClass alloc] initWithParam: param]` it is provided the macros CL-OBJC:OBJC-LET and CL-OBJC:OBJC-LET* that binds variables to new allocated and initialized object.

Another common situation is to call several methods with the same receiver (e.g. self). For the sake of convenience it is provided the CL-OBJC:WITH-OBJECT macro. So the Objective C code

```
obj = [ [MyClass alloc] initWithParam: param
[obj methodA]
[obj methodB: param1 C: param2]
[obj methodD: param1]
```

becomes in the CL-ObjC syntax:

```
(objc-let ((obj my-class :init-with-param param))
  (with-object obj
    (method-a)
    (:method-b param1)
    (:method-c param1 :c param2)
    (:method-d: param1)))
```

hw See [Macro cl-objc:objc-let], page 12 and [Macro cl-objc:with-object], page 13 for details.

2.5.3 Super Calling

In order to send message to the superclass of an instance of a class the macro WITH-SUPER is provided. See ⟨undefined⟩ [Macro objc-cffi:with-super], page ⟨undefined⟩.

2.6 Using Interface Builder: the Currency Converter Example

This section concerns the construction of a Cocoa application using an UI built with XCode. You can find the instructions to use the Interface Builder tool to build the interface used in the example at

http://developer.apple.com/documentation/Cocoa/Conceptual/ObjCTutorial/.

A Cocoa application is usually deployed as a bundle. A bundle is just a directory containing all the resources (images, sounds, data) the application needs. Interface Builder saves descriptions of UI in serialized objects saved in file in the NIB file format stored in the bundles.

If you follow the instructions of the quoted Cocoa tutorial you will find in the directory project your bundle, namingly a folder with the .app extension.

In order to have a Cocoa application using NIB files and integrated with OSX you have to link your common lisp compiler and a script launching your Cocoa lisp application into the "MacOS" subfolder of your bundle. Usually the script execute your compiler to load and launch your code. Otherwise if your lisp implementation supports the creation of a single binary with the image of your program, you can just copy it in the same subfolder.

Note that in that script you can create a swank server you can attach to allowing live debugging of your application[1].

Now let's go through the classic Currency Converter example. First of all, we load the framework we need:

```
(import-framework "Foundation")
(import-framework "AppKit)
(import-framework "Cocoa")
```

The second import statement binds the variable `*nsapp*` to a static variabile used by the Cocoa framework.

Then we can define the class definitions for the model and for the controller of the GUI using the DEFINE-OBJC-CLASS macro found in the CL-OBJC package.

```
(define-objc-class converter ns-object
  ())

(define-objc-class converter-controller ns-object
  ((converter converter)
   (first-currency-field ns-text-field)
   (other-currency-field ns-text-field)
   (rate-field ns-text-field)))
```

The DEFINE-OBJC-CLASS macro gets three arguments, the name of the class and of the superclass transformed with the name translators cited before, and the a list of instance variable definitions. Please note that you can specify the Objective C class of the instance variable, or just the "true" CFFI value, namingly objc-id.

So the `converter-controller` class will be added to the Objective C runtime class list with the name "ConverterController" and defines 4 instance variables with ObjC names: "converter", "firstCurrencyField", "otherCurrencyField", "rateField".

[1] You can find an example of this script in the cl-objc bundle

Then we can define the methods with the CL-OBJC:DEFINE-OBJC-METHOD macro:

```
(define-objc-method :convert (:return-type :void)
    ((self converter-controller) (sender objc-id))
  (declare (ignore sender))
  (with-ivar-accessors converter-controller
    (let* ((currency (invoke (first-currency-field self) float-value))
   (rate (invoke (rate-field self) float-value))
   (amount (invoke (converter self) :convert-currency currency
                                          :at-rate rate)))
      (invoke (other-currency-field self) :set-float-value amount)
      (invoke (rate-field self) :select-text self))))

(define-objc-method (:convert-currency :at-rate) (:return-type :float)
    ((self converter) (currency :float) (rate :float))
  (* currency rate))
```

This macro gets 3 arguments plus the body of the method. The name of the related selector specified with a symbol or a list of symbols, a list of option and a list of argument definition made of the argument name and its type. The argument named SELF is mandatory and will binded to the receiver of the message. The body will be evaluated also with the symbol SEL binded to the selector object of the method.

So the second definition in the example define the instance method named “convertCurrency:atRate:” returning a :float. It accepts 2 arguments and it is binded to the class converter.

The WITH-IVAR-ACCESSORS macro establishes useful accessors to read and modify instance variables of ObjectiveC objects.

At last the main entry point of our application looks like:

```
(defun converter ()
  (invoke 'ns-application shared-application)
  (invoke 'ns-bundle
          :load-nib-named (lisp-string-to-nsstring "MainMenu")
          :owner *nsapp*)
  (invoke *nsapp* run))
```

If we activate the reader macro we can specify `NSString` object using the at-sign so (lisp-string-to-nsstring “MainMenu”) becomes `@"`MainMenu”

3 CFFI Bindings

3.1 API Reference

`objc-cffi:add-objc-class` *class-name super-class* **&optional** *ivar-list* [Function]

Adds and returns a new ObjectiveC class `class-name` deriving from `super-class`.

`ivar-list` is a list of instance variable object that will be added to the new class.

If a class with the same name already exists the method raise an error of type `objc-class-already-exists`.

`objc-cffi:add-objc-method` (*name class* **&key** *return-type class-method*) [Macro]
argument-list **&body** *body*

Add an ObjectiveC method to `class` returning the `cffi return-type` and binding it to a selector with `name`. If `class-method` is true then a class method will be added to `class`.

`argument-list` is a list of list with two elements. The first one is the name of the argument, while the second is its `cffi` type.

In `body` are also bound the symbols `self` pointing to the receiver of the message and `sel` pointing to the selector.

If a method binded to `sel` is already present in `class` it installs the new definition discarding the previous one.

Return a new ObjectiveC Method object.

`objc-cffi:class-get-class-method` *class sel* [Function]

Return the class method binded of `class` to `sel`

`objc-cffi:class-get-instance-method` *class sel* [Function]

Return the instance method of `class` binded to `sel`

`objc-cffi:class-get-instance-variable` *class variable-name* [Function]

Returns the instance variable definition with `variable-name` of `class`

`objc-cffi:class-has-public-ivars` *class* [Function]

Returns the public vars of `class`

`objc-cffi:define-objc-struct` *name-and-objc-options* **&body** [Macro]
doc-and-slots

Wrapper for `cffi:defcstruct` allowing struct to be used as type. `doc-and-slots` will be passed directly to `cffi:defcstruct` while `name-and-objc-options` can be specified in one of the followings format (e.g. for the NSRect STRUCT):

`ns-rect` (NS-RECT 16) (NS-RECT "_NSRect") ((NS-RECT 16) "_NSRect"),

where _NSRect is the struct name used in ObjC methods, and `ns-rect` is the lisp name of the struct. If you don't specify the former the method will try to guess it automatically, but an error will be raised if the trial fails.

`objc-cffi:framework-bindings-pathname` *framework-name type* [Function]
Returns the pathname of the `type` bindings of `framework-name`. At the moment type can be 'clos or 'static.

`objc-cffi:get-class-methods` *class* [Function]
Returns all the class methods of `class`

`objc-cffi:get-class-ordered-list` [Function]
Returns the list of all the ObjectiveC Class available ordered by the number of superclass they have

`objc-cffi:get-instance-methods` *class* [Function]
Returns all the instance methods of `class`

`objc-cffi:get-ivar` *obj ivar-name* [Function]
Returns the value of instance variable named `ivar-name` of ObjectiveC object `obj`

`objc-cffi:make-ivar` *name type* [Function]
Returns a new instance variable object named `name` of `type`

`objc-cffi:method-get-size-of-arguments` *method* [Function]
Returns the size of all the arguments of an ObjC method

`objc-cffi:objc-class-name-to-symbol` *name* [Function]
Returns a symbol that can be used in CL-ObjC to identify the class named `name`.

`objc-cffi:objc-get-class` *name* [Function]
Returns the ObjectiveC Class named `name`

`objc-cffi:objc-nil-class` [Variable]
The Objective `c` Class nil

`objc-cffi:objc-nil-object` [Variable]
The ObjectiveC object instance nil

`objc-cffi:objc-nil-object-p` *obj* [Function]
Returns `true` if obj is a nil object

`objc-cffi:objc-object` [Class]
Class precedence list: `objc-object, standard-object, t`

Slots:

- `isa` — initarg: `:isa`; reader: `objc-cffi:obj-class`
  Returns the Objective `c` class of the specified object

Objective `c` Object Class

`objc-cffi:compile-framework` (*framework-name* **&key** *force clos-bindings*) **&body** *other-bindings* [Macro]

Create bindings for `framework-name`. Frameworks will be searched in `cffi:*darwin-framework-directories*`. The bindings will not be loaded.

`objc-cffi:ensure-objc-class` *class-name super-class* **&optional** *ivar-list* [Function]

Like add-objc-class but if a class with the same name already exists it just returns without adding the new class definition

`objc-cffi:get-class-list` [Function]

Returns the list of all the ObjectiveC Class available

`objc-cffi:import-framework` *framework-name* **&optional** *clos* [Macro]

Import the ObjC framework `framework-name`. If `clos` or `objc-clos:*automatic-clos-bindings-update*` is true then load also the `clos` bindings.

`objc-cffi:objc-struct-slot-value` *ptr type slot-name* [Function]

Return the value of `slot-name` in the ObjC Structure `type` at `ptr`.

`objc-cffi:private-ivar-p` *ivar* [Function]

Returns `true` if the instance variable is private.

`objc-cffi:sel-get-uid` *str* [Function]

Returns the selector named `name`.

`objc-cffi:sel-is-mapped` *sel* [Function]

Returns true if a selector is registered in the ObjC runtime.

`objc-cffi:set-ivar` *obj ivar-name value* [Function]

Set the value of instance variable named `ivar-name` of `obj` with `value`

`objc-cffi:super-classes` *item* [Generic Function]

Get the Super Classes of an Objc Object or of a class viewed as an instance of a Meta Class

`objc-cffi:symbol-to-objc-class-name` *symbol* [Function]

The inverse of `objc-class-name-to-symbol`.

`objc-cffi:typed-objc-msg-send` (*id sel* **&optional** *stret*) **&rest** *args-and-types* [Macro]

Send the message binded to selector `sel` to the object `id` returning the value of the ObjectiveC call.

`args-and-types` is a list of pairs. The first element of a pair is the `cffi` type and the second is the value of the argument passed to the method.

If the method return type is an ObjectiveC struct you can pass a pointer to a an allocated struct that will retain the value returned, otherwise a new struct will be allocated.

If `id` is an ObjectiveC class object it will call the class method binded to `sel`.

`objc-cffi:untyped-objc-msg-send` *receiver selector* **&rest** *args* [Function]
Send the message binded to `selector` to `receiver` returning the value of the ObjectiveC call with `args`.

This method invokes typed-objc-msg-send calculating the types of `args` at runtime.

FIXME: exported accessors need documentation too!

4 Lisp Interface

4.1 API Reference

`cl-objc:*acronyms*` [Variable]
Acronyms used in name translators

`cl-objc:define-objc-class` *symbol-name symbol-superclass* (**&rest** *ivars*) [Macro]
Define and returns a new ObjectiveC class `symbol-name` deriving from `symbol-superclass`. Names are translated by `symbol-to-objc-class-name`.

`ivars` is a list of pairs where the first element is the variable name (translated by symbols-to-objc-selector) and the second on is the `cffi` type of the variable or a name of an ObjC class/struct translated by objc-class-name-to-symbol.

If a class with the same name already exists the method returns without adding the new definition.

`cl-objc:define-objc-method` *list-selector* (**&key** *return-type class-method*) (**&rest argument-list**) **&body** *body* [Macro]
Add an ObjectiveC method binded to a selector defined by `list-selector` (translated by SYMBOLS-TO-OBJC-SELECTOR), returning the `cffi return-type`. If the method get zero or one argument then `list-selector` can be specified with an atom.

If `class-method` is true then a class method will be added.

`argument-list` is a list of list with two elements. The first one is the name of the argument, while the second is its ObjectiveC type. The first pair has to have as first element the symbol self and as second element the class which the method will be added to.

In `body` is also bound the symbol 'SEL' pointing to the selector.

If a method binded to the same selector is already present it installs the new definition discarding the previous one.

Return a new ObjectiveC Method object.

`cl-objc:invoke` *receiver* **&rest** *selector-and-args* [Macro]
Call an ObjectiveC message to `receiver`.

`receiver` can be an ObjectiveC object or class. For the sake of convenience if `receiver` is a symbol it will be mapped to an ObjectiveC class through the `objc-class-name-to-symbol` translator.

`selector-and-args` has the form {selector-part [cffi-type] value}*.

e.g. (invoke 'ns-number alloc) (invoke 'ns-value :value-with-number 3) (invoke 'ns-number :number-with-int :int 4) (invoke (invoke 'ns-windo alloc) :init-with-content-rect frame :style-mask 15 :backing 2 :defer 0) (invoke win :init-with-content-rect frame :style-mask :int 15 :backing :int 2 :defer 0)

`cl-objc:objc-let` *bindings* **&body** *body* [Macro]
objc-let create new variable bindings to new ObjectiveC object, instantiated by the alloc method, and execute a series of forms in `body` that use these bindings.

`bindings` has the form: ((var symbol-class-type [init-form])*).

`var` will be initialized calling `invoke` with 'INIT-FORM' as arguments.

e.g. (objc-let ((num 'ns-number :number-with-float 3.0)) (invoke float-value))

`cl-objc:objc-let*` *bindings* **&body** *body* [Macro]
Serial version of `objc-let`. See its documentation

`cl-objc:objc-selector-to-symbols` **&rest** *args* [Function]
The inverse of `symbols-to-objc-selector`

`cl-objc:selector` **&rest** *symbols* [Function]
Returns the selector object associated to symbols through `symbols-to-objc-selector`.

`cl-objc:slet` *bindings* **&body** *body* [Macro]
slet and slet* create new variable `bindings` to ObjectiveC structs and execute a series of forms in `body` that use these bindings. slet performs the bindings in parallel and slet* does them sequentially.

`bindings` has the form: ((VAR `struct-type` [INIT-FORM])*).

`var` will be binded to `init-form` if present otherwise to a new allocated struct of type `struct-type` (translated by OBJC-CLASS-NAME-TO-SYMBOL).

In body accessories of the form (STRUCT-NAME'-'SLOT-NAME STRUCT-OBJ) will be bound as utilities.

`cl-objc:slet*` *bindings* **&body** *body* [Macro]
See documentation of `slet`

`cl-objc:symbols-to-objc-selector` *lst* [Function]
Translate a list of symbols to a string naming a translator

`cl-objc:with-ivar-accessors` *symbol-class* **&body** *body* [Macro]
Execute `body` with bindings to accessors of the form (CLASS-NAME'-'IVAR-NAME)

`cl-objc:with-object` *obj* **&body** *actions* [Macro]
Calls messages with `obj` as receveir. `actions` is a list of selector and arguments passed to invoke.

5 Reader Macro

5.1 Introduction

The reader macro allows the user to mix ObjectiveC and Common Lisp code.

When the macro dispatch character "[" is readed, then the reader excepts a symbol naming a class (read preserving case), then selectors and arguments. Arguments are read using the default Common Lisp readtable. In order to send methods to instance objects you can use the "," (comma) character to read and eval symbols with the standard readtable. For the sake of convenience it is provided the `@` (at-sign) macro that build *NSString* objects.

5.2 API Reference

`objc-reader:*accept-untyped-call*` [Variable]

If nil, methods have to be invoked with input type parameters.

`objc-reader:activate-objc-reader-macro` **&optional** [Function]

accept-untyped-call use-clos-interface

Installs a the ObjectiveC readtable. If `accept-untyped-call` is `nil` method has to be invoked with input type parameters. It saves the current readtable to be later restored with `restore-readtable`

`objc-reader:restore-readtable` [Function]

Restore the readtable being present before the call of `activate-objc-reader-macro`.

6 CLOS Bindings

6.1 Enabling CLOS bindings and creating bindings for Objective C frameworks

If you want to enable the CLOS interface you need to set the `OBJC-CLOS:*AUTOMATIC-CLOS-BINDINGS-UPDATE*` (See [Variable objc-clos:*automatic-clos-bindings-update*], page 15 for details) to T before loading the framework.

After setting the variable the call to `import-framework` takes few seconds to load thousands of methods typically included in an Objective C framework. In order to use non standard or not yet included frameworks you can use the macro `OBJC-CFFI:COMPILE-FRAMEWORK`([Macro objc-cffi:compile-framework], page 9) that is able at compile time to collect informations and generate CLOS classes and methods definitions, saving them in a compiled form you can load with `IMPORT-FRAMEWORK`.

`OBJC-CFFI:COMPILE-FRAMEWORK` is also used (see the file generate-framework-bindings.lisp for details) the first time you load CL-ObjC when bindings for standard Cocoa framewors are built. This process can take long time.

If you want at any time sync the CLOS bindings without setting you can use `objc-clos:update-clos-bindings`(See [Function objc-clos:update-clos-bindings], page 16 for details).

Tipically in a `COMPILE-FRAMEWORK` form you should declare struct and any function returning struct by value. You can do that with the macro `DEFINE-OBJC-STRUCT` and `DEFINE-OBJC-FUNCTION`. Please note that these macros export the function and struct names and slot names into the `CL-OBJC` package.

See [Macro objc-cffi:define-objc-struct], page 8 and ⟨undefined⟩ [Macro objc-cffi:define-objc-function], page ⟨undefined⟩ for details.

6.2 Value Translation

If you use the CLOS interface the return value of Objective C calls will be translated to CLOS instances of the corresponding class using CONVERT-RESULT-FROM-OBJC([Function objc-clos:convert-result-from-objc], page 15). So a call to

```
(objc:init-with-utf8-string? s "foo")
```

returns a new instance of the CLOS class OBJC:NS-C-F-STRING corresponding to the ObjectiveC class `NSCFString`.

6.3 API Reference

`objc-clos:*automatic-clos-bindings-update*` [Variable]

Set this to t if you want that clos bindings will be updated every time you add classes, method or load libraries.

`objc-clos:convert-result-from-objc` *ret* [Function]

Convert the returned value of an Objc Method to a lisp value (CLOS instance or primitive type)

`objc-clos:update-clos-bindings` **&key** *output-stream force for-framework* [Function]

Generate `clos` classes/generic function for each ObjC class/method behaving to `for-framework`. The default behavior is to not redefine a class/generic function if it is already defined, except if `force` is set. `update-clos-bindings` writes the bindings on `output-stream` if provided.

7 Implementation Notes

7.1 Limits

* Passing structs by value is supported with an ugly hack to overcome the actual limits of CFFI. We splay args in integer values, so it is platform-dependent (only for x86 ABI)

* Use of not exported symbols of CFFI. As ObjC use a different convention to push arguments for variadic functions (it doesn't do argument promotion), we can't use normal defcfun but we are using internal methods of CFFI. We use not exported symbols also to get info about foreign types.

* Exception management is missing because we don't know how to catch SIGTRAP signals in a portable way

* Circle View example doesn't work

Index

C

C

O

O