CFFI-Objects

Roman Klochkov, monk@slavsoft.surgut.ru

2012

Contents

1 Introduction

This document describes CFFI-objects: library, that extends CFFI to support structures, objects and reference parameters.

Other alternatives are Virgil and FSBV/cffi-libffi. Virgil tend to marshall all data back and forth. There are no support for structures as pointers. FSBV is obsoleted by cffi-libffi. Libffi I dislike, because it gives another layer of indirection (so make it slower) without new features (no bit fields in structures).

So I made my own library. It gives the opportunity for programmer to say which structures should be return values and how to save every particular structure – as pointer or as a lisp value.

Example:

```
(defcstruct* foo (bar :int) (baz :int))
(defvar foo-as-ptr (make-instance 'foo :new-struct t))
(defvar foo-as-value (make-instance 'foo))

(defcfun foo-maker (struct foo))
(defcfun proceed-foo :void (param (struct foo :out t)))
(defcfun print-foo :void (param (struct foo)))
```

Here you can use either `foo-as-ptr` or `foo-as-value` in all functions. `Foo-as-ptr` is faster, because it shouldn't convert values from Lisp to C and back, but if foreign pointer is not considered stable (may be freed by another c-function) or you don't want to control, when you need to free foreign pointer, you should use `foo-as-value`.

2 Float numbers, keywords, pathnames

Method expand-to-foreign-dyn

With plain CFFI language become slightly bondage. In lisp i have number, real and integer, but in CFFI only floats and ints. So, for example, this code is wrong

```
 (defcfun sin :double (x :double))
 (sin 0)
should be
 (sin 0.0d0)
```

I think, that this is unnnecessary. So here is my hack (it is hack, because it uses not exported symbols). It makes :double and :float to work, as if corresponding parameters coerced to the needed type.

```
[00001] (defmethod expand-to-foreign-dyn (value var body
[00002]                                   (type cffi::foreign-built-in-type))
[00003]   `(let ((,var
[00004]           ,(case (cffi::type-keyword type)
[00005]                  (:double `(coerce ,value 'double-float))
[00006]                  (:float `(coerce ,value 'single-float))
[00007]                  (t value))
[00008]            ))
[00009]      ,@body))
```

Class cffi-keyword

Constant-like strings often used in C, particulary in GTK. It is good to use lisp symbols in this case. So `cffi-keyword` type use symbol name as a string for C parameter. The name is downcased, because there are more string in downcase, than in upcase (for not downcased string you still may put string as is). Typical case for this type is using lisp keyword. So the name.

```
[00010] (define-foreign-type cffi-keyword (freeable)
[00011]   ()
[00012]   (:simple-parser cffi-keyword)
[00013]   (:actual-type :pointer))
[00014]
[00015] (defmethod translate-to-foreign ((value symbol) (type cffi-keyword))
[00016]   (foreign-string-alloc (string-downcase value)))
[00017]
[00018] (defmethod translate-to-foreign ((value string) (type cffi-keyword))
[00019]   (foreign-string-alloc value))
[00020]
[00021] (defmethod free-ptr ((type (eql 'cffi-keyword)) ptr)
[00022]   (foreign-string-free ptr))
```

Class cffi-pathname

The same case for pathnames. If C function expect path to file, you may send it as a string or as a lisp pathname.

```
[00023] (define-foreign-type cffi-pathname (freeable)
[00024]   ()
[00025]   (:simple-parser cffi-pathname)
[00026]   (:actual-type :string))
[00027]
[00028] (defmethod translate-to-foreign ((value pathname) (type cffi-pathname))
[00029]   (convert-to-foreign (namestring value) :string))
[00030]
[00031] (defmethod translate-to-foreign ((value string) (type cffi-pathname))
[00032]   (convert-to-foreign value :string))
[00033]
```

3 Memory freeing automation

Most of new CFFI types introduced in my library will live in the dynamic memory. There are different policies of memory control in different languages and libraries. Sometimes caller should clean memory (like in GTK), sometimes callee.

In any case programmer should have possibility to say, if he would like to free memory after function call. For example, in GTK it is common for callback to return a newly-allocated string or structure, but in parameters responsibility to clean memory remains to caller.

Another common option for any type is a flag, that it is out-paramter, so value of it should be translated back before freeing,

For uniformity with CFFI :string I chose :free-from-foreign and :free-to-foreign boolean flags to show, when we want to free memory. By default "caller frees" model is used.

Class freeable-base

- `freeable-base` introduces all necessary fields and handlers
- `freeable` have ready cffi-translator methods

```
[00001] (define-foreign-type freeable-base ()
[00002]   ;; Should we free after translating from foreign?
[00003]   ((free-from-foreign :initarg :free-from-foreign
[00004]                       :reader fst-free-from-foreign-p
[00005]                       :initform nil :type boolean)
[00006]    ;; Should we free after translating to foreign?
[00007]    (free-to-foreign :initarg :free-to-foreign
[00008]                     :reader fst-free-to-foreign-p
[00009]                     :initform t :type boolean)))
```

Interface to `freeable-base` consists of three generics for describing, how to free particular type: `free-ptr`, `free-sent-ptr` and `free-returned-ptr`, and two functions to use in CFFI translators: `free-returned-if-needed` and `free-sent-if-needed`.

Generic free-ptr (type ptr)

This generic describes, how to free an object with CFFI type `type` and pointer `ptr`. As `type` should be a symbol, you should specialize this generic with EQL specifier if your objects shouldn't be freed with `foreign-free`.

One can ask, why normal specializer by type of object and `object` as a first parameter is not used. Such strange API is developed, because `free-ptr` is used in `trivial-garbage:finalize` and in some implementation (for example, SBCL) finalizer shouldn't have reference to finalized object.

If you dislike it and you will not use finalizers, simply specialize or redefine `free-sent-ptr` and `free-returned-ptr`

```
[00010] (defgeneric free-ptr (type ptr)
[00011]   (:documentation "Called to free ptr, unless overriden free-sent-ptr
[00012] or free-returned-ptr. TYPE should be symbol and be specialized with EQL")
[00013]   (:method (type ptr) (foreign-free ptr)))
```

Generic free-sent-ptr

This generic describes, how to free an object with CFFI type `type` and pointer `ptr` after sending to foreign space. It has the same parameters as `cffi:free-translated-object`. If complex foreign type has additional conditionals or any additional actions when freeing, specialize it on you type.

Please, don't call it directly. Use `free-sent-if-needed` instead.

```
[00014] (defgeneric free-sent-ptr (cffi-type ptr param)
[00015]   (:documentation "Will be called in free-translated-object.
[00016] CFFI-TYPE: type defined with define-foreign-type.
[00017] PTR: foreign pointer
[00018] PARAM: third parameter of free-translated-object ==
[00019]        returned second value of translate-to-foreign.")
[00020]   (:method ((cffi-type freeable-base) ptr param)
[00021]     (unless (null-pointer-p ptr)
[00022]       (free-ptr (type-of cffi-type) ptr))))
```

Generic free-returned-ptr

This generic describes, how to free an object with CFFI type `type` and pointer `ptr` after receiving from foreign space. It has the same parameters as `cffi:translate-to-foreign`. If complex foreign type has additional conditionals or any additional actions when freeing, specialize it on you type.

Please, don't call it directly. Use `free-returned-if-needed` instead.

```
[00023] (defgeneric free-returned-ptr (cffi-type ptr)
[00024]   (:documentation "Will be called in translate-from-foreign after conversion.
[00025] CFFI-TYPE: type defined with define-foreign-type.
[00026] PTR: foreign pointer")
[00027]   (:method ((cffi-type freeable-base) ptr)
[00028]     (unless (null-pointer-p ptr)
[00029]       (free-ptr (type-of cffi-type) ptr))))
```

¡function free-sent-if-needed

```
[00030] (defun free-sent-if-needed (cffi-type ptr param)
[00031]   "This function should be placed in appropriate place of
[00032] free-translated-object"
[00033]   (when (fst-free-to-foreign-p cffi-type)
[00034]     (free-sent-ptr cffi-type ptr param)))
```

¡function free-returned-if-needed

```
[00035] (defun free-returned-if-needed (cffi-type ptr)
[00036]   "This function should be placed in appropriate place of
[00037] translate-from-foreign"
[00038]   (when (fst-free-from-foreign-p cffi-type)
[00039]     (free-returned-ptr cffi-type ptr)))
```

Class freeable

This is standard base class for freeable pointers. If you happy with default free algorithm, which implies, that `free-sent-ptr` is called after `free-translated-object`

when type described with `:free-to-foreign t` and `free-returned-ptr` is called when type described with `:free-from-foreign t` after `translate-from-foreign`.

If you need more complicated logic (for example, to free object in translate-from-foreign, not after), you should inherit your class from `freeable-base` and call `free-sent-if-needed` from `free-translated-object` and `free-returned-if-needed` from `translate-from-foreign`.

```
[00040] (defclass freeable (freeable-base) ()
[00041]   (:documentation "Mixing to auto-set translators"))
[00042]
[00043] (defmethod free-translated-object :after (ptr (type freeable) param)
[00044]   (free-sent-if-needed type ptr param))
[00045]
[00046] (defmethod translate-from-foreign :after (ptr (type freeable))
[00047]   (free-returned-if-needed type ptr))
```

Class freeable-out

This is standard base class for objects, that should be copied back to lisp after foreign function: so-called "out parameters".

For every class, inherited from `freeable-out`, you must implement method `copy-from-foreign`.

Then user of your class may set `:out t` in initargs for your class and be sure, that all changed data will be copied back to the variables.

When implementing `translate-to-foreign` you must return (values ptr value), because second value will be passed to `free-translated-object`, and then as PLACE to `copy-from-foreign`.

```
[00048] (define-foreign-type freeable-out (freeable)
[00049]   ((out :accessor object-out :initarg :out :initform nil
[00050]         :documentation "This is out param (for fill in foreign side)"))
[00051]   (:documentation "For returning data in out params.
[00052] If OUT is t, then translate-to-foreign MUST return (values ptr place)"))
```

Generic copy-from-foreign

This generic must have an implementation for every class inherited from `freeable-out`.

```
[00053] (defgeneric copy-from-foreign (cffi-type ptr place)
[00054]   (:documentation "Transfers data from pointer PTR to PLACE.
[00055] CFFI-TYPE: type defined with define-foreign-type.
[00056] PTR: foreign pointer
[00057] PLACE: third parameter of free-translated-object ==
[00058]        returned second value of translate-to-foreign"))
[00059]
[00060] (defmethod free-translated-object :before (ptr (type freeable-out) place)
[00061]   (when (object-out type)
```

4 Package definition

We unexport all symbols before `defpackage`, because CFFI-objects will be a drop-in replacemant for CFFI and I don't want to export by hand all symbols exported by CFFI.

```
[00001] (eval-when (:compile-toplevel :load-toplevel)
[00002]   (let ((p (find-package "CFFI-OBJECTS")))
[00003]     (when p
[00004]       (do-external-symbols (v p)
[00005]         (unexport (list v) p)))))
[00006]
[00007] (defpackage #:cffi-objects
[00008]   (:use #:common-lisp #:cffi)
[00009]   (:export
[00010]    #:freeable-base
[00011]    ;; slots
[00012]    #:free-from-foreign
[00013]    #:free-to-foreign
[00014]    ;; freeable-base API
[00015]    #:free-sent-if-needed
[00016]    #:free-returned-if-needed
[00017]    #:free-ptr
[00018]    #:free-sent-ptr
[00019]    #:free-returned-ptr
[00020]
[00021]    #:freeable
[00022]    #:freeable-out
[00023]    #:copy-from-foreign
[00024]
[00025]    #:gconstructor
[00026]
[00027]    #:object
[00028]    #:free-after
[00029]    #:find-object
[00030]    #:object-by-id
[00031]    #:initialize
[00032]    #:*objects*
[00033]    #:*objects-ids*
[00034]    #:object-class
[00035]    #:volatile
[00036]    ;; slots
[00037]    #:pointer
[00038]    ;; methods
[00039]    #:free
[00040]
[00041]    #:*array-length*
[00042]    ;; types
[00043]    #:pstring
[00044]    #:pfunction
```

```
[00045]    #:cffi-object
[00046]    #:cffi-array
[00047]    #:cffi-null-array
[00048]    #:carray
[00049]    #:null-array
[00050]    #:string-array
[00051]
[00052]    #:cffi-keyword
[00053]    #:cffi-pathname
[00054]    #:cffi-string
[00055]
[00056]    #:struct
[00057]    #:cffi-struct
[00058]    #:new-struct
[00059]    #:free-struct
[00060]
[00061]    #:defcstruct-accessors
[00062]    #:defcstruct*
[00063]    #:defbitaccessors
[00064]
[00065]    #:with-foreign-out
[00066]    #:with-foreign-outs
[00067]    #:with-foreign-outs-list
[00068]
[00069]    #:pair
[00070]    #:setf-init
[00071]    #:init-slots
[00072]    #:save-setter
[00073]    #:remove-setter
[00074]    #:clear-setters))
```

Now simply reexport all CFFI symbols.

```
[00075] (eval-when (:compile-toplevel :load-toplevel)
[00076]   (let ((cffi (find-package "CFFI"))
[00077]         (cffi-objects (find-package "CFFI-OBJECTS")))
[00078]     (do-external-symbols (v cffi)
[00079]       (export (list v) cffi-objects))))
[00080]
```