c-lambda: C FFI via raco ctool

Version 5.1.3

August 15, 2011

```
(require compiler/cffi)
```

The compiler/cffi module relies on a C compiler to statically construct an interface to C code through directives embedded in a Racket program. The library implements a subset of Gambit-C's foreign-function interface [Feeley98].

The ffi/unsafe library is a better interface for most tasks; see *The Racket Foreign Interface* for more information on ffi/unsafe. See also *Inside: Racket C API*, which describes Racket's C-level API for extending the run-time system.

The compiler/cffi library defines three forms: c-lambda, c-declare, and c-include. When interpreted directly or compiled to byte code, c-lambda produces a function that always raises exn:fail, and c-declare and c-include raise exn:fail. When compiled by raco ctool --extension, the forms provide access to C. Thus, compiler/cffi is normally required by a module to be compiled via raco ctool. In addition, the raco ctool compiler implicitly imports compiler/cffi into the top-level environment for non-module compilation.

The c-lambda form creates a Racket procedure whose body is implemented in C. Instead of declaring argument names, a c-lambda form declares argument types, as well as a return type. The implementation can be simply the name of a C function, as in the following definition of fmod:

```
(define fmod (c-lambda (double double) double "fmod"))
```

Alternatively, the implementation can be C code to serve as the body of a function, where the arguments are bound to ___arg1 (three underscores), etc., and the result is installed into ___result (three underscores):

```
(define machine-string->float
  (c-lambda (char-string) float
    "___result = *(float *)___arg1;"))
```

The `c-lambda` form provides only limited conversions between C and Racket data. For example, the following function does not reliably produce a string of four characters:

```
(define broken-machine-float->string
  (c-lambda (float) char-string
    "char b[5]; *(float *)b = ___arg1; b[4] = 0; ___result = b;"))
```

because the representation of a `float` can contain null bytes, which terminate the string. However, the full Racket API, which is described in *Inside: Racket C API*, can be used in a function body:

```
(define machine-float->string
  (c-lambda (float) racket-object
    "char b[4];"
    "*(float *)b = ___arg1;"
    "___result = racket_make_sized_byte_string(b, 4, 1);"))
```

The `c-declare` form declares arbitrary C code to appear after `"escheme.h"` or `"scheme.h"` is included, but before any other code in the compilation environment of the declaration. It is often used to declare C header file inclusions. For example, a proper definition of `fmod` needs the `"math.h"` header file:

```
(c-declare "#include <math.h>")
(define fmod (c-lambda (double double) double "fmod"))
```

The `c-declare` form can also be used to define helper C functions to be called through `c-lambda`.

The `c-include` form expands to a `c-declare` form using the content of a specified file. Use `(c-include file)` instead of `(c-declare "#include file")` when it's easier to have Racket resolve the file path than to have the C compiler resolve it.

The `"collects/mzscheme/examples"` directory in the Racket distribution contains additional examples.

When compiling for Racket 3m (see *Inside: Racket C API*), C code inserted by `c-lambda`, `c-declare`, and `c-include` will be transformed in the same was as `raco ctool`'s `--xform` mode (which may or may not be enough to make the code work correctly in Racket 3m; see *Inside: Racket C API* for more information).

```
(c-lambda (argument-type ...) return-type impl-string ...+)
```

Creates a Racket procedure whose body is implemented in C. The procedure takes as many arguments as the supplied `argument-type`s, and it returns one value. If `return-type` is void, the procedure's result is always void. The *impl-string* is either the name of a C function (or macro) or the body of a C function.

If a single *impl-string* is provided, and if it is a string containing only alphanumeric characters and _, then the created Racket procedure passes all of its arguments to the named C function (or macro) and returns the function's result. Each argument to the Racket procedure is converted according to the corresponding *argument-type* (as described below) to produce an argument to the C function. Unless *return-type* is void, the C function's result is converted according to *return-type* for the Racket procedure's result.

If more than *impl-string* is provided, or if it contains more than alphanumeric characters and _, then the concatenated *impl-string*s must contain C code to implement the function body. The converted arguments for the function will be in variables `___arg1`, `___arg2`, ... (with three underscores in each name) in the context where the *impl-string*s are placed for compilation. Unless *return-type* is void, the *impl-string*s code should assign a result to the variable `___result` (three underscores), which will be declared but not initialized. The *impl-string*s code should not return explicitly; control should always reach the end of the body. If the *impl-string*s code defines the pre-processor macro `___AT_END` (with three leading underscores), then the macro's value should be C code to execute after the value `___result` is converted to a Racket result, but before the result is returned, all in the same block; defining `___AT_END` is primarily useful for deallocating a string in `___result` that has been copied by conversion. The *impl-string*s code will start on a new line at the beginning of a block in its compilation context, and `___AT_END` will be undefined after the code.

In addition to `___arg1`, etc., the variable `argc` is bound in *impl-string*s to the number of arguments supplied to the function, and `argv` is bound to a `Racket_Object*` array of length `argc` containing the function arguments as Racket values. The `argv` and `argc` variables are mainly useful for error reporting (e.g., with `racket_wrong_type`).

Each *argument-type* must be one of the following, which are recognized symbolically:

- bool
  Racket range: any value
  C type: `int`
  Racket to C conversion: #f → 0, anything else → 1
  C to Racket conversion: 0 → #f, anything else → #t
- char
  Racket range: character
  C type: `char`
  Racket to C conversion: character's Latin-1 value cast to signed byte
  C to Racket conversion: Latin-1 value from unsigned cast mapped to character
- unsigned-char
  Racket range: character
  C type: `unsigned char`

Racket to C conversion: character's Latin-1 value
C to Racket conversion: Latin-1 value mapped to character

- `signed-char`
  Racket range: character
  C type: `signed char`
  Racket to C conversion: character's Latin-1 value cast to signed byte
  C to Racket conversion: Latin-1 value from unsigned cast mapped to character
- `int`
  Racket range: exact integer that fits into an `int`
  C type: `int`
  conversions: (obvious and precise)
- `unsigned-int`
  Racket range: exact integer that fits into an `unsigned int`
  C type: `unsigned int`
  conversions: (obvious and precise)
- `long`
  Racket range: exact integer that fits into a `long`
  C type: `long`
  conversions: (obvious and precise)
- `unsigned-long`
  Racket range: exact integer that fits into an `unsigned long`
  C type: `unsigned long`
  conversions: (obvious and precise)
- `short`
  Racket range: exact integer that fits into a `short`
  C type: `short`
  conversions: (obvious and precise)
- `unsigned-short`
  Racket range: exact integer that fits into an `unsigned short`
  C type: `unsigned short`
  conversions: (obvious and precise)
- `float`
  Racket range: real number
  C type: `float`
  Racket to C conversion: number converted to inexact and cast to `float`
  C to Racket conversion: cast to `double` and encapsulated as an inexact number
- `double`
  Racket range: real number
  C type: `double`
  Racket to C conversion: number converted to inexact
  C to Racket conversion: encapsulated as an inexact number

- `char-string`
  Racket range: byte string or `#f`
  C type: `char*`
  Racket to C conversion: string → contained byte-array pointer, `#f` → `NULL`
  C to Racket conversion: `NULL` → `#f`, anything else → new byte string created by copying the string

- `nonnull-char-string`
  Racket range: byte string
  C type: `char*`
  Racket to C conversion: byte string's contained byte-array pointer
  C to Racket conversion: new byte string created by copying the string

- `racket-object`
  Racket range: any value
  C type: `Racket_Object*`
  Racket to C conversion: no conversion
  C to Racket conversion: no conversion

- `(pointer bstr)`
  Racket range: an opaque c-pointer value, identified as type `bstr`, or `#f`
  C type: *`bstr`*`*`
  Racket to C conversion: `#f` → `NULL`, c-pointer → contained pointer cast to *`bstr`*`*`
  C to Racket conversion: `NULL` → `#f`, anything else → new c-pointer containing the pointer and identified as type *`bstr`*

The *`return-type`* must be `void` or one of the `arg-type` keywords.

`(c-declare code-string)`

Declares arbitrary C code to appear after `"escheme.h"` or `"scheme.h"` is included, but before any other code in the compilation environment of the declaration. A `c-declare` form can appear only at the top-level or within a module's top-level sequence.

The `code` code will appear on a new line in the file for C compilation. Multiple `c-include` declarations are concatenated (with newlines) in order to produce a sequence of declarations.

`(c-include path-spec)`

Expands to a use of `c-declare` with the content of *`path-spec`*. The *`path-spec`* has the same form as for `mzlib/include`'s `include`.

Bibliography

[Feeley98] Marc Feeley, “Gambit-C, version 3.0.” 1998.