Virgil User Manual

Table of Contents

1 Introduction

Virgil is an extensible and high-level foreign function interface(FFI) built on top of CFFI and oriented towards marshaling lisp data into raw unmanaged memory and back.

Why the name 'Virgil'? Well, have you read Dante's 'Divine Comedy'? :)

Rationale

Why another FFI? CFFI seems perfect in terms of portability, but it exposes quite a low-level interface. CFFI is oriented towards manipulating foreign memory, and forces us to write 'C-style' code in Lisp. Remember the old joke - "You can write FORTRAN in any language"? Using modern FFIs you can also write 'C' in any language - but should you?

Virgil, as opposed to CFFI, is oriented towards marshaling. This means, Virgil does its best to free the programmer from messing up with pointers and the like, and allows to communicate with 'native' code using Lisp data structures.

Thus, the main difference between CFFI and Virgil is that Virgil provides convenient semantics for marshaling aggregate data types and strives to establish a one-to-one mapping between lisp types and foreign ones.

Nevertheless, Virgil's interface is actually a bit similiar to that of CFFI, so you can easily start using it if you are familiar with the latter.

> **Implementor's note:** *Virgil is not a kind of a replacement for CFFI, but instead a kind of a DSL on top of it.*

2 Installation and Prerequisites

Sources are available on github:

- https://github.com/Lovesan/virgil

You can obtain them either using git, or by downloading latest zipball:

- https://github.com/Lovesan/virgil/downloads

Virgil depends on `CFFI`, `alexandria`, `babel` and `trivial-features`. Note that CFFI itself depends on the other three libraries.

You can obtain all of them from their home pages:

- http://common-lisp.net/project/cffi/
- http://common-lisp.net/project/babel/
- http://common-lisp.net/project/alexandria/
- http://www.cliki.net/trivial-features

but i recommend to use Zach Bean's quicklisp - `(ql:quickload :cffi)`

Note that at the moment Virgil only supports x86 and x86-64 platforms.[1]

[1] Mainly because of alignment conventions and because i am planning to add 'by-value' passage of aggregate function parameters

3 Tutorial

TODO

4 User-level API

At the user level, marshaling engine utilizes a concept of `type specifier`, or `typespec` for short. A `typespec` is an s-expression denoting some marshling rules to be applied to some 'foreign' value, such as a pointer, to translate it into lisp, or to some lisp value to convert it to foreign one.

Virgil's `type specifiers` are somewhat similiar to Common Lisp `type specifiers`, in the sense that they are represented either by a symbol denoting a type name or by a list, whose first elment is such a symbol.

Virgil exposes a deftype-like macro for establishing aliases for `typespecs`:

`defalias` *name lambda-list* **&body** *body* ⇒ *name* [Macro]

name A symbol.

lambda-list
A function lambda-list.

body List of forms to be executed, preceded by an optional list of declarations.

Example

```
(defalias float4 (&optional (float-type 'single-float))
  `(simple-array ,float-type (4)))
```

Implementor's note: *You must not define recursive types with* `defalias`*. Recursive types are supported only in structures.* `defalias`*, not unlike* `deftype`*, is unable to handle them - lisp system will crash or hang.*

4.1 User-level Translators

These functions and macros are used with `type specifiers`. Each of them parses a `typespec` into some internal representation and performs relevant operation.

Note that all of the functions mentioned here heavily utilize compiler macros, so you should pass a constant `typespec` argument to them wherever possible.

`sizeof` *typespec* **&optional** *value* ⇒ *count* [Function]

Returns number of bytes required to store value of type denoted by type specifier. This function also accepts optional argument - a lisp value which is used when the *typespec* denotes a variable-sized foreign type.

typespec A Virgil's type specifier.

value A lisp value.

count A number of bytes required to store value of type denoted by `typespec`.

`alignof` *typespec* ⇒ *alignment* [Function]

Computes alignment of type denoted by *typespec*.

typespec A Virgil's type specifier.

alignment A positive integer.

`offsetof` *typespec member* ⇒ *offset* [Function]

Computes offset(in bytes) of the specified structure's slot.

typespec A Virgil's type specifier denoting a structure type.

member A symbol, denoting a structure's slot.

offset A non-negative integer.

`convert` *lisp-value typespec* ⇒ *foreign-value* [Function]

Converts a lisp value into foreign value.

lisp-value A lisp value.

typespec A Virgil's type specifier.

foreign-value
A converted value.

typespec

`translate` *foreign-value typespec* ⇒ *lisp-value* [Function]

Translates a foreign value into lisp one.

foreign-value
A foreign value.

typespec A Virgil's type specifier.

lisp-value A translated value.

Implementor's note: `convert` *and* `translate` *only operate on immediate and primitive types*

`alloc` *typespec* **&optional** *value* ⇒ *pointer* [Function]

Allocates enough memory to hold the value of type denoted by *typespec*. This function accepts an optional parameter - a value, which is written to allocated memory. Unless this optional parameter is supplied, this function writes a type's `prototype` into allocated memory. (`prototype` concept is explained later in this manual).

typespec A Virgil's type specifier.

value A lisp value, which is written to freshly allocated memory.

pointer A pointer to foreign memory.

`clean` *pointer lisp-value typespec* [Function]

`Cleans` foreign memory, pointed by *pointer*, but does not deallocate it. The concept of `cleaning` a memory is roughly equivalent to the concept of C++ `destructors` - one of the tasks this function may perform is deallocatation of pointer slots in structures and arrays, for example.

pointer A pointer to foreign memory.

lisp-value A value which was written to that memory.

typespec A Virgil's type specifier.

This function does not return any useful values.

`free` *pointer* **&optional** *typespec* [Function]

Deallocates a block of foreign memory, allocated for the type denoted by `typespec`. Unless *typespec* is supplied, standard deallocator is used. This function does not `clean` a memory. For such a task, use `clean` or `clean-and-free` functions.

pointer A pointer to foreign memory.

typespec A Virgil's type specifier

This function does not return any useful values.

See also: Section 4.2 [Raw Memory Manipulation], page 10

`clean-and-free` *pointer value typespec* [Function]

Cleans a block of foreign memory and deallocates it afterwards.

pointer A pointer to foreign memory.

value A lisp value which was previously written to that memory.

typespec A Virgil's type specifier.

This function does not return any useful values.

`deref` *pointer typespec* **&optional** *offset output* ⇒ *lisp-value* [Accessor]

`deref` function dereferences a memory pointed by *pointer* and reads a lisp value of type denoted by *typespec*. When *output* parameter is present, and not equals to `NIL`, the result of the marshaling operation is stored into value denoted by this parameter, and this value becomes function's return value. In this case, *typespec* must denote some aggregate type.

`(setf deref)` function performs an inverse operation: it dereferences a foreign memory pointed by *pointer* and writes a `lisp-value` into it, accordingly to marshaling rules denoted by *typespec*.

> **Implementor's note:** `(setf deref)` *does not have output parameter, for obvious reasons.*

pointer A pointer to foreign memory.

typespec A Virgil's type specifier.

offset An integer, defaults to 0.

output A lisp value, defaults to NIL.

lisp-value A lisp value of type denoted by `typespec`.

`with-reference` (*var-or-vars value-var typespec* **&optional** *mode nullable*) **&body** *body* ⇒ *values* [Macro]

Executes *body* forms in dynamic environment where *var* is bound to the pointer to memory allocated for the value of type denoted by `typespec`.

This macro roughly emulates behavior of C++ references. Have a look at examples below.

Note that *var* pointer is invalid outside the macro scope, because the memory allocated to this pointer is freed after the dynamic environment exits. Moreover, this macro performs `clean` operation before exiting dynamic environment, so, for example, internal pointers in structures also become invalid outside the scope.

var-or-vars
::= *var* | (*var size-var*)

var A symbol naming a variable which is bound to the pointer to memory allocated for value denoted by *value-var*. Not evaluated.

size-var A symbol naming a variable which is bound to an integer that represents the size(in bytes) of the memory pointed by *var* pointer. Not evaluated.

value-var A symbol naming a variable which holds a value of type denoted by *typespec*. Not evaluated.

typespec A Virgil's type specifier. Evaluated.

mode A symbol, one of `:in`, `:out`, `:inout`. Defaults to `:in`. Not evaluated. This parameter represents a type of marshaling to be performed - `copy-in`, `copy-out`, or `copy-in-copy-out` respectively.

> **Implementor's note:** *on some lisp systems, with certain types of arrays copying is avoided. On such systems* `:virgil.shareable-arrays` *will be present in *features**

nullable Unless this parameter equals to `NIL`, value represented by *value-var* can be a special constant `VOID`, which denotes a NULL reference. In this case, *var* is bound to NULL pointer, and *size-var* is bound to 0. This parameter is not evaluated.

body A list of forms to be executed.

values Values returned by last form in *body*.

Implementor's note: *If typespec denotes a foreign type of fixed size(that is, all values of this type occupy the same number of bytes in memory), memory may be allocated on stack.*

`with-references` (**&rest** *specs*) **&body** *body* ⇒ *values* [Macro]

specs A list of parameter specifications. Each one corresponds to single `with-reference` parameter form.

body A list of forms to be executed.

values Values returned by last form in *body*.

`with-pointer` (*var-or-vars value typespec* **&optional** *mode nullable*) **&body** *body* ⇒ *values* [Macro]

Equivalent to `with-reference` but *value* parameter denotes a value itself, not a variable name, and is evaluated.

`with-pointers` (**&rest** *specs*) **&body** *body* ⇒ *values* [Macro]

Equivalent to `with-references` but each *value* parameter denotes a value itself, not a variable name, and is evaluated.

`with-value` (*var pointer typespec* **&optional** *mode nullable*) **&body** *body* ⇒ *values* [Macro]

An opposite operation, compared to `with-reference`. Executes *body* forms in dynamic environment where *var* is bound to value that corresponds to lisp representation of memory pointed by *pointer*.

var A symbol, naming a variable which is bound to lisp value. Not evaluated.

pointer A foreign pointer. Evaluated.

typespec A Virgil's type specifier. Evaluated.

mode A symbol, one of `:in`, `:out`, `:inout`. Not evaluated. Defaults to `:in`. `:in` correspond to `copy-in` operation - a data is readen from *pointer*(as with `deref`) and is bound to *var*. `:out` correspond to `copy-out` operaton - a lisp value which is bound to *var* during the execution of *body* forms is written into foreign memory pointed by *pointer*. `:inout` correspond to combination of this operations.

nullable Unless this parameter equals to NIL, *pointer* may be equal to NULL pointer, and *var* may be bound to special constant `VOID`, which represents NULL reference. Wherever one of these conditions occurs, no marshaling is performed. This parameter is not evaluated.

body A list of forms to be executed.

values Values returned by last form in *body*.

`with-values` (**&rest** *specs*) **&body** *body* ⇒ *values* [Macro]

specs A list of parameter specifications. Each one corresponds to single `with-value` parameter form.

body	A list of forms to be executed.
values	Values returned by last form in *body*.

Examples

```
(sizeof 'uint32)
⇒ 4

(alignof 'byte)
⇒ 1

(offsetof '(struct ()
             (x float)
             (y float)
             (z float))
          'z)
⇒ 8

(convert #\A 'char)
⇒ 65

(translate 1 'boolean)
⇒ T

(let* ((list '("Hello, " "world!"))
       (pointer (alloc '(sequence (& string)) list)))
  (unwind-protect
      (concatenate 'string
                   (deref pointer '(& string))
                   (deref pointer '(& string) (sizeof 'pointer)))
    (clean-and-free pointer list '(sequence (& string)))))
⇒ "Hello, world!"

(let ((x 1))
  (with-reference (p x 'int :inout)
    (with-value (val p 'int :inout)
      (incf val)))
  x)
⇒ 2
```

4.2 Raw Memory Manipulation

Sometimes it is neccessary to directly manipulate foreign memory. This section describes functions and macros that are used to allocate and free uninitialized memory, as well as ones that are used in pointer arithmetic.

`&` *address* ⇒ *pointer* [Function]

Constructs a foreign pointer from *address*.

address A non-negative integer.

pointer A foreign pointer.

`&&` *pointer* ⇒ *address* [Function]

Returns an integer representation of a *pointer*.

pointer A foreign pointer.

address A non-negative integer.

`&p` *object* ⇒ *T or NIL* [Function]

A predicate for a pointer. Returns `T` if an *object* is a foreign pointer and `NIL` otherwise.

`&?` *pointer* ⇒ *T or NIL* [Function]

A predicate for a non-NULL pointer. Returns `NIL` if a *pointer* is a NULL pointer and `T` otherwise.

`&=` *pointer1 pointer2* ⇒ *T or NIL* [Function]

Pointer comparator. Returns `T` if *pointer1* points to the same location in memory as *pointer2* and `NIL` otherwise.

`&0` ⇒ *a NULL pointer* [Function]

Returns a NULL pointer. The symbol `&0` also denotes a corresponding symbol macro.

`&+` *pointer offset* **&optional** *typespec* ⇒ *new-pointer* [Function]

Increments a *pointer* by an *offset*. If *typespec* parameter is supplied, offset is measured in sizes of type denoted by it, otherwise offset is measured in bytes.

`&-` *pointer offset* **&optional** *typespec* ⇒ *new-pointer* [Function]

Decrements a *pointer* by an *offset*. If *typespec* parameter is supplied, offset is measured in sizes of type denoted by it, otherwise offset is measured in bytes.

`raw-alloc` *size* ⇒ *pointer* [Function]

Allocates *size* bytes of foreign memory.

size A non-negative integer,

pointer A foreign pointer.

`raw-free` *pointer* [Function]

Deallocates a block of foreign memory that was previously allocated by `raw-alloc`.

pointer A foreign pointer

This function does not return any useful values.

`with-raw-pointer` (*var size* **&optional** *size-var*) **&body** *body* ⇒ *values* [Macro]

Executes *body* forms in dynamic environment where *var* is bound to a pointer to a block of foreign memory of *size* bytes.

Note that *var* pointer is invalid outside the macro scope, because the memory allocated to this pointer is freed after the dynamic environment exits.

var A symbol denoting a variable name that is bound to a pointer. Not evaluated.

size A non-negative integer. Evaluated.

size-var An optional parameter that denotes a variable name that is bound to the result of evaluation of *size* parameter. Not evaluated.

body A list of forms to be executed.

values Values returned by last form in *body*.

Implementor's note: *If size parameter is a constant expression, memory may be allocated on stack.*

`with-raw-pointers` (**&rest** *specs*) **&body** *body* ⇒ *values* [Macro]

specs A list of parameter specifications. Each one corresponds to single `with-raw-pointer` parameter form.

body A list of forms to be executed.

values Values returned by last form in *body*.

Examples

```
(&& &0)
⇒ 0

(&= &0 (&- (& 1) 1))
⇒ T

(with-raw-pointer (p 100 size)
  (when (/= 0 (external-function-call
                #+windows "_snprintf"
                #-windows "snprintf"
                ((:cdecl) (int)
                 (pointer buffer)
                 (size-t size)
                 ((& string) format)
                 (int x)
                 (int y)
                 (int z))
                p size "%d+%d=%d" 1 2 (+ 1 2)))
    (deref p 'string)))
⇒ "1+2=3"
```

4.3 Handling Circular References

Vigil is able to automatically marshal circular structures that occur either in lisp or in foreign memory. However, due to the fact that the process of tracing circular references has a significant performance impact, it is disabled by default.

You can control the mentioned process by the means of one of the following functions and macros:

`enable-circular-references` [Function]
: Enables the process of circular reference tracing in the current dynamic environment (either global or established by the means of `with-circular-references` or `without-curcular-references`)

This function does not return any useful values.

`disable-circular-references` [Function]
: Disables the process of circular reference tracing in the current dynamic environment (either global or established by the means of `with-circular-references` or `without-curcular-references`)

This function does not return any useful values.

`clear-circular-reference-cache` [Function]
: Clears the internal cache of circular references in the current dynamic environment (either global or established by the means of `with-circular-references` or `without-curcular-references`)

This function does not return any useful values.

`with-circular-references` **&body** *body* ⇒ *values* [Macro]
: Executes *body* forms in the dynamic environment where the process of circular reference tracing is enabled.

body A list of forms to be executed.

values Values returned by last form in *body*.

`without-circular-references` **&body** *body* ⇒ *values* [Macro]
: Executes *body* forms in the dynamic environment where the process of circular reference tracing is disabled.

body A list of forms to be executed.

values Values returned by last form in *body*.

Example

```
(define-struct node
  (data int)
  (next (& node :in t)))

;;  Reference types('&') will be explained later
;;  in this manual. Third parameter of this typespec
;;  designates whether a reference is nullable or not.
```

```
(with-circular-references
  (let* ((circle (make-node)))
    (setf (node-next circle) circle)
    (with-pointer (p circle 'node)
      (let ((node (deref p 'node)))
        (eq node (node-next node))))))
⇒ T
```

5 Translators and Translatable Types

TODO

5.1 Translators

TODO

5.2 Defining and Parsing Type Specifiers

TODO

5.3 Primitive Types

TODO

5.4 Immediate Types

TODO

5.5 Aggregate Types

TODO

5.6 Proxy Types

TODO

6 Built-in Types

TODO

6.1 Built-in Primitives

TODO

6.2 References

TODO

6.3 Arrays and Sequences

TODO

6.4 Enumerations

TODO

6.5 Structures and Unions

TODO

6.6 Strings

TODO

6.7 Strictly Aligned Types

TODO

6.8 Filtered Types

TODO

6.9 Const Types

TODO

7 Functions

TODO

8 Symbols Re-Exported from CFFI

TODO

Appendix A Glossary

TODO

Index

(Index is nonexistent)