Clostrum

First-class global environments for Common Lisp

Robert Strandh

2020

Contents

Chapter 1

Introduction

Clostrum is an extended version of the concept of first-class global environments [Str15]. The extension consists of new classes for the evaluation environment and the compilation environment, and of new generic functions applicable to instances of those new classes.

Chapter 2

General principles

Clostrum is designed to provide most of the functionality that typical clients will need. But Clostrum is also designed to be extended by clients that need additional functionality.

The main mechanism for providing extensions is the use of a `client` parameter in all of the generic functions defined by Clostrum. None of the methods of the generic functions defined by Clostrum specialize to the `client` parameter. However, it is highly recommended that client code respect these two conventions:

1. Clients calling a Clostrum generic function should always supply an instance of some client-specific standard object as the `client` argument.
2. Client-specific methods on Clostrum generic functions should always specialize to some client-specific class.

These rules exist so that several different clients may coexist in the same Common Lisp image, without interfering with one another.

Chapter 3

Run-time environment

3.1 Class

⇒ `run-time-environment` [*Class*]

This is the base class for run-time environments.

3.2 Protocol functions

⇒ `fboundp` *client environment function-name* [*Generic Function*]

This generic function is a generic version of the Common Lisp function `fboundp`.

It returns true if *function-name* has a definition in *environment* as an ordinary function, a generic function, a macro, or a special operator.

⇒ `fmakunbound` *client environment function-name* [*Generic Function*]

This generic function is a generic version of the Common Lisp function named `fmakunbound`.

This function makes *function-name unbound* in the function namespace of *environment*.

If *function-name* already has a definition in *environment* as an ordinary function, as a generic function, as a macro, or as a special operator, then that definition is lost.

If *function-name* has a `setf` expander associated with it, then that `setf` expander is lost.

⇒ `special-operator` *client environment function-name* [*Generic Function*]

If *function-name* has a definition as a special operator in *environment*, then that definition is returned. The definition is the object that was used as an argument to `(setf special-operator)`. The exact nature of this object is not specified, other than that it can not be `nil`. If *function-name* does not have a definition as a special operator in *environment*, then `nil` is returned.

⇒ `(setf special-operator)` *new client environment function-name* [*Generic Function*]

Set the definition of *function-name* to be a special operator. The exact nature of *new* is not specified, except that a value of `nil` means that *function-name* no longer has a definition as a special operator in *environment*.

If a value other than `nil` is given for *new*, and *function-name* already has a definition as an ordinary function, as a generic function, or as a macro, then an error is signaled. As a consequence, if it is desirable for *function-name* to have a definition both as a special operator and as a macro, then the definition as a special operator should be set first.

⇒ `fdefinition` *client environment function-name* [*Generic Function*]

This generic function is a generic version of the Common Lisp function named `cl:fdefinition`.

If *function-name* has a definition in the function namespace of *environment* (i.e., if `fboundp` returns true), then a call to this function succeeds. Otherwise an error of type `undefined-function` is signaled.

If *function-name* is defined as an ordinary function or a generic function, then a call to this function returns the associated function object.

If *function-name* is defined as a macro, then a list of the form `(cl:macro-function` *function*`)` is returned, where *function* is the macro expansion function associ-

ated with the macro.

If *function-name* is defined as a special operator, then a list of the form `(cl:special` *object*`)` is returned, where the nature of *object* is currently not specified.

⇒ `(setf fdefinition)` *new-def client environment function-name* [*Generic Function*]

This generic function is a generic version of the Common Lisp function named `(setf cl:fdefinition)`.

new-def must be an ordinary function or a generic function. If *function-name* already names a function or a macro, then the previous definition is lost. If *function-name* already names a special operator, then an error is signaled.

If *function-name* is a symbol and it has an associated `setf` expander, then that `setf` expander is preserved.

⇒ `macro-function` *client environment symbol* [*Generic Function*]

This generic function is a generic version of the Common Lisp function named `cl:macro-function`.

If *symbol* has a definition as a macro in *environment*, then the corresponding macro expansion function is returned.

If *symbol* has no definition in the function namespace of *environment*, or if the definition is not a macro, then this function returns `nil`.

⇒ `(setf macro-function)` *new-def client environment symbol* [*Generic Function*]

This generic function is a generic version of the Common Lisp function `(setf cl:macro-function)`.

new-def must be a macro expansion function or `nil`. A call to this function then always succeeds. A value of `nil` means that the *symbol* no longer has a macro function associated with it. If *symbol* already names a macro or a function, then the previous definition is lost. If *symbol* already names a special operator, that definition is kept.

If *symbol* already names a function, then any proclamation of the type of that function is lost. In other words, if at some later point *symbol* is again defined as a function, its proclaimed type will be `t`.

If *symbol* already names a function, then any `inline` or `notinline` proclamation of the type of that function is lost. In other words, if at some later point *symbol* is again defined as a function, its proclaimed inline information will be `nil`.

If *symbol* has an associated `setf` expander, then that `setf` expander is preserved.

⇒ `compiler-macro-function` *client environment function-name* [*Generic Function*]

This generic function is a generic version of the Common Lisp function named `cl:compiler-macro-function`.

If *function-name* has a definition as a compiler macro in *environment*, then the corresponding compiler macro function is returned.

If *function-name* has no definition as a compiler macro in *environment*, then this function returns `nil`.

⇒ `(setf compiler-macro-function)`
new-def client environment function-name [*Generic Function*]

This generic function is a generic version of the Common Lisp function `(setf cl:compiler-macro-function)`.

new-def can be a compiler macro function or `nil`. When it is a compiler macro function, then it establishes *new-def* as a compiler macro for *function-name* and any existing definition is lost. A value of `nil` means that *function-name* no longer has a compiler macro associated with it in *environment*.

⇒ `function-type` *client environment function-name* [*Generic Function*]

This generic function returns the proclaimed type of the function associated with *function-name* in *environment*.

If *function-name* is not associated with an ordinary function or a generic function in *environment*, then `nil` is returned.

If *function-name* is associated with an ordinary function or a generic function in *environment*, but no type proclamation for that function has been made, then this generic function returns `t`.

⇒ `(setf function-type)` *new-type client environment function-name* [*Generic Function*]

This generic function is used to set the proclaimed type of the function associated with *function-name* in *environment* to *new-type*.

If *function-name* is associated with a macro or a special operator in *environment*, then an error is signaled.

⇒ `function-inline` *client environment function-name* [*Generic Function*]

This generic function returns the proclaimed inline information of the function associated with *function-name* in *environment*.

If *function-name* is not associated with an ordinary function or a generic function in *environment*, then `nil` is returned.

If *function-name* is associated with an ordinary function or a generic function in *environment*, then the return value of this function is either `nil`, `inline`, or `notinline`. If no inline proclamation has been made, then this generic function returns `nil`.

⇒ `(setf function-inline)`
new-inline client environment function-name [*Generic Function*]

This generic function is used to set the proclaimed inline information of the function associated with *function-name* in *environment* to *new-inline*.

new-inline must have one of the values `nil`, `inline`, or `notinline`.

If *function-name* is not associated with an ordinary function or a generic function in *environment*, then an error is signaled.

⇒ `function-cell` *client environment function-name* [*Generic Function*]

A call to this function always succeeds. It returns a `cons` cell, in which the `car` always holds the current definition of the function named *function-name*. When *function-name* has no definition as a function, the `car` of this cell will contain a function that, when called, signals an error of type `undefined-function`. The

return value of this function is always the same (in the sense of `eq`) when it is passed the same (in the sense of `equal`) function name and the same (in the sense of `eq`) environment.

⇒ `function-unbound` *client environment function-name* [*Generic Function*]

A call to this function always succeeds. It returns a function that, when called, signals an error of type `undefined-function`. When *function-name* has no definition as a function, the return value of this function is the contents of the `cons` cell returned by `function-cell`. The return value of this function is always the same (in the sense of `eq`) when it is passed the same (in the sense of `equal`) function name and the same (in the sense of `eq`) environment. Client code can use the return value of this function to determine whether *function-name* is unbound and if so signal an error when an attempt is made to evaluate the form `(function` *function-name*`)`.

⇒ `function-lambda-list` *client environment function-name* [*Generic Function*]

This function returns two values. The first value is an ordinary lambda list, or `nil` if no lambda list has been defined for *function-name*. The second value is true if and only if a lambda list has been defined for *function-name*.

⇒ `(setf function-lambda-list)`
new-lambda-list client environment function-name [*Generic Function*]

This generic function is used to associate a new lambda list with a function name.

new-lambda-list is a new lambda list for the function named *function-name*

⇒ `boundp` *client environment symbol* [*Generic Function*]

It returns true if *symbol* has a definition in *environment* as a constant variable, as a special variable, or as a symbol macro. Otherwise, it returns `nil`.

⇒ `constant-variable` *client environment symbol* [*Generic Function*]

This function returns the value of the constant variable *symbol*.

If *symbol* does not have a definition as a constant variable, then an error is signaled.

⇒ `(setf constant-variable)` *value client environment symbol* [*Generic Function*]

This function is used in order to define *symbol* as a constant variable in *environment*, with *value* as its value.

If *symbol* already has a definition as a special variable or as a symbol macro in *environment*, then an error is signaled.

If *symbol* already has a definition as a constant variable, and its current value is not `eql` to *value*, then an error is signaled.

⇒ `special-variable` *client environment symbol* [*Generic Function*]

This function returns two values. The first value is the value of *symbol* as a special variable in *environment*, or `nil` if *symbol* does not have a value as a special variable in *environment*. The second value is true if *symbol* does have a value as a special variable in *environment* and `nil` otherwise.

Notice that the symbol can have a value even though this function returns `nil` and `nil`. The first such case is when the symbol has a value as a constant variable in *environment*. The second case is when the symbol was assigned a value using `(setf symbol-value)` without declaring the variable as `special`.

⇒ `(setf special-variable)` *value symbol environment init-p* [*Generic Function*]

This function is used in order to define *symbol* as a special variable in *environment*.

If *symbol* already has a definition as a constant variable or as a symbol macro in *environment*, then an error is signaled. Otherwise, *symbol* is defined as a special variable in *environment*.

If *symbol* already has a definition as a special variable, and *init-p* is `nil`, then this function has no effect. The current value is not altered, or if *symbol* is currently unbound, then it remains unbound.

If *init-p* is true, then *value* becomes the new value of the special variable *symbol*.

⇒ `symbol-macro` *client environment symbol* [*Generic Function*]

This function returns two values. The first value is a macro expansion function

associated with the symbol macro named by *symbol*, or `nil` if *symbol* does not have a definition as a symbol macro. The second value is the form that *symbol* expands to as a macro, or `nil` if symbol does not have a definition as a symbol macro.

It is guaranteed that the same (in the sense of `eq`) function is returned by two consecutive calls to this function with the same (in the sense of `eq`) symbol as the first argument, as long as the definition of *symbol* does not change.

⇒ `(setf symbol-macro)` *expansion client environment symbol* [*Generic Function*]

This function is used in order to define *symbol* as a symbol macro with the given *expansion* in *environment*.

If *symbol* already has a definition as a constant variable, or as a special variable, then an error of type `program-error` is signaled.

⇒ `variable-type` *client environment symbol* [*Generic Function*]

This generic function returns the proclaimed type of the variable associated with *symbol* in *environment*.

If *symbol* has a definition as a constant variable in *environment*, then the result of calling `type-of` on its value is returned.

If *symbol* does not have a definition as a constant variable in *environment*, and no previous type proclamation has been made for *symbol* in *environment*, then this function returns `t`.

⇒ `(setf variable-type)` *new-type client environment symbol* [*Generic Function*]

This generic function is used to set the proclaimed type of the variable associated with *symbol* in *environment*.

If *symbol* has a definition as a constant variable in *environment*, then an error is signaled.

It is meaningful to set the proclaimed type even if *symbol* has not previously been defined as a special variable or as a symbol macro, because it is meaningful to use `(setf symbol-value)` on such a symbol.

Recall that the HyperSpec defines the meaning of proclaiming the type of a symbol macro. Therefore, it is meaningful to call this function when *symbol* has a definition as a symbol macro in *environment*.

⇒ `variable-cell` *client environment symbol* [*Generic Function*]

A call to this function always succeeds. It returns a `cons` cell, in which the `car` always holds the current definition of the variable named *symbol*. When *symbol* has no definition as a variable, the `car` of this cell will contain an object that indicates that the variable is unbound. This object is the return value of the function `variable-unbound`. The return value of this function is always the same (in the sense of `eq`) when it is passed the same symbol and the same environment.

⇒ `variable-unbound` *client environment symbol* [*Generic Function*]

A call to this function always succeeds. It returns an object that indicates that the variable is unbound. The `cons` cell returned by the function `variable-cell` contains this object whenever the variable named *symbol* is unbound. The return value of this function is always the same (in the sense of `eq`) when it is passed the same symbol and the same environment (in the sense of `eq`). Client code can use the return value of this function to determine whether *symbol* is unbound.

⇒ `find-class` *client environment symbol* [*Generic Function*]

This generic function is a generic version of the Common Lisp function `cl:find-class`.

If *symbol* has a definition as a class in *environment*, then that class metaobject is returned. Otherwise `nil` is returned.

⇒ `(setf find-class)` *new-class client environment symbol* [*Generic Function*]

This generic function is a generic version of the Common Lisp function `(setf cl:find-class)`.

This function is used in order to associate a class with a class name in *environment*.

If *new-class* is a class metaobject, then that class metaobject is associated with the name *symbol* in *environment*. If *symbol* already names a class in

environment than that association is lost.

If *new-class* is `nil`, then *symbol* is no longer associated with a class in *environment*.

If *new-class* is neither a class metaobject nor `nil`, then an error of type `type-error` is signaled.

⇒ `setf-expander` *client environment symbol* [*Generic Function*]

This generic function returns the `setf` expander associated with *symbol* in *environment*. If *symbol* is not associated with any `setf` expander in *environment*, then `nil` is returned.

⇒ `(setf setf-expander)` *new-expander client environment symbol* [*Generic Function*]

This generic function is used to set the `setf` expander associated with *symbol* in *environment*.

If *symbol* is not associated with an ordinary function, a generic function, or a macro in *environment*, then an error is signaled.

If there is already a `setf` expander associated with *symbol* in *environment*, then the old `setf` expander is lost.

If a value of `nil` is given for *new-expander*, then any current `setf` expander associated with *symbol* is removed. In this case, no error is signaled, even if *symbol* is not associated with any ordinary function, generic function, or macro in *environment*.

⇒ `default-setf-expander` *environment* [*Generic Function*]

This generic function returns the default `setf` expander, to be used when the function `setf-expander` returns `nil`. This function always returns a valid `setf` expander.

⇒ `(setf default-setf-expander)` *new-expander client environment* [*Generic Function*]

This generic function is used to set the default `setf` expander in *environment*.

⇒ `type-expander` *client environment symbol* [*Generic Function*]

This generic function returns the type expander associated with *symbol* in *environment*. If *symbol* is not associated with any type expander in *environment*, then `nil` is returned.

⇒ `(setf type-expander)` *new-expander client environment symbol* [*Generic Function*]

This generic function is used to set the type expander associated with *symbol* in *environment*.

If there is already a type expander associated with *symbol* in *environment*, then the old type expander is lost.

⇒ `find-package` *client environment name* [*Generic Function*]

Return the package with the name or the nickname *name* in the environment *environment*. If there is no package with that name in *environment*, then return `nil`. Contrary to the standard Common Lisp function `cl:find-package`, for this function, *name* must be a string.

⇒ `(setf find-package)` *new-package client environment name* [*Generic Function*]

This function is used in order to associate a package with a package name in *environment*. The argument *name* must be a string.

If *new-package* is a package object, then that package object is associated with the name *name* in *environment*. If *name* already names a package in *environment* than that association is lost.

If *new-package* is `nil`, then *name* is no longer associated with a package in *environment*.

If *new-package* is neither a package object nor `nil`, then an error of type `type-error` is signaled.

Chapter 4

Evaluation environment

4.1 Mixin class

⇒ `evaluation-environment-mixin` [*Class*]

This class should be used in order to create an evaluation environment class that has this mixin class and a run-time environment class as superclasses.

⇒ `:parent` [*Initarg*]

This initialization argument must be provided and should be either another evaluation environment, or a run-time-environment.

⇒ `parent` *environment* [*Generic Function*]

⇒ `parent` *(environment* `evaluation-environment-mixin`*)* [*Method*]

This generic function returns the environment objection that was passed as the `:parent` initialization argument when *environment* was created.

4.2 Methods

⇒ `fboundp` *client (environment* `evaluation-environment-mixin`*)*
function-name [*Method*]

This method first calls `(call-next-method)`. If that call returns true, then that true value is returned. If that call returns false, then this method calls `fboundp` with the same `client` argument, a second argument resulting from calling `(parent environment)` and the same `function-name` argument.

⇒ `special-operator` *client (environment* `evaluation-environment-mixin`*)* *function-name* [*Method*]

This method first calls `(call-next-method)`. If that call returns true, then that true value is returned. If that call returns false, then this method calls `special-operator` with the same `client` argument, a second argument resulting from calling `(parent environment)` and the same `function-name` argument.

⇒ `fdefinition` *client (environment* `evaluation-environment-mixin`*)* *function-name* [*Method*]

This method first calls `(call-next-method)`. If that call returns true, then that true value is returned. If that call returns false, then this method calls `fdefinition` with the same `client` argument, a second argument resulting from calling `(parent environment)` and the same `function-name` argument.

⇒ `macro-function` *client (environment* `evaluation-environment-mixin`*)* *symbol* [*Method*]

This method first calls `(call-next-method)`. If that call returns true, then that true value is returned. If that call returns false, then this method calls `macro-function` with the same `client` argument, a second argument resulting from calling `(parent environment)` and the same `symbol` argument.

⇒ `compiler-macro-function` *client (environment* `evaluation-environment-mixin`*)* *function-name* [*Method*]

This method first calls `(call-next-method)`. If that call returns true, then that true value is returned. If that call returns false, then this method calls `compiler-macro-function` with the same `client` argument, a second argument resulting from calling `(parent environment)` and the same `function-name` argument.

⇒ `function-type` *client (environment* `evaluation-environment-mixin`*)* *function-name* [*Method*]

This method first calls `(call-next-method)`. If that call returns true, then

that true value is returned. If that call returns false, then this method calls `function-type` with the same `client` argument, a second argument resulting from calling `(parent environment)` and the same `function-name` argument.

MORE METHODS HERE...

Chapter 5

Compilation environment

5.1 Class

⇒ `compilation-environment` [*Class*]

An instance of this class is passed as the `&environment` argument to macro functions during compile time.

⇒ `:parent` [*Initarg*]

This initialization argument must be provided and should be an evaluation environment.

⇒ `parent` *environment* [*Generic Function*]

⇒ `parent` *(environment* `compilation-environment`*)* [*Method*]

This generic function returns the environment objection that was passed as the `:parent` initialization argument when *environment* was created.

5.2 Generic functions

⇒ `function-lambda-list` *client environment function-name* [*Generic Function*]

This generic function returns two values. The first value is the lambda list of the function named *function-name*. The second value is true if and only if information about the lambda list of the function named *function-name* exists. In other words, if no information about a the lambda list of a function with that name exists in *environment*, then the values `nil` and `nil` are returned.

This function is used in a Trucler method in order to return information about the name of a function that is being encountered in source code.

⇒ `(setf function-lambda-list)` *lambda-list client environment function-name* [*Generic Function*]

This generic function sets the lambda list class of the function named *function-name*.

This generic function can be called by the expansion of the macros `defun` or `defgeneric`, in the `:compile-toplevel` situation of an `eval-when` form. Thus, even though the function can not be created at compile time, the compiler is informed about the lambda list when a reference to the function is encountered later during compilation.

⇒ `function-class-name` *client environment function-name* [*Generic Function*]

This generic function returns the name of a the class of the function named *function-name*. If no information about a function with that name exists in *environment*, then `nil` is returned.

This generic function is typically used in the expansion of the `defmethod` macro, in the `:compile-toplevel` situation of an `eval-when` form. Recall that the MOP function `make-method-lambda` is called at compile time by the expansion of this macro. In order to pass a class prototype for the `generic-function` argument to `make-method-lambda`, this generic function can be used to determine which class prototype to use.

⇒ `(setf function-class-name)` *class-name client environment function-name* [*Generic Function*]

This generic function sets the name of a the class of the function named *function-name*.

Information about the class of a function is useful mostly for generic functions,

but can also be used for ordinary functions and instances of the class named `funcallable-standard-object`, defined by the metaobject protocol.

⇒ `method-class-name` *client environment*
function-name [*Generic Function*]

This generic function returns the name of a the class of the methods on the generic function named *function-name*. If no information about a generic function with that name exists in *environment*, then `nil` is returned.

Like the generic function `function-class-name` this generic function is used in the expansion of the `defmethod` macro, and for the same reason. The generic function `make-method-lambda` must be given the class prototype of a method class, and that method class is determined using this generic function.

⇒ `(setf method-class-name)` *class-name client environment*
function-name [*Generic Function*]

This generic function sets the name of a the class of the methods on the generic function named *function-name*.

MORE GENERIC FUNCTIONS HERE

Chapter 6

Using Trucler with Clostrum

6.1 ASDF system definition and package

Trucler can optionally be used with Clostrum. The ASDF system `clostrum-trucler` should be used for this purpose.

The code for the method definitions described in this chapter is in a separate package named `clostrum-trucler`.

6.2 Trucler methods

⇒ `describe-variable` *client (environment compilation-environment) name* [*Method*]

This method makes an instance of an appropriate Trucler class as follows:

- If *name* is the name of a special variable in *environment*, then an instance of the Trucler class `global-special-variable-description` is made.
- If *name* is the name of a constant variable in *environment*, then an instance of the Trucler class `constant-variable-description` is made.

- If *name* is the name of a symbol macro in *environment*, then an instance of the Trucler class `global-symbol-macro-description` is made.

⇒ `describe-function` *client (environment compilation-environment) name* [*Method*]

This method makes an instance of an appropriate Trucler class as follows:

- If *name* is the name of an ordinary function in *environment*, then an instance of the Trucler class `global-function-description` is made.
- If *name* is the name of a generic function in *environment*, then an instance of the Trucler class `generic-function-description` is made.
- If *name* is the name of a macro in *environment*, then an instance of the Trucler class `global-macro-description` is made.

⇒ `describe-block` *client (environment compilation-environment) name* [*Method*]

This method returns `nil`.

⇒ `describe-tag` *client (environment compilation-environment) name* [*Method*]

This method returns `nil`.

⇒ `global-environment` *client (environment compilation-environment)* [*Method*]

This method returns *environment*.

⇒ `macro-function` *client (environment compilation-environment) symbol* [*Method*]

⇒ `compiler-macro-function` *client (environment compilation-environment) function-name* [*Method*]

⇒ `symbol-macro-expansion` *client (environment compilation-environment) symbol* [*Method*]

Chapter 7

Using Clostrum

There are several use cases for Clostrum depending on the type and level of ambition of the client.

7.1 Compilation environment only

An existing (perhaps relatively new) Common Lisp implementation that has decided to take advantage of the permission to merge the run-time environment and the evaluation environment, but that wants a separate compilation environment can use Clostrum as follows:

- It would define an environment class (say `client:global-environment`) that has `clostrum:run-time-environment` as a superclass.
- It would implement methods on the generic functions in Section 3.2 that trampoline to existing implementation-specific functions in the single global environment of the client.
- It would create a constellation consisting of an instance of `client:global-environment` as the parent of an instance of `clostrum:compilation-environment`

7.2 Compilation and evaluation environment

An existing Common Lisp implementation with a traditional, single global run-time environment that wants a separate evaluation environment in order to avoid compile-time side effects to its run-time environment can use Clostrum as follows:

- It would define an environment class (say `client:run-time-environment`) that has `clostrum:environment` as a superclass.
- It would implement methods on the generic functions in Section 3.2 that trampoline to existing implementation-specific functions in the single global environment of the client.
- It would create a class (say `client:evaluation-environment`) as a subclass of `clostrum:evaluation-environment-mixin` and `client:run-time-environment`.
- It would create a constellation consisting of an instance of `client:run-time-environment`, an instance of `client:evaluation-environment`, `clostrum:compilation-environment`

Chapter 8

Extending Clostrum

Bibliography

[Str15] Robert Strandh. First-class global environments in common lisp. In *Proceedings of the 8th European Lisp Symposium*, ELS '15, pages 79 – 86, April 2015.

Index