The OOPS Package for Elk Scheme

Oliver Laumann

1. Introduction

The OOPS package provides a minimal set of tools that enables a Scheme programmer to program in an object oriented style. The functionality of OOPS is similar to that of packages like CLOS and SCOOPS, although the current version does not support multiple inheritance. The rest of this memo serves as a reference guide to the OOPS package; the reader is assumed to be familiar with the terminology of object oriented programming.

2. Using OOPS

Programs that make use of the OOPS package should include the line

```
(require 'oops)
```

Since this involves autoloading of an object file, it may be desirable to dump Scheme after the OOPS package has been loaded.

3. Defining Classes

New classes are defined by means of the `define-class` macro. The syntax of `define-class` is

(`define-class` *class-name* `.` *options*)

where *class-name* is a symbol. *options* can be of the form

(`super-class` *class-name*)

where *class-name* is the name of the super-class (a symbol), or

(`class-vars` `.` *var-specs*)

or

(`instance-vars` `.` *var-specs*)

to specify the class variables and instance variables of the newly defined class. Each *var-spec* is either a symbol (the name of the variable) or of the form

(*symbol* *initializer*) .

Variables for which no initializer has been specified are initialized to the empty list. The initializers for class variables are evaluated immediately; initializers for instance variables are evaluated each time an instance of the newly defined class is created. Evaluation of initializers is performed in a way that the initializer of a variable can reference all variables appearing at the left of the variable being initialized; for instance

```
(define-class foo (class-vars (a 10) (b (* a 2))))
```

would initialize the class variable `b` to 20.

A class inherits all class variables, instance variables, and methods of its super-class. When a class and its super-class each have an instance variable with the same name, the corresponding *var-specs* must either both have no initializer or initializers with the same value, otherwise an "initializer mismatch" error is signaled by `define-class`.

Each instance of a class has an instance variable named `self`. The value of `self` is the instance with respect to which `self` is evaluated. `self` can be used by methods as the argument to `send` (see below) to invoke another method within the current instance.

`define-class` does not have a meaningful return value, instead it has a side-effect on the environment in which it is invoked.

4. Creating Instances of a Class

The macro `make-instance` is used to create an instance of a class; it returns the instance as its value. The syntax is

(`make-instance` *class* . *args*)

where *class* is the class of which an instance is to be created. Each *arg* of the form

(*symbol initializer*)

where *symbol* is the name of an instance variable of the class, is used to initialize the specified instance variable in the newly created instance. In this case the *initializer* supersedes any initializer specified in the call to `define-class`. Thus it is possible to have instance variables with a *default initializer* that can be overridden for individual instances. The initializers are evaluated in the current environment.

`make-instance` initializes the newly created instance by invoking the `initialize-instance` method for the class and all super-classes in super-class to sub-class order. That is, the `initialize-instance` method of the class specified in the call to `make-instance` is called after all other `initialize-instance` methods. The arguments passed to the `initialize-instance` method of a class are those arguments of the call to `make-instance` that do not represent an initialization form for an instance variable. These arguments are evaluated in the current environment. It is not required for a class to have an `initialize-instance` method.

Consider the following example:

```
(require 'oops)

(define-class c (instance-vars a))
(define-class d (instance-vars (b 10)) (super-class c))

(define-method c (initialize-instance . args)
  (print (cons 'c args)))

(define-method d (initialize-instance . args)
  (print (cons 'd args)))
```

In this example evaluation of

```
(define x 99)
(define i (make-instance d (a 20) 'foo (b x) x))
```

would print

```
(c foo 99)
(d foo 99)
```

Note that first the `initialize-instance` method of `c` is invoked and then that of the class `d`. The instance variables `a` and `b` would be initialized to 20 and 99, respectively.

5. Defining Methods

A new method can be defined by means of the `define-method` macro. The syntax is

(`define-method` *class lambda-list* `.` *body*)

where *class* is the class to which the method is to be added, *lambda-list* is a list specifying the method's name and formal arguments (having the same syntax as the argument of `define`).

`define-method` simply creates a class-variable with the name of the method, creates a lambda closure using the *lambda-list* and the *body* forms, and binds the resulting procedure to the newly-created variable. When a message with the name of the method is sent to an instance of the class, the method is invoked, and the *body* is evaluated in the scope of the instance (so that it can access all instance and class variables).

6. Sending Messages

A message can be sent to an instance by means of the function `send`. The syntax of `send` is

(`send` *instance message* `.` *args*)

where *instance* is the instance to which the message is to be sent, *message* is the name of the method to be invoked (a symbol), and *args* are the arguments to be passed to the method. Example:

```
(define-class c (instance-vars a) (class-vars (b 10)))

(define-method c (foo x)
  (cons (set! a x) b))     ; set! returns previous value

(define i (make-instance c (a 99)))


(send i 'foo 1)            returns (99 . 10)
(send i 'foo 2)            returns (1 . 10)
```

When a message is sent to an instance for which no method has been defined, a "message not understood" error is signaled.

The function `send-if-handles` is identical to `send`, except that it returns a list of one element, the return value of the method, or `#f` when the message is not understood by the instance.

7. Evaluating Expressions within an Instance

The macro `with-instance` can be used to evaluate expressions within the scope of an instance. The syntax is

(`with-instance` *instance* `.` *body*)`.`

The *body* forms are evaluated in the same environment in which a method of *instance* would be evaluated, i. e. they can access all and class and instance variables (including `self`). `with-instance` returns the value of the last *body* form. Example:

```
(define-class c (class-vars (x 5)) (instance-vars y))

(define i (make-instance c (y 1)))

(define x 10)
(with-instance i (cons x y))          returns (5 . 1)
```

8. Setting Instance and Class Variables

Generally class and instance variables are manipulated by methods or, if applicable, from within a `with-instance` form. In addition, values can be assigned to class and instance variables without involving a message send by means of the `instance-set!` macro. The syntax of `instance-set!` is

(`instance-set!` *instance variable value*)

where *variable* is a symbol, the name of the class or instance variable. `instance-set!` returns the previous value of the variable (like `set!`).

Class variables can be modified without involving an instance of the class by means of the macro `class-set!`:

(`class-set!` *class variable value*)`.`

variable must be the name of a class variable of *class*. Note that one difference between

```
(instance-set! i 'var x)
```

and

```
(with-instance i (set! var x))
```

is that in the former case `x` is evaluated in the current environment while in the latter case `x` is evaluated within the scope of the instance (here `x` might be a class or instance variable).

9. Obtaining Information about Classes and Instances

The function `class-name` returns the name of a class (a symbol) or, when applied to an instance, the name of the class of which it is an instance.

The predicate `method-known?` can be used to check whether a method of a given name is known to a class. The syntax is

(method-known? *method class*)

where *method* is a symbol.

The type predicates `class?` and `instance?` can be used to check whether an object is a class or an instance, respectively.

The functions

(check-class *symbol object*)

and

(check-instance *symbol object*)

check whether *object* is a class (i. e. satisfies the predicate `class?`) or an instance, respectively, and, if not, signal an error; in this case *symbol* is used as the first argument to `error`.

The functions `describe-class` and `describe-instance` print the components (name, class/instance variables, etc.) of a class or instance, respectively. The function `describe` has been extended in way that when `(feature? 'oops)` is true, `describe-class` or `describe-instance` are called when `describe` is applied to an object that satisfies `class?` or `instance?`, respectively.

Index

Table of Contents