A-4-2

Discriminating Eval

Masayuki Ida
(Aoyama Gakuin University)

The design of Discriminating EVAL (D-Eval) and its first implementation are described. D-Eval is a CommonLoops interpreter. The extension of the definitions of type-of and subtypep are proposed. This paper also tries to give a transparency of the basic mechanism for CommonLoops execution.

1.Introduction

Discriminating EVAL (D-Eval) is a multiple-value oriented extension of a EVAL to cope with CommonLoops. In other words, it is an interpreter centered Common-Loops implementation. This paper describes D-Eval to show a user transparent mechanism for CommonLoops execution, but is slow.

The main purposes of this experimental developement of D-Eval are:

1) to clarify the semantics base of CommonLoops
2) to get a simple CommonLoops for a base of the extensive design
3) to analize the Common Lisp / CommonLoops functionality

CommonLoops [Bob86] is an object oriented facility for Common Lisp and is the base for the comming standard of Common Lisp object oriented facility. Xerox Parc, which is the original designer of CommonLoops distributes Portable CommonLoops (PCL) among the Common Lisp community.

PCL is compiler centered and is no longer a simple pilot implementation for the novices. The primary concern on developing D-Eval at first is to make a milestone between Common Lisp and CommonLoops.

D-Eval is written in PCL, but can be rewritten in Common Lisp without PCL. This paper assumes the basic knowledge on CommonLoops [Bob86] and Common Lisp [Steele84].

2. D-Eval as a multiple value Eval

Fig. 1 shows the conversation with D-Eval. Two values are always returned after evalutaing the form. The first value is the value (first value) of the form, and the second value is the 'type' of the value. 'type' is explained in 4. Except for excluding VALUES-form from top level, D-Eval is quite simular to an usual Eval.

```
D-Eval>(+ 1 2)
3
; FIXNUM
D-Eval>'x
X
; SYMBOL
D-Eval>((lambda (x) (cons 'x x)) 1)
(X . 1)
; CONS
```

Fig. 1 D-Eval as a multiple value Eval

3. Internal representation of methods in D-Eval

Methods with the same selector make a method-list. Method-list is stored in the function cell of a selector, not stored in the global table. Method discrimination is a procedure to find the most specific method in the given method-list.

Each methods bundled in a method-list, has a key for the discrimination. It is called a class-specifiers. Or more specific case, it is sometime called type-specifiers. Type is a subset of class, and is Common Lisp data type. Class(type)-specifiers is a list of classes (types) of the arguments for a method.

On DEFMETHing a method, it generates an internal form of the method and pushes the internal form on the current method-list of the specified selector. The internal form is from the idea in [ida85].

The following formation happens during defmeth.

```
  (defmeth selector ((arg1 class1) (arg2 class2) ... (argn classn))
        form1 form2 ... )
===>
  push (METHOD n (class1 class2 ... classn)
        (lambda (arg1 arg2 ... argn)
                (BLOCK selector form1 form2 ...) ) )
```
on top of the method-list for the selector,
where n is the number of arguments. ida85 has no 'n' paprameter. 'n' is introduced for efficiency sake.

Fig. 2 shows an example.

For a generic method, class-specifiers is a list of T, whose length is the same as that of arguments, and is automatically generated.

For a classical method, the second or later classes are not specified. Then default T is automatically inserted.

For a multi method, the complete set of user written class specifiers makes a class-specifiers for the method.

4. Class-specifiers based Discrimination

4.1 Basic primitives for the discrimination

Method discrimination is based on the relation of the class-specifiers. The primitives of the discrimination are
TYPE-OF and SUBTYPEP.

TYPE-OF function of Common Lisp takes an object and returns its type as the value. CLtL definition of the TYPE-OF is so simple that there are several different interpretations among the Common Lisp implementation. The author claims the returned value of TYPE-OF should be defined more in the X3J13 spec. For example, among the most interesting things there are;
(TYPE-OF ()) returns SYMBOL in KCL and VAXLisp both,
and
(TYPE-OF "a string ") returns STRING in KCL, but returns (SIMPLE-STRING 9) in VAXlisp. From the author's point of view, (TYPE-OF ()) should return NULL, which is the type for the symbol NIL. And, (TYPE-OF "a string ") should return SIMPLE-STRING.

SUBTYPEP defines the hierarchical relations among the methods.
If (subtypep x y) is true, then x is more specific than y.
So, SUBTYPE and EQ are the basic predicates for the method discrimination.

TYPE-OF and SUBTYPEP can be extended to have the same semantics for user defined classes in D-Eval. User defined classes are defined by an extended defstruct [bob86].

If the name of a structure defined by a defstruct can be viewed as a type, TYPE-OF also see it, such as;

```
>(defstruct foo x y z)
foo
>(setq x (make-foo))
...
>(type-of x)
FOO
>(defmeth func ((x foo)) ...)
func
>(func x)
...
```
This causes the invokation of the above method.

Furthermore, if :include relation can be understood by SUBTYPEP, it is possible to discriminate defstruct-ed classes in D-Eval.

```
>(defstruct (bar (:include foo)) q w)
bar
>(setq y (make-bar))
...
>(subtypep (type-of y) (type-of x))
t
; t
>(defmeth func1 ((x bar)) ...)
func1
>...
```
Func is called more specific than func1 at this point.

Taking account the above all discussions, TYPE-OF and SUBTYPEP in D-Eval is extended.

4.2 Search for the most specific class-specifiers as a kernel of Method search

Candidates for the most specific method which will be picked up and executed, are obtained through the search on the method-list of the specified selector. The primary rule is the same-length rule. The methods which have

different length against the type-specifiers of the current message do not participate the discrimination. For example, to find the most specific method for (foo a b), (method 3 (x y z) ...) can not be a candidate even though it is in the method-list of foo.

The body for the method discrimination is defined as follows

```
(do ((ftype type-specifiers (cdr ftype))
     (mtype types (cdr mtype)))
    ((null ftype)  ...)  ; a method which has 'types' is a candidate
    (unless
       (or (eq (car ftype) (car mtype)) (subtypep (car ftype) (car mtype)))
         (return ))     ; exit the loop
    )
```

where type-specifiers is the class-specifiers of the invoked message, types is the class-specifers of a method stored in the method-list of the selector. If the loop is finished on ((null ftype) ....), types is checked against the current candidate, which is a local variable of the outer loop which steps types in the method-list. After the outer loop is over, the current candidate contains the most specific method to be executed.

To cope with the multiple super, it should have a class-precedence list. In the type system of Common Lisp, NULL type and VECTOR type (and related subtypes of vectors, simple-'s) have a multiple supers. NULL has LIST and SYMBOL, while VECTOR has ARRAY and SEQUENCE. Bob86 defines LIST advance rule. ida85 defines sequence advance rule. These rules are stored in the class-precedence list for selectors. Multiple inheritance with :include, makes a class-precedence list for the class.

4.3 Multiple discrimination

The discrimination algorithm for multi method is based on the left-to-right principle, which is derived from Bob86. if the left most class specifier comparison can determine the relation, right part is not checked, like cond. For example, on comparing two method; a method which have (number symbol) as a class-specifiers and a method which have (fixnum t), fixnum specifier is more specific than number specifier, so symbol is not checked against T.

5. D-eval

Fig.3 shows the defintion of d-eval and d-apply. d-evlis is defined in [ida85]. D-Evlis is basically parallel to Evlis.

```
D-evlis (x) is
   (if x (pcons (d-eval (car x)) (d-evlis (cdr x)))
         (values nil nil))
```

where pcons is a primitive to cons each values parallely.

```
(pcons x y) =
   (multiple-value-setq x (x1 x2)),
   (multiple-value-setq y (y1 y2)),
   (values (cons x1 y1) (cons x2 y2))
```

For example, the values of (d-evlis '(1 'a "string")) are (1 a "string") as the first value and (fixnum symbol simple-string) as the second value. Fig.4 shows the examples of the conversation to D-Eval.

The current version of D-Eval has no provision for lexical scoping. Class definition is done with usual defstruct. So, static properties of Common-Loops are inherited from the CommonLoops concept.

6. Future work

Among the most interesting area to be explored for CommonLoops, the author think the design and implementation of the indivisual-method is the key to next step.

Acknowledgement

This research was partly supported by Grants in aid No.61750344 from the Ministry of Education, Science and Culture of Japan. The work to write a D-Eval at Computer Centre of Univiersity of Tokyo (ccut) was partly carried under the joint research with Prof. Haruhisa Ishida of ccut.

References

[Bob86] D.G.Bobrow, K.Kahn, G.Kiczales, et.al. CommonLoops: Merging Common Lisp and Object-oriented Programming, Proc. OOPSLA'86 ACM Sep. 1986

[ida85] M.Ida, An Interpretation of the CommonLoops specification, WGSYM 35-3. IPSJ. Dec. 1985

[Steele84] Guy.L.Steele, et.al. Common Lisp: the Language, Digital Press, 1984

```
D-Eval>(ndefmeth foo ((x number) y)(print 'foo-number-t) (cons x y))
selector = foo args = ((x number) y) forms = ((print 'foo-number-t)
                                              (cons x y))

foo
; symbol
D-Eval>(ndefmeth foo ((x fixnum)(sy symbol))(print 'foo-fixnum-symbol)(cons x sy))
selector = foo args = ((x fixnum) (sy symbol)) forms = ((print
                                                         'foo-fixnum-symbol)
                                                        (cons x sy))

foo
; symbol
D-Eval>(symbol-function 'foo)
d-apply entered. fn = symbol-function args = ('foo)
((method 2 (fixnum symbol)
         (lambda (x sy)
           (block foo (print 'foo-fixnum-symbol) (cons x sy))))
 (method 2 (number t)
         (lambda (x y) (block foo (print 'foo-number-t) (cons x y)))))
; cons
D-Eval>(exit-d-eval)
nil
```

Fig. 2 Method definitions and their internal representations : an example

```
D-Eval>(ndefmeth bar (x y) (print 'the-most-generic-case) (cons x y))

bar
; symbol
D-Eval>(ndefmeth bar ((x list) y) (print 'classical-method) (cons x y))

bar
; symbol
D-Eval>(ndefmeth bar ((x number) (y cons)) (print 'multi-method-1) (cons x y))

bar
; symbol
D-Eval>(ndefmeth bar ((x fixnum) (y list)) (print 'multi-method-2) (cons x y))

bar
; symbol
D-Eval>(bar 'a 'b)
d-apply entered. fn = bar args = ('a 'b)
the-most-specific-method-is
(lambda (x y) (block bar (print 'the-most-generic-case) (cons x y)))
the-most-generic-case
(a . b)
; cons
D-Eval>(bar 1 '(a b))
d-apply entered. fn = bar args = (1 '(a b))
the-most-specific-method-is
(lambda (x y) (block bar (print 'multi-method-2) (cons x y)))
multi-method-2
(1 a b)
; cons
D-Eval>(bar 1.2 'a)
d-apply entered. fn = bar args = (1.2 'a)
the-most-specific-method-is
(lambda (x y) (block bar (print 'the-most-generic-case) (cons x y)))
the-most-generic-case
(1.2 . a)
; cons
D-Eval>(bar 2/3 '(1 / 3))
d-apply entered. fn = bar args = (2/3 '(1 / 3))
the-most-specific-method-is
(lambda (x y) (block bar (print 'multi-method-1) (cons x y)))
multi-method-1
(2/3 1 / 3)
; cons
D-Eval>(exit-d-eval)
nil
```

Fig. 4 Examples of CommonLoops type method invokation

```
;;;;    Discriminating Eval
;;;;                              by Masayuki Ida 1986.10.06 1st version
(in-package 'pcl)

;;;; ---- D-eval ----
(defmeth d-eval ((x number))    (values x (type-of x)))
(defmeth d-eval ((x string))    (values x 'string))
(defmeth d-eval ((x character)) (values x 'character))
(defmeth d-eval ((x null))      (values x 'null))
(defmeth d-eval ((x symbol))
   ;; caution : this version does not take an account of lexical scoping or binding
   (let (( val (cond ((keywordp x) x)
                     ((eq x t)     x)   ; singular point in symbol
                     ((boundp x)   (symbol-value x))
                     (T     (error "Attempt to eval an unbound symbol ~s" x))
                     )))
                (values val (type-of val))))
(defmeth d-eval ((x cons))
    (declare (special x))
    (let (( val    (d-eval-cons (car x)(cdr x))))
        (values val (type-of val))))

;;;; ---- D-eval-cons ---- for d-eval with cons type
(defmeth d-eval-cons ((fn symbol) args)
    (declare (special x))
    ;; if (car form) is symbol,
    ;;   it should be a macro name, special-form name, or a function name
    (cond     ((macro-function fn) (d-eval (macroexpand x)))
              ((special-form-p fn) (d-eval-special-form x))
              ((fboundp fn) (d-apply fn args))
              (t (eval x)))
              )
(defmeth d-eval-cons ((fn cons) args)
        (cond   ((eq (car fn) 'lambda) (apply fn args))
                (T (error "illegal form for D-eval-cons ~s" fn))))

;;;; ---- D-eval-special-form ----
(defun d-eval-special-form (x)
        (cond
        ;; caution : only QUOTE is processed
              ((eq (car x) 'quote) (values (cadr x) (type-of (cadr x))))
        ))

;;;; ---- D-apply ----
(defun d-apply (fn args)
    (format t "d-apply entered. fn = ~s args = ~s" fn args)
    (let ((fn-body (symbol-function fn)))
      (if (and (consp fn-body)(consp (car fn-body))
               (eq (caar fn-body) 'method) )
          ;; then,
          (multiple-value-bind (x y) (d-evlis args)
                 (apply (most-specific-method fn-body (length y) y) x) )
                        ;; will find the most-specific method in fn-body
                        ;; keys are (length y) : number of the elements types in y
                        ;;          y  :  type-specifiers of the arguments
          ;; else
          ;;   escape to the usual apply
          (apply fn (d-evlis args)))
        ))

(defmacro ndefmeth (selector args &rest forms)
   (if (not (fboundp selector)) (setf (symbol-function selector) nil))
   (push
   (let ((types (mapcar #'(lambda (x) (if (consp x) (cadr x) t)) args)))
  (list 'method (length types) types
 ((list 'lambda (mapcar #'(lambda (x) (if (consp x)(car x) x)) args )
  (list* 'block selector forms))))

    (symbol-function selector)
  )
   (list 'quote selector)
   )
```

Fig. 3 D-Eval, D-apply definitions