記 号 処 理 35－3
(1 9 8 5. 1 2. 2 0)

An Interpretation of the CommonLoops specification

Masayuki Ida, Satosi Utida

Aoyama Gakuin University

1. Introduction

CommonLoops is a proposal of the Common Lisp language extension for object oriented facilities, and was appeared at IJCAI'85 Common Lisp meeting. The name COMMONLOOPS is from "COMMON Lisp Object Oriented Programming System". The language overview of CommonLoops was appeared as Bob85. But, many undefined or uncertain parts are left in Bob85. It has been discussed and improved through arpanet mail and other communication efforts for now. This paper describes an interpretation of the specification of CommonLoops for APCL by the authors through the discussion and implementation of a pilot version .

There are three points in Bob85 to discuss; One is the goals, another is the kernel specification, the other is the extensions (not fixed parts). As to the goals, this paper has no specific opinion but has summary in chapt. 2. As to the kernel specification, this paper try to make the kernel more clear. It will appear in chapt. 3. As to the extensions, this paper try to select most appropriate ones, and is described in chapt. 4.

2. Goals of the CommonLoops

The goals of CommonLoops are the follows;

1) Compatibility:::: CommonLoops is as compatible as possible with Lisp's functional programming style. Message sending uses the same syntax as function call. Object-oriented capabilities are simple extensions of Common Lisp. For the comparison to the usual Common Lisp terminology, following correspondence may assist to comprehension.

Usual CommonLisp	CommonLoops
type	class
slot	slot
function	method
function call	message send

2) Small kernel:::: CommonLoops provides a small kernel.

3) Powerful base:::: No need for higher level object languages for building interesting applications

4) Universal kernel:::: CommonLoops provides a base in which Flavors, Smalltalk-80 and LOOPS can be implemented. The major mechanism for this is the metaclass.

5) Common Lisp style:::: CommonLoops uses the same style of names and syntax as Common Lisp.

6) Efficiency:::: Using proven software techniques, CommonLoops can be implemented to run efficiently without special hardware support.

7) Extensibility:::: CommonLoops allows several desirable extensions.

3. the Kernel CommonLoops

This section summarizes the kernel, which is small and simple. Keywords for the specifications are multi-methods, class precedence list, method precedence list, and discriminator. Keywords for the language constructs are defmethod, defstruct extension, meta class, simple method combination.

3.1. Separation of definitions of classes and definitions of methods

From the authors' stand point of view, the major characteristics is the separation of class definition related syntax and method definition related syntax. As the consequence, simple kernel is possible. To define method, defmethod is newly introduced. To define class, defstruct syntax is extended.

3.2. Defstruct to define classes

Simple extension to defstruct syntax, such as multiple inheritance and meta-class.

CommonLoops adds a :class option to defstruct.

```
(defstruct (point (:class class))
                  (x-position 0)
                  (y-position 0))
```

(:class flavor) is a class with Flavors manner.
(:class class) is a class with LOOPS manner.

CommonLoops provides several standard meta-class; such as built-in-class, structure-class, list-structure-class, vector-structure-class.

3.3 defmethod, class-specifiers, method-precedence-list, discriminator object, remove-method

```
  (defmethod move ((obj block) x y)
              ... )
```

It requires obj should be a block class object. x,y may be of any class.

Class-specifiers is a list of class specifier. For the above example, (block t t) is the class-specifiers of the method for a selector 'move'. One of the built-in types and a name defined by defstruct can be used as a component of a class-specifiers.

It is possible to define several methods with the same selector. Methods related to the same selector is gathered into a special list. The list is called method-precedence-list or discriminator object. With the list, method search, specialization, and other operations are carried.

'remove-method' removes the method for the selector with class-specifiers. (remove-method 'move '(block t t)) is an example.

The syntax of defmethod is parallel to defun. See Fig.1. We design the internal form of a method as

```
 (METHOD class-specifiers
         lambda-expression).
```

Method-precedence-list is a list of the above form of method. This simple structure enables us to make method body equivalent to function body, i.e. lambda-expression.

Specialize function adds a new method to method-prcedence-list hold in the function cell, and arranges the order of the method-precedence-list of a selector. Key for ordering is class-specifiers. Aside the method-precedence-list, it had better to construct method cache. Method cache is a short term lemitted size memory for fast look up.

3.4. Message send nearly equal to function call, multi-method

Message sending to a selector has the same syntax as usual function invocation. There can be many methods with the same selector name. A method is selected and invoked only if all the arguments of it match the required specifications. The classical object oriented systems are special cases where only the first argument has its type specified.

For example, if there is a method 'move' with three arguments, moving block1 from position 33 to position 120 is described in CommonLoops as

```
 (move block1 33 120),
```

whose equivalent form in message sending style is

```
 (send block1 :move 33 120).
```

The interpretation of (move block1 33 120) is

```
 (funcall
    (method-specified-by 'move 'block 'fixnum 'fixnum)
    block1 33 120),
```

and is not

```
 (funcall
    (method-specified-by 'move 'block)
    block1 33 120).
```

Arguments for a selector are all effective for the discrimination. More generally, method-specified-by has a following form.

```
 (method-specified-by
    selector
    (type-of arg1)
    (type-of arg2)
    ...            )
```

where (type-of x) = (name-of (class-of x)) and the value is the most specific type.

This type of discrimination is called multiple-discrimination in CommonLoops. Methods discriminated by multiple-discrimination are called multi-methods.

Furthermore, we assume the number of arguments among methods with the same selector must be the same.

'self'is just the first argument in CommonLoops.

3.5. Slot Access

Defstruct for defining a class implicitly insert (:conc-name nil) defstruct option. Access functions are methods in CommonLoops. Then a slot name is also used as a selector for the method of accessor function (accessor method).

```
(defun foo (x y) ...)  => FOO
 <==>   (setf (symbol-function 'foo) `(lambda (x y) (block nil ...)))

(defmethod foo ((x cons) y) ...) => FOO
 <==>   (setf (symbol-function 'foo)
              (specialize (symbol-function 'foo)
                    `(method (cons t) (lambda (x y) (block nil ...)))
              ))
```

Fig 1 Defmethod is parallel to defun

3.6. Multiple Inheritance

Multiple inheritance is written with an extended syntax of :include defstruct option.

3.7. Method Combination

The primary mechanism for method combination is run-super. The following definition is inadequate but assist the basic understanding;
"run-super = run this method with the same arguments but do the next method that would have been done if this one had'nt been there" Aug. 7th 14:44 By L.Masinter

3.8. class, metaclass, and class precedence list

Class is a name of a defstruct (with :class) or one of the basic types of Common Lisp.

Metaclass is class of classes. The functions of metaclass are; class-instance creation, slot access function creation, class precedence list creation, and determination of the relation between different metaclasses. Several types of metaclasses are described in Bob85. But, they have the same (only one) name space. Differences are due to treatement of several options and the resulted specific actions. Flavor class is one example and it indicates the method combination is to be a Flavors style. the basic precedence among classes is type hierarchy of Common Lisp. The type hierarchy makes essential class metaclass.

Fig.2 shows the schematic metaclass modules. In the comments of Fig.2 program, there are 7 logical precedence list. Class precedence can not be expressed by one hierarchical list. One example is the relation between sequence and symbol. If nil (type 'null') is concerned, there is a relation such that sequence is prior than symbol. But, for any other than nil, there is no precedence relation. Normal left-to-right should be effective for the case. We add a rule that sequence is prior than array in the list from vector. With the rule and list advance (sgainst symbol) rule for nil which is in Bob85, there is no ambiguity in essential types. The class precedence among classes created by defstruct are,

```
 if there is no :include relation,
     then independent and disjoint,
 else
     a class-advance relation between a
class and the inherited class as shown in
Fig.2, is created.  For example,
  (defstruct (foo (:class class)
                  (:include (bar baz)))
            ... )
 then class precedences are
  foo => bar, and foo => baz.
 If bar is defined as
   (defstruct (bar (:class class)
                   (:include the-super))
           ... )
 then foo -> bar -> the-super.  the-super
and baz are independent.
```

4. An Interpretation of the Extension Discussions

In Bob85, there are many extension discussions and some proposals. In this section, we try to name each problem and to make comments.

4.1 individual-specialization problem

problem: the following is possible ?
```
 (defmethod find-file (name (host-name
'MIT-AI)) ...)
```
further problem: if possible, the above is more specific than the following.
```
 (defmethod find-file ((name string)
host-name) ...)
```

key: need modification of the normal left-to-right determining rule ?

answer: Yes. But do not need the modification of 'normal' left-to-right rule. an example is shown in Fig.3. Fig.3 also shows the contents of the function cell of foo, i.e. discriminator object. Method with individual specialization is consed to the top of the discriminator. Indicator for individual specialization method is I-METHOD as in Fig.3. I-method tells us the discriminator contains the individual method. I-methods are sequentially scanned first. Then other methods are checked.

related comments : we also exclude 'member' type specifier as bob85 says.

4.2 WITH problem

problem: will WITH-syntax such as follows be added?
```
 (defmethod move ((b block) new-x new-y)
   (with (b)
        (erase b)
        (setq x new-x y new-y)
        (draw b)))
```
x and y are assumed to be slots of 'block' defstruct, and are expected to be treated as usual lisp variable here.

key: should a slot look like a variable?

answer: we do not want to include with-syntax with the current clarity of the syntax and the benefit.

```
;
; Essentical class precedence      (schematic diagram)
;
; 1) string -> vector -> sequence -> array
; 2) null -> list -> sequence -> symbol
; 3) cons -> list -> sequence
; 4) standard-char -> string-char -> character
; 5) {fixnum , bignum} -> integer -> rational -> number
; 6) {short-float , single-float , double-float, long-float}
;       -> float -> number
; 7) ratio -> rational -> number
;
(defun class-advance ( x y) (put x y 'prior))
(class-advance 'string 'vector)
(class-advance 'string 'sequence)
(class-advance 'string 'array)
(class-advance 'vector 'sequence)
(class-advance 'vector 'array)
(class-advance 'sequence 'array)
(class-advance 'null 'list)(class-advance 'null 'sequence)
(class-advance 'null 'symbol)
(class-advance 'list 'sequence)
(class-advance 'cons 'list)(class-advance 'cons 'sequence)
(class-advance 'standard-char 'string-char)
(class-advance 'standard-char 'character)
(class-advance 'string-char 'character)
(class-advance 'fixnum 'integer) (class-advance 'bignum 'integer)
(class-advance 'fixnum 'rational)(class-advance 'bignum 'rational)
(class-advance 'fixnum 'number)(class-advance 'bignum 'number)
(class-advance 'integer 'rational) (class-advance 'integer 'number)
(class-advance 'rational 'number)
(class-advance 'short-float 'number)(class-advance 'short-float 'float)
(class-advance 'single-float 'number)(class-advance 'single-float 'float)
(class-advance 'double-float 'number)(class-advance 'double-float 'float)
(class-advance 'long-float 'number) (class-advance 'long-float 'float)
(class-advance 'float 'number)
(class-advance 'ratio 'number)(class-advance 'ratio 'rational)
(defun class-prior-p (x y)
        (let ((val (get x y))) (if val val (if (get y x) 'posterior nil))))
```

Fig.2 Class precedence list of Common Lisp essential types

```
(defmethod foo ((x 'cons) (y number)) ...) => FOO
(symbol-function 'foo)
  => ((i-method ('cons number) (lambda (x y) (block nil ...)))
      (method (cons t) (lambda (x y) (block nil ...))))
```

Fig.3 Individual specialization is indicated by I-method

4.3 LOOPS extension problems

There are five categories.

problem 1 : allocation option as a new slot option
    :allocation {class, dynamic, instance, none}
 'class' means 'shared by all instances and global'
 'dynamic' means 'automatic allocation',
    i.e. allocate the slot in the instance on the first appearance.
 'instance' is the default interpretation
 'none' means 'the slot name is inherited but will be prohibited/deleted in the class'

answer: Ok, we will try to implement.

problem 2 : undeclared slots
  add two functions; get-dynamic-slot and remove-dynamic-slot
  which look like plist mixin flavor of Flavors.

answer: we cannot realize the fruit of the implementation effort currently.

problem 3 : annotated value
  new slot option for get/put-FN; :get-function :put-function.
  One application is to implement active values of LOOPS.

answer: we agree the needs. but implement later.

problem 4 : use 'initialize' for the initialization of the slots and
 the constructors of defstruct with :class should use 'generic make', which enables run-time evaluation.

answer : we can not understand the intention of the problem.

problem 5 : changing classes
  can reorganize structure of a defstruct as a new class dynamically?
  and use 'change-class'

answer : we can not understand the intention of the problem.
 is it feasible to do so?

4.4 runsuper problem

problem: can change the arguments passed to the super method?

answer: no

4.5 method combination problem

problem: can allow Flavors style method combination ?

```
 (allow :before and :after ?)
 (defmethod (move :before) ...)
 (defmethod (move :after)  ...)
```

  In the method-precedence-list, :before methods, the applicable normal method (primary method), and :after methods are collected into a combined method.

answer: Yes we can.

4.6 REF problem

 problem: introduce a new function named 'ref' which is equivalent to a access function of slot but extended to undeclared slot.
 remarks: this problem has a relation to the 4.3 problem 2.

answer: we can not realize the fruit of the implementation of it.

4.7 mlet problem

problem: can allow lexical bind of the methods?
  mlet, mlabels are parallel to flet, lables.
 (mlet specializes the function-cell, while flet binds it)

answer: as arpanet discussion shows, we have a doubt of implementability.

4.8 discriminator primitives problem

problem: there should be two primitives, specialize and make-discriminator.

answer: we use 'specialize' function as (specialize discriminator method) in defmethod. we do not use 'make-discriminator'.

note: discriminator is a functional object normally created and installed in function-cell.

4.9 Complex specialization problem

problem: allow complex specialization specifiers such as;
 (or type1 type2 ..),
 (integer lower-limit upper-limit),
 (not type) and so on.

answer: no

4.10 lambda-list keyword problem

problem: allow lambda-list keywords, &optional, &rest, and &key ?

answer: methods for a selector must have the same number of arguments. So, we wanted to exclude &optional, &rest, and &key from methods for safety sake. But, it is possible to pass them to lambda-list of the method. For example,

```
(defmethod foo
   ((x cons) (y string) &optional (z 1))
   ...    )
 is processed as
  (method (cons string)
         (lambda (x y &optional (z 1))
         ...    ).
```

Then our system process lambda-list keywords as only lambda-list, not as sources of class-specifiers.

4.11 Method slot problem

problem: allow a new defstruct option, :method-slots for virtual slots.

```
 (defstruct
    (steam (:class method-slot-class)
           (:method-slots
(tyo 'default-tyo)
                (tyi 'default-tyi)))
```

tyo, tyi are virtual and slot access for them means method invocation. (Methods like (defmethod tyi ....) (defmethod tyo ...) are called.)

answer: we need the clarity of the syntax.

5. Discriminating-Eval and D-evlis

Discriminating-eval (d-eval) is an eval-like language construct defined by the author. D-eval is used for universal discrimination base of method. The major difference between usual eval and d-eval is the latter is the multiple value function, on the other hand the former is the single value function. Since our interpretation prohibits multiple valued multi method, there is no problem. The return values of d-eval are;
evaluated value, which is the same value as the usual evaled, and, the class name of the evaluated value. for example,

```
 (d-eval 1) => 1 ; FIXNUM
 (d-eval "Common Loops") => "Common Loops"
; SIMPLE-STRING
 (setq a '(x y))
 (d-eval a) => (X Y) ; CONS
 (setq a ())
 (d-eval a) => NIL ; NULL
```

D-eval is the quite parallel with the usual eval, like a man on the sunny side road and his shadow. If a method is to be evaluated, d-eval is used mainly inside the evlis. Arguments for a method is processed by d-evlis (discriminating evlis), which combines each multivalue into two lists. Arguments of a method to be evaluated are processed using d-evlis. for example,

```
 (foo 1 nil " string ")
 ==> (d-evlis (1 nil " string "))
  => (1 nil " string ") ; (FIXNUM NULL
SIMPLE-STRING)
```

==> (1 nil " string ") is used for the arguments for FOO,
while, (FIXNUM NULL SIMPLE-STRING) is used for searching the appropriate method with a selector 'FOO'. (FIXNUM NULL SIMPLE-STRING) is logically compared to the class-specifiers of methods.

The schematic definition of d-eval and d-evlis are shown in Fig.4 and Fig.5.

Future works

We want to implement a complete object oriented facility of Common Lisp. We think CommonLoops is the base. After asking the conclusion of december meeting of US committee, we will go on the implementation on our machine. On the other hand, the first author want to discuss our implementation by the Jeida Common Lisp committee and object oriented (CommonLoops) working group.

Acknowledgements

The authors would express their gratitude to Dr. Kenneth Kahn of Xerox PARC.

Reference

[Bob85] Bobrow D.G., Kahn K., Kiczales G., Masinter L., Stefik M., Zdybel F. "COMMONLOOPS" pre-IJCAI'85 draft, ISL-85-8, XEROX PARC 12 August 1985

[Ste84] Steele Guy L., et.al. Common Lisp:the language, Digital Press 1984 (japanese edition by M.Ida, kyouritu pub. Co. 1985)

```
(defmethod d-eval ((exp cons))
                             (cond ((symbolp (car exp)) ... )
                                    ((eq (caar exp) 'lambda) ...)
                              ... ))
(defmethod d-eval ((exp fixnum))  (values exp 'fixnum))
(defmethod d-eval ((exp bignum))  (values exp 'bignum))
...
(defmethod d-eval ((exp symbol))
   (if (null exp) (values nil 'null)
       (let ((slot (bind-val exp)))
              (if (or (null slot) (eq (value-part slot) 'special-value))
                  (values (symbol-value exp) (type-of (symbol-value exp)))
                  (values (value-part slot) (type-of (value-part slot)))
              )
        )))
```

Fig 4   Overview of D-eval methods

```
(evlis args)   <==> (do (val (args args (cdr args)))
                        ((null args) val)
                        (rpush (eval (car args)) val))
(d-evlis args) <==> (do (val classes (args args (cdr args)))
                        ((null args) (values val classes))
                        (multiple-value-bind  (a b)
                                (d-eval (car args))
                                (rpush a val)
                                (rpush b classes)))
(rpush x y) <==> (if (consp y)
                        (rplacd (last y) (list x))
                        (setf y (list x)))
```

Fig. 5 Parallel description of D-evlis and evlis