記号処理 3 － 1
（1978. 2. 9）

Function-class —— its definition and application

Toshiaki Kurokawa
Information Systems Lab., TOSHIBA R & D Center

Abstracts

We present a new declaration and classification method for functions or procedures, which distinguishes the internal body of function and the external subprocedure for function invocation. And that external procedure is divided along two coordinates: input vs. output, and requirement vs. procedure.

New concept, 'function-class', is introduced in this paper along this line. The function-class concept promotes 'a declarative kind of function definition' and opens 'an easy way to write assertions for procedures'.

It will be applied to the area of program writing, program verification, compiler optimization and automatic program analysis, i.e. to the automated software production system.

Introduction

There are two problems which gave birth to this paper. The one is the classification problem on the program. When the author made an attempt to make another LISP system which would be a child of LISP 1.9 [1], the problem was the great number of built-in functions in LISP 1.9. A method to group and classify these functions is necessary.

The other problem is the software production problem. The more efficient method to make and write a program is now a popular and a hard problem. As the author is employed to the LISP language, it is, in other words, how to write a function in a more efficient style.

This paper presents a step to the solution which is called a function-class concept. A function-class can be considered to be an extension of a function type, but it has a more formal definition and has much power. It provides a kind of declarative way of program writing.

We present the definition of the function-class and its calculus in chapter I, which is the main purpose of this paper. In chapter II, an informal description of function-class is given. And the application is given in chapter III where the function-class is applied to LISP language (esp. to LISP 1.9), the LISP specialities (i.e. body type) are explained, and some basic classes in LISP are given.

At last some utilizations of function-class are given. They are program construction, program modification, program checking and compiler optimization.

The term 'function-class' is adopted to distinguish the existing term, 'function type'. Also the effect of the term 'object class' in the SMALLTALK [2] cannot be neglected. If you consider the term 'function' as a class name for the functions, then you should interpret 'function-class' as 'subclass for the class of functions'.

I. Definition of Function Class

(1) function, input, output

In this paper, we use the term "function" in the most usual meaning in the programming language circle; function has input and output,

such that output = function (input), as shown in Fig. 1.

We include procedure to function; a procedure is a special function such that output = ϕ, i.e. output is not necessary. This kind of interpretation of procedure is already adopted in the language LISP. [1]

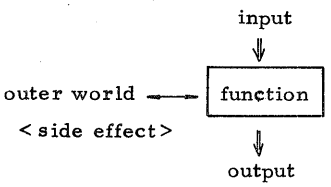

Fig. 1 Input, output and function. Side-effect is shown as the interaction with outer-world.

(2) an internal structure of function

A function has its internal structure as shown in Fig. 2. A function consists of 5 parts; procedure for input (IP), requirements for input (IR), requirements for output (OR), procedure for output (OP), and the body of function. Not all functions have the above 5 parts. We denote the null part by ϕ.

Both requirements are represented in an expression of elementary predicates; i.e. a finite set of predicates combined with operators and, or, not. These requirements should not have the side-effects. And that they have no effects on the input or output.

If an input or output does not meet the requirement, then the requirement violation (i.e. error) will occur and the execution of the program will stop.

Procedures are represented in a finite sequence of atomic processes. Input procedure processes input, i.e. input is modified. And output procedure processes output.

In some cases, the procedure will interact with the outer-world and input procedure returns the output value (without invoking the body of function). The trace mechanism of a function can be represented by these input/output procedures.

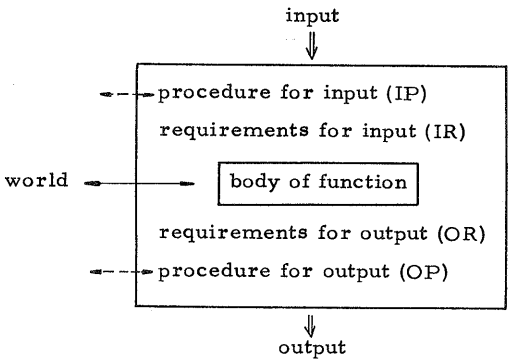

Fig. 2 Internal structure of function

(3) Definition of the function-class

Function-class is defined by the 4-tuple < IP, IR, OR, OP >.

Thus the function can be defined by a pair, < function-class, body >. A function f belongs to a class fc if f is defined by < fc, body >.

In this way, the programmer can concentrate the construction of the core body of function, aside from the bothersome input, output trivialities.

Function-class is an extension of function type which is used merely for convention in the existing language, and has no clear structures and no way to introduce new types.

Function-class has, however, a clear structure. And the user can introduce his own function-class, and that it is possible to define new classes concisely as explained in the next section.

(4) Calculus on the function-class

There are 2 ways to introduce a new function-class. The one is to define a new class with the full 4 elements; IP, IR, OR and OP. The other is to construct a new class

from the elementary function-class using the following operators; *, -, +,

(4.1) * (multiplication)

Definition Let a and b are function classes. $a = < IP_a, IR_a, OR_a, OP_a >$, $b = < IP_b, IR_b, OR_b, OP_b >$. A new class $c = a * b$ is defined as follows: $c = < IP_a \cdot IP_b, IR_a$ and IR_b, OR_a and $OR_b, OP_a \cdot OP_b >$.

In this definition, procedural concatenation $P_a \cdot P_b$ means the following procedures:

begin execute $P_a \cap P_b$;
execute $P_a - (P_a \cap P_b)$;
execute $P_b - (P_a \cap P_b)$;
end

In otherwords, if $a = c * d$ and $b = c * e$ then $a * b = c * d * e$. And that $a * a = a$.

Figuratively speeking, $a * b$ is an intersection of classes a and b as shown in Fig. 3.

It should be noted that the commutative law does not hold because $P_a \cdot P_b$ may not be equal to $P_b \cdot P_a$.

(4.2) - (differentiation)

Definition If function class $a = < IP_a, IR_a, OR_a, OP_a > = a_1 * a_2 * \cdots * a_n$, then $a - a_i = a_1 * a_2 * \cdots * a_{i-1} * a_{i+1} \cdots * a_n$.

This '—' operation is the reverse operation to '*'.

(4.3) + (addition)

This '+' operator has an extension operator. It has 2 way of extension; requirement extension and procedure extension.

(4.3.1) requirement extension

Definition Let function-classes a and x be such that, $a = < IP_a, IR_a, OR_a, OP_a >$ and $x = < \phi, IR_x, OR_x, \phi >$ then function-class $a + x$ is defined to be $< IP_a, IR_a \vee IR_x, OR_a \vee OR_x, OP_a >$, where $\vee$ denotes a or operator for predicates.

(4.3.2) procedure extension

Definition Let function classes a and x be such that $a = < IP_a, IR_a, OR_a, OP_a >$ and $x = < IP_x, \phi, \phi, OP_x >$ where $P_x =$ if pred then proc fi. A function class $a + x$ is defined to be < if I_{pred} then I_{proc} else IP_a fi, IR_a, OR_a, if O_{pred} then O_{proc} else OP_a fi >

Requirement extension may be equivalent with '-' operation on the requirements.

If P_a has the form, if P then Q fi, we can introduce the 2 types of procedural extension. If the both predicates in P_a and P_x are exclusive ($P \wedge_{pred} = \phi$) then we call the extension, if pred then proc else if P then Q fi fi, a natural extension. This synthesis extends the domain of the procedure. Another equivalent form will be case pred do proc; P do Q; end-case.

If predicate P_x is included in that of P_a ($P_x = P_a \vee \alpha$), then we call the extension if pred then proc else if P then Q fi fi, a supplementary extension. Another form of the synthesized procedure will be if P then if pred then proc else Q fi fi.

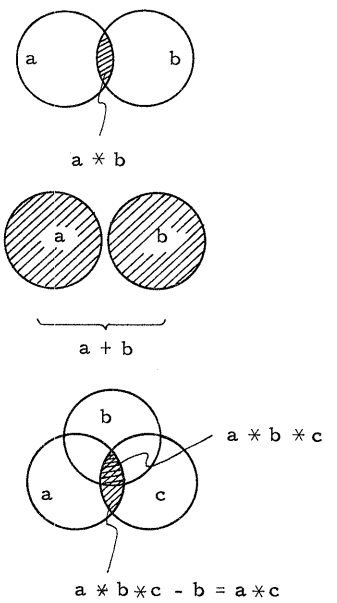

Fig. 3 An illustration of the calculus on function-classes.

(5) Declarative definition of a function

In this section, we present some example forms to define function in declarative ways. Considering the power of function-class, we will have other declarative definition formats later. Examples are adopted from the LISP language.

(5.1) Unification

Definition Let f and g be functions such that f = < fc, bodyf >, g = < gc, bodyg >. And that $IP_{fc} = IP_{gc} = IP$ and $OP_{fc} = OP_{gc} = OP$. h = h + g can be defined as follows: h = < hc, bodyh >. hc = < IP, $IR_{fc} \vee IR_{gc}$, $OR_{fc} \vee OR_{gc}$, OP >

bodyh = if IR_{fc} then bodyf.
else if IR_{gc} then bodyg fi fi

(example)

EQUAL = EQ + NUMBEREQUAL + STRINGEQUAL + LISTEQUAL, where EQ checks the address equality, NUMBEREQUAL checks the numeric equality, STRINGEQUAL checks character sequence equality, and LISTEQUAL checks list structure equality.

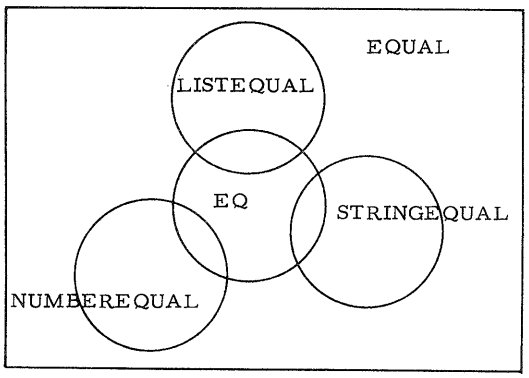

Fig.4 Unification of functions. EQUAL's domain is the union of the domains of EQ, NUMBEREQUAL, STRINGEQUAL and LISTEQUAL.

(5.2) Function-class change

f = < fc, bodyf > is a function. A new function g is defined using the body of f:
g = < gc, bodyf >.
define g class = gc body = body-of f end

(example)

SET, SETQ, SETQQ share the same body that is (set-value x0 x1).

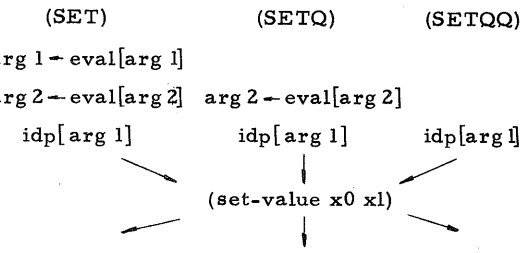

Fig.5 SET, SETQ, SETQQ share the same body.

(5.3) Macro expansion

f = < fc, bodyf > is a function. A macro expansion of f is defined as follows:
define g class = macro (x)
body = macro-expansion (x) of f end
where x denotes the macro expansion type.

(example)

2-argument functions such as *PLUS, *APPEND are macro expanded to multi-argument functions as PLUS, APPEND.

(6) Relations about function-class

We define the predicates about inclusion and element.

Definition A function-class c is called to be included in a function-class d when d = $d_1 * c * d_2$ where d_1, d_2 may be ϕ.

We write the above relation $c \subset d$, borrowing the notation of set inclusion.

We also define the relation 'element' for function and its class.

Definition A function f belongs to a function-class c if f = < c, body >.

We write the above relation $f \in c$.

The following definition extends this relation.

Definition A function f belongs to a function-class d if there exists a function-class c such that $f \in c \subset d$.

(example)

A function-class PREDICATE has an output requirement that the output is a boolean value (tree or false). EQUAL, GREATER, NUMBERP belong to PREDICATE. A function class DATA-TYPE-PREDICATE is included in PREDICATE. As a function INTEGERP belongs to DATA-TYPE-PREDICATE, so INTEGERP $\in$ PREDICATE.

We get the following theorems about the operators, +, -, *.

(1) $y \subset y+z$, $z \subset y+z$

(2) $y \subset y-z$

(3) $x*y \subset y$, $x*y \subset x$

These inclusion and element relations are efficient tools to classify functions. And that a compiler can use them in efficient manners.

II. Informal description about function-class

1. Synfax for function definition

(1.1) Standard definition

define <function-name> class = <class expression>
body = <body expression> end

<class expression> = <class name> |
<class expression> <class operator>
<class expression>

<class operation> = * | + | -

<body expression> = <system dependent codes> | body = of <function name> |
macro-expansion (<macro type>) of
<function name>

In the case of LISP language, <system dependent codes> can be λ-expression or lap-codes.

(1.2) Unificational definition

define <function name> union-of <function name sequence>
end

2. Syntax for function-class definition

define-function-class <class name>
<class body> end

<class body> = <class expression> |
IP = <function> IR = <predicate>
OR = <predicate> OP = <function>

If <function> or <predicate> does not exist, then ϕ will be placed.

III. Applications to LISP language (esp. on LISP 1.9)

1. body type

To apply the function-class concept to LISP language, the function-class should be an extension of function-type in LISP.

The problem is that LISP function type will be change after compilation. And that in some systems, hand-coded type and compiled type are different function types. As the function-class denotes the outer subprocedures of a function, the function-class should not change after compilation.

We introduce the body type to fill this gap. In LISP 1.9, there are 3 body types; lambda, index-register, and stack. Lambda type body is expressed in λ-expression. Both index-register type and stack-type are expressed in lap-codes. Only the argument passing process differs; index-register or stack.

Once after the actual arguments are set through input procedure, the formal-actual argument binding (or argument passing through index-register or stack) are executed in the body-invocation process.

We also introduce the predicate lambdap, xrp, and stackp to check the body type, and that these predicates can be used in the input requirements.

Thus after compilation, only this part of body type predicate will change. And that if a function is defined without body type require-

ment, then no change is necessary.

2. arglist and value

We denote the arglist for actual argument list (i.e. input to the function body) and the value for the output value.

(example)

(1) For the expr-type, input procedures will be arglist ← mapcar[eval, arglist]. We will call this procedure eval-proc, and call the function class eval whose input procedure is eval-proc and other 3 parts are ϕ.

(2) For the trace-type, the input procedure will be print [list[function, arglist]] and the output procedure will be print [list[function, value]] , where function denotes the traced function name.

3. basic function classes

We will give some of the basic function class for the basic functions in LISP.

(1) lambda = $<\phi$, lambdap, ϕ, $\phi>$

(2) xr = $<\phi$, xrp, ϕ, $\phi>$

(3) stack = $<\phi$, stackp, ϕ, $\phi>$

(4) eval = < eval-proc, ϕ, ϕ, $\phi>$

(5) f = < arglist ncons[arglist], ϕ, ϕ, $\phi>$

This is for the fexpr type.

(6) ℓ = < lexpr-process, ϕ, ϕ, $\phi>$

```
lexpr-process =
      begin
         push-mark (lexpr);
         for x in arglist do push (x)
            endfor;
         arglist ← ncons[length[arglist]]
      end
```

(7) m = < arglist ← cons [function, arglist], ϕ, ϕ, eval[value] >

This is for the macro type.

Now we have the following function types expressed with the above basic classes.

```
expr  = eval * lambda,
subr  = eval * xr,
csubr = eval * stack,
fexpr = f * lambda,
fsubr = f * xr,
lexpr = eval * ℓ * lambda,
lsubr = csubr,
clsubr= eval * ℓ * stack,
nexpr = lambda,
macro= m * lambda
```

(8) ev_(n) = < arg[n] ← eval[arg[n]], ϕ, ϕ, $\phi>$

The procedure, arg[n] ← eval[arg[n]] means that the nth argument is evaluated.

The function-class of SETQ will be arg_2*ev_2*id_1.

The procedure eval-proc can be defined as

```
for i from 1 to length[arglist] do ev_(i)
end-for.
```

(9) num_(n) = < ϕ, numberp[n], ϕ, $\phi>$

(10) id_(n) = < ϕ, idp[n], ϕ, $\phi>$

(11) def args_(n) = < ϕ, length[arglist] = n, ϕ, $\phi>$

The number of arguments are definite.

(12) args_(n) = < args-n-procedure, ϕ, ϕ, $\phi>$

The args-n-procedure is a kind of matching procedure which matches the actual-formal number of arguments. It is defined as follows:

```
if length[arglist]<n; n=length[formal_arguments]
  then for i from 1 to n - length[arglist]
    do arglist ← nconc[arglist, '(nil)] end-for;
  elseif length[arglist] > n
    then arglist ← sublist[arglist, 1, n]
  fi fi
```

The function sublist is defined as follows;

```
define sublist class = eval * num_2 * num_3 *
defargs_3
 body = (lambda (x y z)
        begin local n
          n ← length[x];
          for i from 1 to (y-1)
            do x ← cdr[x] end-for;
          x ← reverse[x] ;
          for i from 1 to (n-z)
            do x ← cdr[x] end-for;
          return reverse[x] ;
        end
```

end-define

4. Some utilizations of function-class

We will briefly survey here in what kind of areas this function-class can be utilized.

(4.1) Ease of program construction

This is one of the main object of the function class attempt.

(4.2) Ease of program modification

This is also a direct effect. The function definition format which change the function class can be thought of a program modification.

If a user wants to modify a function, he can do this, only to append some piece of codes to input or output procedure to its function class (i.e. slightly change its function class), without altering its body codes.

(4.3) Program checking

Not to say that the function-class provides a verification tool, but it provides an efficient way of program checking. Requirements part of function-class can check both input and output of the function, and can check to see if the output is reasonable or not according to the input. Requirements can be regarded as 'assertions' for the procedure.

Generally speaking, it is not an easy task to attach precise assertions to programs. In the case of function-class system, the user need not write the requirement checking codes to his function. He has only to select the function-class whose requirement meets it. For example, if you write an arithmetic function and that you want to avoid the case when the non-numeric arguments are supplied, then you only have to declare the function belongs to numeric function-class where both input and output requirements are 'numberp'. Further if the scope of the numeric argument is limited (eg. positive or integer), then you can make a new function-class which is a subclass of the numeric function-class.

We have also an idea that we can determine the requirement automatically through the trial execution of the object program. The utilization of the input/output procedure is, of course, a key idea that we can gather the dynamic information of the arguments and value of the function and that we will analyze it and abstract its property as the requirements for the function.

(4.4) Compiler optimization

The compiler can utilize the information of the function-class and is possible to generate much more optimized codes. First, the compiler can utilize the body-of declaration in the function definition. Multiple functions can share the same code for the body, so the number of codes can be made smaller.

Second, the compiler can check the requirements at the compile time. If it is proved that the input meet its input requirement, then the no-code generation for the requirements is necessary, i.e. you can delete the code for dynamic checking. Further, the compiler can serve a wider range of cautious execution type. It can ignore all requirements which are proved neither true nor false. Or it inserts the all requirements' codes when it cannot determine the value at compile time.

Third, some of the input/output procedures can be executed at compile time. For example, the argument-number matching process can be executed at compile time, when the arguments are already given in the text (or list) format.

Fourth, if the requirements information and procedure information are combined, then the compiler can eliminate a part of input/output procedure and a part of input/output requirements.

For example, in the case of numeric

function whose argument is fixed at the input procedure, you can eliminate the fixing process when you can conclude its argument must be integer because the argument is a function (f(x)) and the output requirement of the function f is integer.

Although the thorough application of proof procedure is perhaps impossible today, we believe that the partial application can give the great power to the compiler to generate a good code.

Summary

The definition of the function-class is presented. It has the four components; input procedure, input requirement, output requirement, and output procedure. The calculus on the function-class is also defined. The operator among them are *, -, and +.

Using the function-class, a new way of defining function is given. It can be said a declarative definition where a calculus (an expression made from a set of functions and function-classes) is given instead of a sequence of codes.

An application is given to the LISP languages, and it is suggested that the function-class is a powerful tool for the program writing, modification, checking, and optimization.

We have a plan to reconstruct an existing LISP system using this function-class.

References

[1] LISP User's Manual, ETL and Toshiba, 1977.

[2] SMALLTALK manual, Xerox PARK, 1976.