記号処理 12 − 13
(1980. 6. 20)

A Note on Variable Bindings and Function Invocations

Toshiaki Kurokawa

Information Systems Laboratory, TOSHIBA R & D Center
Kawasaki 210, JAPAN

Abstract

The problems of the variable binding and the function invocation are discussed. The considered system is limited to the Lisp language, but the materials are applicable to any other programming languages.

After the brief look at the current schemes, the requirements of the new scheme are listed. User control, distributed control, advanced user interface, efficiency, and debugging and maintenance are the targets for the new system.

A solution is proposed. It utilizes the function class method and the database facilities to satisfy the above requirements.

§1. Introduction

The variable bindings and the function invocations are classical problems. It can be thought as a solved problem. However, there remain lots of problems to be solved, especially from the engineering viewpoints.

It is true that the great number of programming languages has proposed lots of solutions. Restricting the theme to the Lisp languages, there are pattern directed invocation and binding, shallow and deep bindings [1,2], and spaghetti and macaroni stacks [3,4].

Still, the Lisp users employ the classical Lisp 1.5 or Lisp 1.6 binding and invocation schemes. A reason might lie in the conservative tendency of the human beings. It may be owing to the simple beauty of the classical system. It may be only because the new scheme is not available on the hardware which the user can access.

In this paper, the author tries to discuss about what kinds of variable bindings and function invocations are desired, and what solutions will be available in the present technology.

To pursueing this kind of work, it might be advised that the establishment of the application area is indispensable. It is true in the case of implementation, because the elimination of the 'now - unnecessary' elements leads to the simple and efficient system.

However, in this paper, the author would like to discuss from the unrestricted viewpoints so that the obtained results can be applied to the any areas, any languages, and any systems.

The materials are, in fact, rather restricted to that of Lisp language owing to the fact that Lisp has more interesting features than any other languages and the author's present interests lie on the Lisp system.

§2. Targets of the new scheme

The new scheme should satisfy the new requirements. There are five major requirements:

(1) User control

The fatal drawback of the current scheme is that the only interpreter (or compiler) can control the variable binding and the process activation.

It is true that the user can establish his own interpreter to handle his own scheme, but it is not efficient nor easy to control whole system.

It is necessary that the user can control the basic mechanism of the system.

(2) Distributed control

This requirement is a corollary of the above. It is necessary to distribute the control of the binding and activation so that the user can share the control of the system.

In other words, the current problem lies in the fact that the large interpreter handles all the details of function invocations.

The current situation is that the big interpreter is too difficult to understand. It is not easy to extend the facilities of the interpreter. It is also difficult to create a compact application system in Lisp. In order to supply a small application program, much of the big interpreter is not necessary, even if it does not cause any overhead.

(3) Advanced user interface

Why the users want to have lots of kinds of the variable binding and the process activation is that it provides a comfortable, advanced user interface.

There are several experiments or proposals for this interface. Some of them are as follow:

(i) variable binding through pattern matching

This is an idea that you need not describe the full details about how to pass the arguments.

(ii) function invocation through pattern matching

A function will be activated when the suitable pattern is presented. The user need not denote the name of the function. Non-deterministic parallel processing can be programmed through this scheme. It is usual that the variables are bound as the side-effect of this function invocation.

(iii) variable binding in the form of assignment

Generally Lisp binding follows the lambda binding, i.e. only the order of the formal and actual arguments has the effect of matching. In some cases, especially when lots of arguments are employed, it is convenient to denote the assignment directly. For example, (READ file = "Kurokawa" linenumber = None eof-handler = (Ask-other-file) character = EBCDEC) is more convenient to write and/or read the program.

(iv) default setting

The default setting mechanism is an important ability of the human beings. The frame theory [5], so it seems to the author, depends on this default system.

Some functions need complicated arguments, most of which are only necessary in the special environments. In the above example, if it is known that the character form of the file is usually EBCDEC, it is not necessary to declare EBCDEC only if the default mechanism works behind.

The default provides not only the pleasant user interface but also the applicability of the system. The READ function from default source can work on any actual devices (files, cards, paper tapes).

In summing up the user interface, two facts can be pointed out:

1) It is preferable that the user can describe the full process of the variable bindings and the function invokation.

2) Although there are lots of user interface mechanisms to be considered in general, it is not realistic to provides plural interfaces for each function.

Each function has its own interface which may differ from the interfaces of other functions.

In other words, it is not necessary to have a huge interfacing routine in the interpreter, instead it is preferable that each function has its own interfacing routine which will be shared with other functions.

(4) Efficiency

The new scheme must be efficient. If the efficiency is not concerned, the user will establish his own scheme on the existing interpreter system.

There are three major possibilities for the efficient processing:

(i) parallel processing

The variable evaluation and binding process can be parallelized as well as the body of the function. It is reasonable to assume that some classes of functions are difficult or unnecessary to be parallelized. It is desirable that a function can be declared that it hopes to be compouted in parallel.

(ii) coexistence of shallow and deep bindings

It is needless to repeat the disputes over the shallow vs. deep here in this paper. It is well known that in general the shallow binding scheme is efficient. However, when the complicated situation occurs such as hypothesis handling or context switching, the deep binding scheme is efficient.

From the users' point of view, it is better to have both schemes until a new efficient scheme over any circumstances will be invented.

It is doubtful that the dynamic mixing of the shallow binding and deep binding is technically feasible, but the mixing of the both schemes is far better than the mere coexistence and switching of them.

(ii) sharing the codes

When lots of binding and invoking schemes are provided, it is necessary to acknowledge what parts of the functions are common among many. It is necessary from the engineering standpoint to extract the common parts and share the parts among lots of functions.

Otherwise, the efficient and inefficient codes will be dispersed all over the user system.

(5) Debugging and maintenance

It is well known that how the variables are bound and how the functions are activated are very important information for debugging.

The new scheme should have at least the following facilities:

(i) requirement checking

The functions and the variables have their own requierments. The new scheme should provides the facility to declare and check the requirements. The checking includes the tracing the values and environments and the trapping of the process when the predefined requirement violation occurs.

If this type of the checking has to be processed efficiently, it must be distributed to each objects so that the checking will not cause any overhead on the irrelevant functions and variables.

(ii) detailed reference of the variable

It is necessary to refer all the variables at any time under the debugging process. However, in the block structured language, it is difficult to refer the upper level variable which is now declared to be local.

In the debugging stage, the variable should be referenced in what function's body and in which block's. When the context mechanism is employed, the context should be able to be referred.

(iii) systematic modification
The new scheme should enable the easy modification, i.e. the systematic modification instead of the current tedious and bug-generating hand modifications.
The requirements of the functions and variables should be changed. The arguments of the function can be modified both in number and types.
In addition to the above major targets, there are several facilities which will increase the power of the new scheme but will be difficult to include.

(6) Object-level linkage
It is useful to have a linkage through other modules generated by other languages. Lisp is not an easy language to link with other language, when you claim the efficiency on space and time.

(7) Database
In fact the Lisp language accompanies with a kind of database; OBLIST and property lists. The database facility is the key issue for the new scheme. However, it is not likely that the new scheme can easily incorporate the general-purpose database facilities in the whole.

§3. A solution for the new scheme

In this section, a solution is proposed to satisfy the requirements in the preceding section. The solution is based upon the two basic concepts; the function-class and the database of symbols (functions and variables).

The detailed explanation of the function-class will not be given here, but the interested reader can consult the published paper [6,7]. In short, the function class method enables the user control over the variable binding and function activation. The process will be described under the function class to which several functions will belong.

The database is necessary in general. The specific need for the database lies in the handling of the pattern match oriented function invocation and the detailed reference of the variables. In fact, the detailed history of the system utilization might be included in this database.

The introduction of the function class reduces the interpreter into the following form:

```
eval(exp) = begin
case exp ε constant then exp;
     exp ε reference then get-reference
                                  (exp);
     exp ε list then begin local func,
                                 args;
                 func: = car(exp);
                 args: = cdr(exp);
               case func ε function-name then
                    class-exec(get-class (func),
                               args, get-body (func));
                    func ε lambda-expression then
                       lambda-exec(func, args);
                    otherwise
                        apply(eval (func), args);
             end case
                              end
      end case
end
```

The function has its class and its body. Several functions can share the body (Req. (4)-ii) and share the class.

The class prescribes how the actual arguments are processed (i.e. evaluated), how the requirements are satisfied, how the body is activated. Most requirements (Req. (1), (2), (3)-i, iii, iv, (4)-i, iii, (5)-i, iii) will be satisfied by the mere introduction of the function class.

The main work will be shifted to provide as many function classes as possible, eg. pattern matching class or parallel processing class.

The database consists of the two main parts. One is the database of the variables. The record will be the following (in the form of the relational database): (variable-name, enviornment, value) where environment denotes the environmental information, i.e. in which context, in which function, in which block. Additional information about the variable will be useful; eg. reference only, modified, range of the expected value and so on. The requirement (5)-ii will be satisfied through this database.

The other part consists of the data on functions; in what situation the function should be activated, what functions are activated now, what kinds of classes are registered and so on. The form of the datum might not be fixed because the necessary information will vary in its amount and its properties.

The coexistence of the shallow and deep bindings can be realized through the utilization of this database.

The variable database provides the total information so that it is easy to establish the shallow binding system where the variable name with the current environment determines the value and the deep binding system where one can search the value through the guidance of the name and structure of the environment.

Problems of this solution

There exist problems on this solution except the fact that the solution need considerable resources.

(1) binding and unbinding
In the current scheme, the bound variables have to be unbounded after the execution. It means that the function class provides the way not only how it will be bound but also how it will be unbound. Actually the unbinding is itself a kind of bindings, i.e. binding the old value. Thus it might be necessary to hold the old values of the variables.
This overhead might be diminished when the variable database is fully utilized. However, it might be helpful that the basic binding and unbinding facility might be associated with the body of the function, not with the function class.

(2) parallel processing
The parallel processing invokes the problems known in the area of operating systems, i.e. how to handle the multiprocessing.
A process might be halted, not terminated. Later it will be resumed again. When the process is halted, can the variables in the process be referenced or not? It is not easy to answer this kind of questions.

(3) pattern directed function invocation
In the language like Planner [8] or Conniver [9], this type of invocation is a main one, and the overhead can be permitted.
In the proposed situation, this might be realized by the time consuming interrupt facility or by the special function PATTERN-DIRECTED-INVOCATION. These solutions are not very good, because it may cause some troubles to the user.
There also exists a problem about how to do when unexpected functions are invoked through this mechanism. Some kinds of fireman facilities will be necessitated.

§4. Summary

In this paper discussed are the requirements and the possible solution of the new scheme for the variable bindings and the function activation.

Both of the requirements and the solution are not perfect. The author would appreciate the any comments about the materials in this paper.

The subjects on this paper should not be limited to the Lisp language (the present target is the Lisp language, though). Most of the requirements discussed can be, the author believes, applied to any existing languages.

The critical point is that it is really necessary to have a specialized hardware to satisfy these requirements in the efficient system.

References

[1] Baker, H. G. Jr., "Shallow Binding in Lisp 1.5", CACM 21, 7, 565-569 (July 1978).

[2] Kurokawa T., "On the shallow binding of variables", J. of IPSJ, 20, 6, 524-526, (Nov. 1979).

[3] Bobrow, D. G., and Wegbreit, B., "A Model and Stack Implementation of Multiple Environments", CACM 16, 10, 591-603 (Oct. 1973).

[4] Steele, G. L. Jr., "Macaroni is Better than Spaghetti", Proc. of the Symposium on AI and Programming Languages, SIGPLAN Notices 12, 8, 60-66 (Aug. 1977).

[5] Minsky, M., "A Framewark for Representing Knowledge", The Psychology of Computer Vision, McGraw-Hill, 211-277 (ed. Winston, P. H.) (1975).

[6] Kurokawa T., "Function-class-its definition and application", Kigoshori WG of IPSJ, 1-8, (Feb. 1978).

[7] Kurokawa T., "Introduction of Function Class Method - Its definition and application -", Proc. of 13th Hawaii International Conference on System Science, (Jan. 1980).

[8] Sussman, G. J., et al., "Micro-Planner Reference Manual", AI Memo 203a, AI Lab. MIT (Dec. 1971).

[9] McDermott, D. V., Sussman, G. J., "The Conniver Reference Manual", AI Memo 259a (MIT) Updated (Jan. 1974)