Some Non-standard Issues on Lisp Standardization

Takayasu ITO * Taiichi YUASA †

1 Introduction

Lisp was born about 25 years ago as an AI language with a precise operational semantics. Since then many Lisp dialects have been proposed, implemented and used. In 1960's Lisp 1.5 was a kind of Lisp standard, although there were many Lisp 1.5 dialects which depend on I/O and computer systems. In 1970's various Lisp dialects were spawned to respond to the need of more powerful Lisp systems for AI research and symbolic computation. Among them we know that Lisp 1.6, Interlisp and Maclisp had big influences in the development of Lisp; especially, Maclisp brought us various interesting successors, including Franzlisp, Scheme, Zetalisp and Common Lisp. Common Lisp [STEELE] may be taken as a result of standardization activities of Maclisp and its successors, and Scheme is one of the first steps to try to settle and resolve some syntactic and semantic incompatibilities in Maclisp families. Eulisp [PADGET] may be taken to be another attempt along with this line with greater ambition for Lisp standardization, employing the design philosophy of extensible languages based on "leveling of languages". Various design and standardization issues have been considered and examined in the course of designing Common Lisp and Eulisp. The current standardization activities in US and Europe are mainly placed on "clean-up" of Common Lisp and "refinement and development" of Eulisp, in addition to efforts of designing object-oriented features in Lisp systems. In a sense major (standard) issues in Lisp standardization have been considered in these efforts of clean-up of Common Lisp and design of Eulisp. However, we think that there are some other important issues which may be called "non-standard issues" according to the current state of arts of Lisp systems.

Taking into account of Japanese activities on Lisp systems and their applications, we made the following comments on Lisp standardization at the ballot of NWI on ISO-Lisp.

> Common Lisp is a good starting point to design ISO-Lisp, but Common Lisp contains various technical deficiencies, as is pointed out

*Dept of Information Engineering, Tohoku University, Sendai, Japan
†Dept of Information and Computer Sciences, Toyohashi University of Technology, Toyohashi, Japan

(at least) in Japanese Lisp community. ISO-Lisp should be designed so as to resolve and remedy these technical problems. Moreover, in the course of designing ISO-Lisp, the following points should be examined

1) multi-process concepts in Lisp and its systems

2) interfaces to logic programming, object-oriented programming, UNIX, etc.

3) clear and simple formal syntax and semantics of the language

4) compactness and portability issues of systems

5) efficiency and implementation issues of the language

Some of these considerations (especially, 1), 2) and 3)) should be reflected in ISO-Lisp.

Our original comments contained the following item 6):

6) Japanese character set (including Kanji, Katakana and Hirakana) should be available in JIS-Lisp based on ISO-Lisp (JIS: Japan Industrial Standards)

which was not contained in our final comments, since the problem of Japanese character set in programming languages has been handled at the Japanese SC22 working group in more general settings.

Among the above items from 1) to 6) the items 1), 2) and 6) may belong to a class of non-standard issues in Lisp standardization. We discuss some of these non-standard issues in Lisp standardization on the basis of some Japanese experiences of Lisp systems and their applications.

2 Multi-process Concepts

Multi-programming and multi-processing have been popular in computer science, and Lisp systems are about to reside at the age of multi-process environments in hardware and software. Also, the next generation of computers will be the world of highly parallel machines based on VLSI, so that any language standardization activities should take into account of concurrency and parallelism.

TAO (developed at NTT) supports a small scale concurrent programming on NTT's Lisp machine ELIS [OKUNO]. There are several proposals of Parallel Lisp such as Multilisp [HALSTEAD], Q-Lisp [GABRIEL], PaiLisp [ITO], and *Lisp on Connection Machine [T.M.].

The concurrency and parallelism in Lisp systems should include the following aspects.

a) control of processes The continuation like "catch/throw" or "call/cc" will be a basic and necessary construct.

b) parallel evaluation of functions and processes

- par($e_1, ..., e_n$), which means the parallel evaluation of $e_1, ..., e_n$.
- future(e), which is the construct introduced by Halstead.
- delay(e), which means the delayed evaluation of e.

c) synchronization and communication mechanisms

- function-closure(e), which returns the exclusive function closure of e.
- signal(v) and wait(v), which are semaphore primitives assuming that v is a variable within a closure.

d) representation of "infinitary"

- loop(e), which means an infinitary evaluation of e.

We may have some other mechanisms like "parallel COND", "parallel AND/OR", and parallel list manipulation like "parallel MAPCAR".

It is premature to introduce these constructs in Lisp standard at present. But we feel that the use of parallel machines will become more popular in the near future than we think. If we can recommend some basic and good multi-process concepts in Lisp, such a recommendation will accelerate and direct the research on Parallel Lisp. Also it may be helpful to avoid some confusions in this area.

3 Logic Programming Interfaces

Prolog and Japanese FGCS project of ICOT have become a motive force for current active researches of logic programming. Logic programming is expected to possess nice and favorable properties in correct and specification-oriented programming and also to fit in non-determinism and parallelism.

We think that Prolog is premature as a standard logic programming language. Prolog is featured by unification and automatic backtracking. "Unification", which is a powerful mechanism spawned in logic programming, is worthwhile to be imported in Lisp standard, but "backtracking" is doubtful to be included in Lisp standard. But if we see the situation of Prolog programs and GHC/ESP programs in Japan, we may need more careful considerations, since some interesting Prolog programs have been developed around ICOT and many GHC/ESP programs are desirable to have some good interfaces between Lisp and logic programming languages. In this respect, LOGLISP and TAO are interesting attempts to amalgamate/fuse Lisp and logic programming. In order to import "unification" in Lisp, we must clear up the concept of unification in terms of objects to be unified. We may think of the following unifiable objects:

- variables
- list structures

- structured objects
- linear strings (string unification may be a research topic but we do not know any good application that requires string unification in nature.)

Boldly speaking, we may say that, if we can use "unification" in Lisp in a comfortable manner, we will not need a poor logic programming language. (Notice that we believe that we need a powerful logic programming language with favorable properties as mentioned above.)

Let us mention some technical aspects of Japanese logic programming systems which have close relationship with Lisp system.

Prolog/KR: Some of the Prolog systems were built on top of Lisp. Among them, the most widely used system in Japan is perhaps Prolog/KR, which was originally written in Utilisp, a dialect of Maclisp, and then rewritten in Maclisp, Zetalisp, and Common Lisp. In Prolog/KR, assertions are given in the form of S-expressions

(assert *head* . *body*)

where *head* and *body* are also in the form of S-expressions. The use of Lisp facilities such as Lisp reader greatly reduced the implementation costs. What is more important is that Prolog/KR has gained relatively high portability. We hope it will gain higher portability once an international Lisp standard is established. As Prolog/KR has proved, the mechanisms of Prolog, namely the unification could be realized efficiently on top of Lisp. However, if the Lisp system has its own unification mechanism built-in, applications of logic programming would run much more efficiently, without to say.

TAO and its unification: TAO attempts to embed unification mechanism to Lisp environment. TAO provides a special function == which unifies its two arguments. The result of unification is obtained by giving the == form as an argument to another special function GOAL-ALL. For example, the form

(goal-all (== _answer ,*form*))

first evaluates *form* (the comma indicates *form* should be evaluated first), unifies the logic variable _ANSWER (logic variables are prefixed with underscore) to the value of *form*, and then prints all logic variables that were instantiated during the evaluation of the == form, together with their instantiation values. The efficiency of TAO unification is based on the use of locative pointers. That is, when a logic variable is instantiated, the variable is given a locative pointer to the value. Thus, references to logic variables are nothing more than references to ordinary variables. Unfortunately, This implementation technique is hardly acceptable for Lisp systems on stock machines, because of the execution cost of the check whether a value cell has an actual value of the variable or a locative pointer. It is still to be discussed whether the unification mechanism in TAO could also be efficiently implemented in Lisp systems on stock machines.

4 OS Interfaces

Now a days, some Lisp machines are commercially available. For most of these machines, it is very difficult to discriminate the operating system from the Lisp proper, since the Lisp language can cover the facilities of the operating system for ordinary general-purpose machines. However, this fact does not mean we do not need the operating system interfaces in the Lisp standard. However widely Lisp machines become available and used, there are much more Lisp applications running on stock machines, under some operating systems. Naturally, users of such Lisp systems would like to have certain OS interfaces which guarantee the portability of his/her applications that make use of the underlying operating systems.

One solution to this problem is to add many Lisp functions each implementing a facility commonly found in most operating systems. It seems that Common Lisp aimed at this direction. This solution, however, is suffered from the fact that it is very difficult to determine a satisfactory set of OS facilities to be implemented by Lisp functions. Another solution is to define some very simple but highly useful interface functions. Unix has a powerful mechanism for interfacing an application program with another application and with facilities supplied by the OS. All programs, including applications and OS-supplied, can be connected with each other via the standard input and output. If we regard a Lisp system as a single application program, it is straightforward to connect the Lisp system with other applications by means of this mechanism. However, what most Lisp users expect is to invoke other facilities from within the Lisp system. To this end, a very simple-minded way would be to define a function that invokes applications by specifying the input/output stream to/from the applications, as well as arguments to the applications.

Since many Lisp users are also Unix users (at least in Japan), it seems worth considering how to interface Unix with the Lisp standard. Such an interface specific to a certain operating system may be very difficult to implement in those Lisp systems that run under other operating systems or that run on Lisp machines. If ever defined, such a standard interface should be an extension to the Lisp standard for those Lisp systems running under the particular operating system. A similar activity has already started for X-window interface from Common Lisp.

5 International Character Set Handling

5.1 What Should We Support?

In Japan, many Lisp applications need to handle Japanese text. For instance, expert systems for Japanese users are expected to communicate with the user in the Japanese language. Natural language recognition and translation systems are required to handle input and output in the Japanese language. Recently, some text formatters are written in Lisp and it is necessary for the underlying Lisp system to be capable of handling Japanese text in order to format Japanese

text. In addition to these needs from the Lisp applications, many Japanese Lisp programmers want to interact with Lisp systems in more natural way, i.e. in Japanese. For instance, they would like to give Japanese names to their variables and functions, and they would be more comfortable if the Lisp system speaks Japanese.

Among the features commonly found in many Lisp systems, the followings are expected to cope with programming in Japanese.

1. character objects
2. character strings, including format strings
3. symbol names
4. I/O
5. on-line documents
6. messages from the system
7. readtables and read macros

5.2 How Has It Been Handled?

Unlike the English language and its families, the Japanese language uses a large set of characters. It is said that more than 5000 characters are used in ordinary Japanese text. Therefore, it is very difficult to input Japanese characters directly using ordinary keyboard which has only 60 or so keys. The most popular way among computer programmers to input Japanese text is to use a front-end processor which receives the pronunciation of the text as its input, translates it into the corresponding complete written Japanese text, and sends the result to the application.

Several coding systems have been used to represent Japanese characters. These coding systems essentially use two bytes to represent each character, and the conversion from one coding system to another is very simple. Although most modern computer systems in Japan are capable of handling Japanese characters, they can also handle western character sets such as ASCII and EBCDIC. Japanese character coding systems allow both western characters and Japanese characters to appear in a single text. There are two methods to distinguish Japanese characters from westerns. One is to represent Japanese characters with those bytes that are not used for western characters. For example, since the ASCII encoding uses seven bits and the most significant bit is always 0, some Japanese coding systems use only those bytes whose most significant bits are 1 to represent Japanese characters. The other method is to surround a sequence of Japanese characters with a certain "escaping" code. The JIS coding system, which is the JIS extension to the ISO coding system, uses this method. With this method, a text can be represented with only those bytes whose most significant bits are 0. Therefore, such a text can be handled by non-Japanese computer systems as well. With this method, however, language processors

which are originally developed for western coding systems must be drastically modified in order to handle Japanese input, because the character code for a special character such as parenthesis may be a part of the representation of a Japanese character. Thus, the tendency is that modern Japanese computer systems employ those coding systems with the former method. An example of such coding systems is UJIS, which is the *de facto* standard coding system for Unix machines.

5.3 The Current Situation

On those computer systems that use such coding systems like UJIS, it is possible for a Lisp system to cope with Japanese programming to some extent, without any modification of the system. The system can handle Japanese strings and symbols with Japanese names. Ordinary readtables can be used for Japanese programming if all Japanese characters are supposed to be constituent (in terms of Common Lisp). Of course, on-line documents and system messages must be translated into Japanese, but this is a simple work. However, there still remain two major problems in order for the system to cope with Japanese programming. One is that the number of characters in a string is not always the same as the number of bytes in the string, and the n-th byte in the string does not always correspond to the n-th character. This means that string-handling functions such as LENGTH and ELT in Common Lisp need to be modified if they are to handle Japanese strings "correctly". The other problem is that ordinary readtables only for western characters are not always sufficient. Japanese character coding systems include those characters that correspond to western characters. As a result, it is usually the case that a single western character has both the single-byte representation and the two-byte representation. It is quite natural that the programmer expects the Lisp system to treat, say, the opening parenthesis represented with two bytes in the same way as the opening parenthesis represented with a single byte. Fortunately, when printed, a single-byte western character can be easily discriminated from the corresponding double-byte character, since a double-byte character occupies twice as large space as a single western character. The only exception is the space character, which, without to say, plays a very important role in Lisp programs. A double-byte space looks exactly the same as two consecutive single-byte spaces.

5.4 On International Standard

The above problems in Programming in Japanese are not only for the Japanese language but also for those languages which use large character sets. Providing a common basis for treating these languages is highly expected in the international Lisp standard. The expected proposal should contain some mechanism that allows a single Lisp system to cope with multiple languages without major (or hopefully no) modification to the system.

The IBM proposal [Linden] for international character set handling is a good candidate for this purpose. One of the key features of the proposal is the notion

of equivalence classes among the character objects. It clearly solves one of the above problems, by allowing the user to borrow the syntactic attribute of a standard character for non-standard characters specific to his/her language.

6 Some General Remarks

6.1 On Kernel of Common Lisp

There are some controversies between Common Lisp standardization and Eulisp activity. The leveling approach taken by Eulisp is nicer than a simple-minded Common Lisp standardization. However, Eulisp lacks in practical experiences. One possible solution is to extract a kernel of Common Lisp and to give a formal operational semantics for such a Lisp kernel in the spirit of Scheme and Eulisp. Such a kernel should contain some constructs which support "multi-process", "unification", and some other basic features of Lisp standard.

6.2 On Formal Semantics of Lisp

According to our understanding, most people in Lisp standardization are interested in having a formal and clear semantics of Lisp. We think that denotational semantics of Lisp will be no good when we think about semantics of Lisp with advanced features such as object-oriented programming, infinite streams, and concurrency. The best way will be to have a formal operational semantics for Lisp. The mixture of operational semantics [PLOTKIN], natural semantics [KAHN], and action semantics [MOSES] may give us a natural way of defining operational semantics of Lisp.

6.3 On Object-oriented mechanism in Lisp

CLOS [BOBROW] is proposed as a standard object-oriented system for Common Lisp. Most of object-oriented programming languages are featured by

- message passing and method with interface
- class and inheritance

According to our understanding, CLOS is unusual in its message passing, or CLOS lacks in message passing at all. Any object-oriented language is a kind of language for modeling and simulation. Most computer systems will be concurrent systems in nature, so that partial ordering structures among class objects are very natural in modeling. The inheritance mechanism in CLOS is quite elegant on a sequential machine but may not be so on a parallel machines.

6.4 On Performance Issues

Japanese Lisp activities have been activated and accelerated by the 1st, 2nd, and 3rd Lisp contests of Information Processing Society of Japan by the sets of

Lisp benchmark programs. In the course of standardization activities we should think about

- Lisp benchmark programs for performance evaluation
- Lisp test sets for validation of Lisp systems.

6.5 Remarks from Common Lisp experiences

Although Common Lisp is intended to be an international standard and the language specification in [STEELE] is relatively rigorous compared with conventional Lisp manuals, there still remains portability problem of Common Lisp applications. The portability of applications, which should be one of the most important issues in standardization activities for any programming language, will be guaranteed once a formal specification of the Lisp standard is established in the way described above. Until it is established, however, we have to make use of currently available technologies. From our experiences on Common Lisp, especially on implementation of Kyoto Common Lisp [YUASA] of which one of the authors has designed and implemented, we expect the following issues to be taken into standardization considerations.

1. The informal specification of the language should not assume "common sense". Sentences like "perform *normal* compiler processing" do not make sense unless the specification explicitly mention what "normal" means.

2. Formal (or at least clear) syntax should be provided. BNF and its variants are not always appropriate for describing Lisp syntax as will be clear if we think, say, the syntax of Common Lisp DEFMACRO lambda lists.

3. The language standard should provide a means to distinguish implementation-specific features from the standard. The notions of packages and keyword parameters are great inventions of Common Lisp for this purpose. Yet there still remain problems such as implementation-specific syntactic extensions for macros are not clear to the programmer.

4. It is highly expected that a highly portable implementation be supplied as part of the standard. The easiest way for a Lisp system implementor to check unclear language features is perhaps to see the behavior of other reliable implementations. In order to make the standard to be widely available on different computer systems, we expect a "standard" implementation of the language, which may not be so efficient but has extremely high portability and reliability hopefully with no its own extensions.

REFERENCES

[**STEELE**] Guy L. Steele Jr., Common Lisp the Language, Digital Press, 1984.

[**PADGET**] Julian Padget et al, Desiderata for the standardisation of LISP, ACM Symposium on Lisp and Functional Programming, 1986.

[**HALSTEAD**] Robert H. Halstead Jr., Implementation of Multilisp: Lisp on a Multiprocessor, ACM Symposium on Lisp and Functional Programming, 1984.

[**GABRIEL**] Richard P. Gabriel and John McCarthy, Queue-based Multiprocessing Lisp, ACM Symposium on Lisp and Functional Programming, 1984.

[**ITO**] Takayasu Ito et al, An MC68000-based Multi-micro Processor System with Shared Memory and Its Application to Parallel Lisp Interpreter, JIP Computer System Symposium, 1987.

[**T.M.**] Thinking Machines Corporation, Introduction to Data Level Parallelism, Technical Report 86.14, 1986.

[**LINDEN**] Thom Linden, Common LISP - Proposed Extensions for International Character Set Handling, Version 01.11.87, 1987.

[**OKUNO**] Hiroshi G. Okuno et al, TAO: A Fast Interpreter-Centered Lisp System on Lisp Machine ELIS, ACM Symposium on Lisp and Functional Programming, 1984.

[**NAKASHIMA**] Hideyuki Nakashima, Prolog/KR – Language Features, International Logic Programming Conference, 1982.

[**PLOTKIN**] G. D. Plotkin, A Structural Approach to Operational Semantics, DAIMI FN-19, Aarhus University, 1981.

[**KAHN**] Gilles Kahn, Natural Semantics, INRIA Rapports de Recherche No. 601, 1987.

[**MOSES**] Peter D. Moses, A Basic Abstract Semantic Algebra, International Symposium on Semantics of Data Types, Springer LNCS 173, 1984.

[**BOBROW**] Daniel G. Bobrow et al., Common Lisp Object System Specification, Draft X3 Document 87-002 and 003, 1987.

[**YUASA**] Taiichi Yuasa and Masami Hagiya, Kyoto Common Lisp Report, Teikoku Insatsu Publishing, 1985.