数理解析研究所講究録
第 436 巻 1981 年 218-232

Design, implementation and philosophy of Hyperlisp

By Masami Hagiya
(Univ. of Tokyo)

introduction

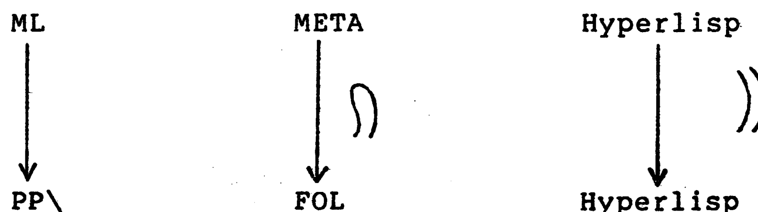
Hyperlisp (in its narrow sense) is a Lisp-like programming language designed by Masahiko Sato, whose features (from the theoretical point of view) are:

* Its domain of symbolic expressions is mathematically neater than that of Lisp.
* It is a completely monotype language.
* Its semantics is defined in a precisely constructive (or operational) manner.

But many who have heard about Hyperlisp, especially those who are actually using Lisp for practical purposes, say, `Hmmmm, it is beautiful, elegant. But what are you going to do with it?' or more directly, `What on earth is the aim of Hyperlisp?' Here in this short paper, we try to answer such questions but it seems quite difficult...

One of the principle motivations for the design of Hyperlisp was to construct a verification system for (or about) Hyperlisp by Hyperlisp itself. It may be worth comparing our approach with

that of, say, LCF[4] or FOL[10].



In LCF, the object language (called PP\) and the meta language (called ML) are entirely different languages, while in FOL, the meta language (called META) is one of the theories of FOL. What we are planning to do resembles the latter, but our approach is much simpler: we are just going to implement Gödel's Incompleteness Theorem! (says Sato.)

The merits of this approach are:

* Only one language is required: towards a universal language.
* We can easily construct arbitrarily high level theories: metatheory, metametatheory, ...

It seems that the technique of embedding into a theory its metatheory is now considered to be important in order to construct extensible theorem provers, e.g. Boyer and Moore's[1] or Brown's[2].

Moreover we believe that all the arguments including metatheoretical ones should be carried out in a constructive (or finitary in the sense of Hilbert) manner, because the constructive method is the most fundamental in finite mathematics and the theory of computation may be studied as a part of finite mathematics. It is also practically important, since it requires

no highly mathematical notions such as c.p.o. or inductive limit.

At present we have designed and implemented the programming language Hyperlisp, and are now constructing the theory of Hyperlisp, which, unfortunately, we cannot present in this paper. In the following we give an overview of the language and its implementation without going into details. We refer the interested readers to the references[8, 9].

Gödel numbering

Recall Gödel's Incompleteness Theorem[3]. In his proof, he encoded terms, formulas etc. by natural numbers with the method called Godel numbering or coding. This makes it possible to express a metatheorem about the natural number theory as a theorem about natural numbers. But the coding he used was a theoretical or conceptual one; it is almost impossible to implement a prover or a proof checker on a computer using his coding directly. This is due to the fact that N (the set of all the natural numbers) is too simple as a data structre.

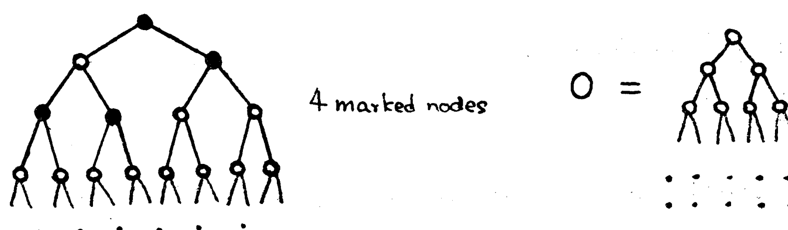
sexp

We now search a data structure which is richer than N. The first candidate is of course that of Symbolic expressions of Lisp. But it has a problem: the set of atoms may be any set having any structure. It means that the domain of Lisp Symbolic expressions is not a fixed data structure (or type) but rather a type having a type parameter (which corresponds to the atoms).

In other words, lists and atoms have completely different structures; e.g. we cannot take the car of an atom.

Taking these into account, Sato defined the symbolic expression of Hyperlisp as follows.

A symbolic expression (or sexp for short) is an infinite leaf-free binary tree with a finite number of its nodes marked (black). See the left figure.



We assume that those nodes outside the drawings are not marked. Since only a finite number of nodes are marked in a sexp, it is in fact a finite figure. The tree with no marked nodes is denoted by Ø. This plays a similar role as that of the natural number Ø.

We have two selectors as in Lisp i.e. car and cdr; car selects the left subtree and cdr selects the right subtree. In Hyperlisp, car and cdr are total functions. Note that car(Ø) = cdr(Ø) = Ø.

We have two constructors cons and snoc. (`snoc' is the reverse of `cons'.) They are defined as follows:

cons(Ø, Ø) = Ø.

Finally we define the recognizer called atom:

atom(x) <=> the root of x is marked

The primitives of sexps are: car, cdr, cons, snoc, atom and the equality between sexps.

The set of all the sexps is denoted by S.

We set

A = { x | atom(x) }

M = S - A

We call an element of A an atom, and an element of M a molecule.

Since cons: S * S -> M and snoc: S * S -> A are bijective, we have the following set theoretic isomorphisms:

S = A + M

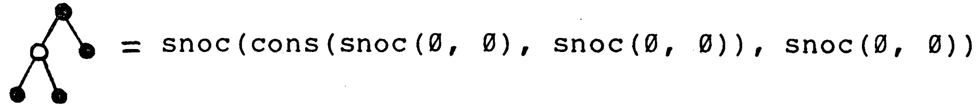
= A + S * S

= S * S + S * S

(* denotes the cartesian product and + denotes the direct sum) The second equation is an evidence that S is at least as rich as the domain of Lisp, since in Lisp, we have S = A + S * S, where S is the set of Lisp Symbolic expressions and A is the set of atoms. (For this resemblence, we use the terminology `atom'.) But the difference is that in Hyperlisp we also have A = S * S; this means that we can take the car of an atom.

Obviously, every sexp can be constructed from Ø by a finite

number of applications of cons and snoc; e.g.



From this, we have the folowing induction schema:

$$\frac{A(\emptyset) \qquad A(x)\ \&\ A(y) \rightarrow A(cons(x, y))\ \&\ A(snoc(x, y))}{A(t)}$$

It will be utilized when we axiomatize S.

<u>notation</u>

We use two pairs of parentheses: (), [].

Dot notation:

(x . y) = cons(x, y)

[x . y] = snoc(x, y)

List notaton:

(x, y, z) = (x . (y . (z . Ø)))

[x, y, z] = [x . [y . [z . Ø]]]

We cannot omit , in Hyperlisp.

<u>natural numbers and literals</u>

Because of the richness of the structure of S, we can easily implement many important data structures in S. Take N as an example.

We define r: N -> S as

r(Ø) = Ø   r(n+1) = snoc(r(n), r(n))

By r, N is embedded in S. We regard N as a subset of S. E.g. 1 = [Ø . Ø].

Another important data structure is that of what we call literals (which are strings of lowercases). In the present version, we implement literals as follows:

"ab" = [[1, 1, Ø, Ø, Ø, Ø, 1], [1, 1, Ø, Ø, Ø, 1, Ø]]

where a is ascii 141 in octal i.e. 11ØØØ1 in binary.

Remember that natural numbers and literals are atoms.

<u>evaluator</u>

Hyperlisp is a programming language to do the computation on S. In Hyperlisp, like in Lisp, a program or a function is represented by a sexp. Then mathematically the semantics of Hyperlisp is given by the partial (recursive) mapping eval; S -> S; eval(x) = z means that x is evaluated to z. Here we give the definition of eval in Algol-like notaton.

```
eval(x)
= if atom(x) then apply(car(x), cdr(x))
             else apply(car(x), evlis(cdr(x))) fi

evlis(x)
= if x = Ø then Ø
```

```
              else cons(eval(car(x)), evlis(cdr(x))) fi

    apply(f, x)
    = if f = Ø then Ø
      elif atom(f) then
            if f = 1 then car(x)
            elif f = "eq" then
                    if car(x) = car(cdr(x)) then 1 else Ø fi
            elif f = "cond" then evcon(x)
(*)         elif f is defined then apply(the definition of f, x)
            else apply(eval(f), x) fi
      else
            if car(f) = "lambda" then
                    eval(subst(x, car(cdr(f)), car(cdr(cdr(f)))))
            elif car(f) = "label" then
                    apply(subst(f, car(cdr(f)), car(cdr(cdr(f)))), x)
            else apply(eval(f), x) fi fi

    evcon(x)
    = if x = Ø then Ø
      elif atom(eval(car(car(x)))) then eval(car(cdr(car(x))))
      else evcon(cdr(x)) fi

    subst(x, p, b)
    = if p = Ø then b
      elif atom(p) then point(x, car(p))
      elif atom(b) then
            snoc(subst(x, car(p), car(b)), subst(x, cdr(p), cdr(b)))
      else  cons(subst(x, car(p), car(b)), subst(x, cdr(p), cdr(b))) fi

    point(x, q)
    = if q = Ø then Ø
      elif atom(q) then x
      elif cdr(q) = Ø then point(car(x), car(q))
                     else point(cdr(x), cdr(q)) fi
```

Literals are double-quoted. If we omit the line marked (*), we will have `Pure Hyperlisp'.

<u>reference language</u>

Hyperlisp functions or data (they are all sexps) are written in what we call the reference language of Hyperlisp, which is an extension of dot and list notation. Natural numbers and literals are of course permitted. Literals are not double-quoted in the refernece language. (From now we will use <u>x</u>, <u>y</u> etc. for

variables to avoid confusion.)

There are some abbreviations in the reference language:

$\underline{x} : \underline{y} = (\underline{x}, \underline{y})$

$\underline{f}(\ldots) = (\underline{f}, \ldots)$ $\underline{f}[\ldots] = [\underline{f}, \ldots]$

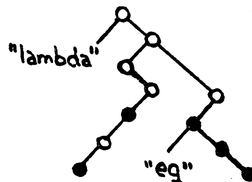
$'\underline{x} = 1[\underline{x}] = [1, \underline{x}]$

The first one is intended to be used in a conditional expression as a conditional pair. The second one is used to write function applications in a prefix way. The last one is for quotation; $'\underline{x}$ is always is evaluated to $\underline{x}$. These notations make Hyperlisp programs resemble Lisp programs written in meta-expressions.

For lambda and label expressions, special syntax is prepared. We just give an example:

Lambda([X]; eq[X, Ø])

where Lambda is the keyword for introducing the syntax and X is a formal parameter, which we call a metaliteral. A metaliteral is a string beginning with an uppercase. The above expression denotes the following sexp:



The expression Lambda([Y]; eq[Y. Ø]) also denotes the same sexp. Here we do not give the presice rule by which these expressions are translated to sexps. Just understand that this is a very complicated read macro like the back-quote macro of MacLisp or other Lisps.

Function definitions are written as follows:

```
#null[X] = eq[X, Ø]
```

By this, the above lambda expression is assigned to the literal null.

remarks

$\underline{x}$ > $\underline{z}$ denotes eval($\underline{x}$) = $\underline{z}$ in this section.

Look at the definition of eval. When the sexp to be evaluated is an atom, its car (i.e. function part) and its cdr(i.e. argument list) are sent to apply just as they are, while when it is a molecule, its argument list is argument-wise evaluated by evlis and then sent to apply. Namely, the first case is call-by-name, and the latter case is call-by-value.

We have only three primitive functions in the evaluator: l, eq and cond. (eq and cond are literals.) l is the identity function: l[$\underline{x}$] > $\underline{x}$ i.e. '$\underline{x}$ > $\underline{x}$. eq represents equality.

$\underline{x}$ = $\underline{y}$ => eq[$\underline{x}$, $\underline{y}$] > 1

$\underline{x}$ ≠ $\underline{y}$ => eq[$\underline{x}$, $\underline{y}$] > Ø

Note that in Hyperlisp, every atom represents truth, and evary molecule represents falsity. cond is for the conditional expression, which resembles that of Lisp very much. Is it of the form:

```
cond[ (..., ...),
      (..., ...),
        ...
```

```
(..., ...) ]
```

We may also write

```
cond[ ... : ... ;
      ... : ... ;
         ...
      ... : ... ]
```

; may be used in the place of ,.

The parameter binding of Hyperlisp is very much different from that of Lisp; in Hyperlisp, the arguments are actually substituted in the function body. As an example let's define cons as

```
#cons[X, Y] = '(X . Y)
```

Now evaluate cons[a, b]. First, the first argument a and the second argument b are (actually) substituted to X and Y in the body '(X . Y) to yield '(a . b). Next '(a . b) is evaluated to yield the final value (a . b); i.e. cons[a, b] > (a . b). So we can define cons in Hyperlisp. In MacLisp, using the back-quote macro, we can do the same thing:

```
(DEFUN CONS (LAMBDA (X Y)
  `(,X . ,Y) )
```

We can also define car and cdr in Hyperlisp. The definitions are:

```
#car[X = [X1 . X2]] = 'X1
#cdr[X = [X1 . X2]] = 'X2
```

where X is the first parameter and X1 is the car of X and X2 is the cdr of X. We can name any part of the argument list by such declarations.

As an exercise, please try the following evaluations:

```
car[(a . b)]  >  a
cdr[(a . b)]  >  b
cons(car[(a . b)], 'c)  >  (a . c)
```

Finally we define the familiar append function as follows.

```
#append[X = (X1 . X2), Y]
= cond[ null[X] : 'Y;
        '1 : cons('X1, append[X2, Y]) ];
```

summary

Here we summarize the features of Hyperlisp from the practical point of view.

* Programs may be written in a style which is similar to that of meta-expressions of Lisp.
* The scope of a variable is static.
* No prog feature is prepared.
* There is no distinction between expr and fexpr; any function may become expr and fexpr depending on how it is called.
* Any part of the parameter (e.g. its car, cadr, caddr, cadar etc.) may be named in a function.

* It has a feature similar to the back-quote macro of Lisp; this is called Quine's quasi-quotation.

The last two save many of the explicit uses of selectors and constructors; e.g. compare the two definitions of the naive reverse:

```
#reverse[X = (X1 . X2)]
= cond[ null[X] : Ø;
        '1 : append(reverse[X2], '(X1)) ];

(DEFUN REVERSE (LAMBDA (X)
  (COND ((NULL X) NIL)
        (T (APPEND (REVERSE (CDR X)) (CONS X NIL))) )))
```

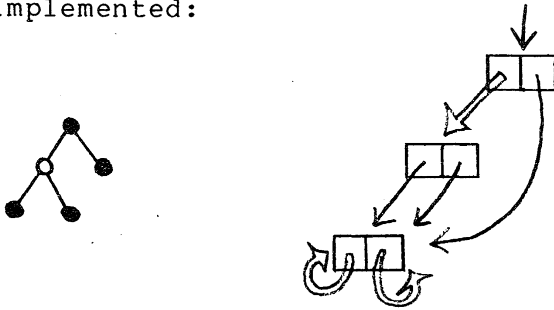
implementation

The first interpreter[6] is implemented on PDP11 under UNIX[7]. A pointer is a PDP11 word, i.e. consists of 16 bits. There are 8K cells. The cell space is divided into H space and L space (these are the terminology of HLISP[5]). S is implemented in H space i.e. it is monocopied. The reasons for this are:

* Each atom may have a function definition.
* Literals should be put out as they are put in.

The size of H space and L space are the same. L space is used for improving the efficiency of the interpreter. (We avoid the actual substitution of the arguments in the function body as much as possible in the implementation; for this, a kind of special technique is devised, for which L cells are used.) We sketch how

a sexp is implemented:



Recently, we implemented the interpreter on VAX11 under UNIX, in which case, a pointer is 32 bit long. The cell space consists of 32K cells now. 20 subrs are prepared.

In the present version (for both PDP11 and VAX11), there are two global environments; one is for function definitions, and the other is for global values of atoms. The latter resembles the property list of Lisp, because the values are extracted and updated through explicit calls of subrs. Since the scope of Hyperlisp is static, this is indispensible for writing large programs. (We are also planning to introduce progs.)

references

[1] Boyer, R. S., Moore, J. S.: Metafunctions: Proving Them Correct and using Them Efficiently as New Proof Procedures, Technical Report CSL-108, SRI International (1979)

[2] Brown, F. M.: An Investigation into the Goals of Research in Automated Theorem Proving as Related to Mathematical Reasoning, Artificial Intelligence 14, 221-242 (1981)

[3] Gödel, K.: Über formal unentscheidbare Sätze der Principia mathematica und verwandter Systeme I, Monatshefte für Mathematik und Physik 38, 173-198 (1931)

[4] Gordon, M., Milner, R., Wadsworth, C.: Edinburgh LCF, Lecture Notes in Computer Science 78, Springer-Verlag (1979)

[5] Goto, E.: Monocopy and Associative Algorithms in an Extended Lisp, TR74-03, Information Science Laboratories, Faculty of Science, University of Tokyo (1974)

[6] Hagiya, M: Hyperlisp2.1 Manual (not published)

[7] Kernighan, B. W., McIlroy, M. D.: Unix Programmer's Manual, Seventh Edition, Virtual VAX-11 Version (1979)

[8] Sato, M.: Theory of Symbolic Expressions, TR80-16, Department of Information Science, Faculty of Science, University of Tokyo (1980)

[9] Sato, M., Hagiya, M.: Hyperlisp (to appear)

[10] Weyhrauch, R. W.: Prolegomena to a Theory of Mechanized Formal Reasoning, Artificial Intelligence 13, 133-170 (1980)