QUTE: A PROLOG/LISP TYPE LANGUAGE FOR LOGIC PROGRAMMING

Masahiko Sato
Takafumi Sakurai

Department of Information Science, Faculty of Science
University of Tokyo
7-3-1 Hongo, Bunkyo-ku, Tokyo 113, JAPAN

ABSTRACT

A new Prolog/Lisp type programming language called Qute is introduced. Qute computes (partial) recursive functions on the domain S of symbolic expressions in the sense of Sato[3], Sato and Hagiya[4].

Qute amalgamates Prolog and Lisp in a natural way. Any expression that is meaningful to Qute is either a Prolog expression or a Lisp expression and a Prolog (Lisp) expression is handled by the Prolog (Lisp, resp.) part of Qute. Moreover, the Prolog-part and the Lisp-part calls each other recursively.

Compared with the traditional Lisp symbolic expressions, our symbolic expressions are mathematically much neater and yet constitute a richer domain. Qute is a theoretically well-founded language defined on this domain of symbolic expressions.

Many interesting features of Qute are described in this paper.

Qute has been implemented on VAX/UNIX and is used to develop a programming system for proving properties of our domain of symbolic expressions.

0. Introduction

In this paper, we introduce a new Prolog/Lisp type programming language called Qute that is designed to compute (partial) recursive functions on the domain S of symbolic expressions in the sense of Sato [3], Sato and Hagiya[4].

Since Qute combines the features of Prolog and Lisp quite naturally, it provides a comfortable environment for developing programs interactively. Users of Qute can not only enjoy both Prolog and Lisp style programming but also combine them in a unique way.

To be more precise, any expression that is meaningful to Qute is either a Prolog expression or a Lisp expression. A Prolog expression may contain Lisp expressions as its subexpressions and conversely a Lisp expression may contain Prolog expressions as its subexpressions. Prolog (Lisp) expression is handled by the Prolog (Lisp, resp.) part of Qute. In this way, the Prolog-part naturally contains the Lisp-part and the Lisp-part contains the Prolog-part.

This paper is based on the result of activities of working groups for the Fifth Generation Computer Systems Projects.

Another characteristic feature of Qute is that, like Lisp but unlike Prolog, symbolic expressions play the double role of data and programs. It is therefore possible to write a simple metacircular interpreter of Qute by Qute itself. In fact, we can write the interpreter using only the Prolog-part of Qute. The interpreter of Qute can be defined formally by inductive definitions as we did for Hyperlisp[3]. This makes Qute a theoretically well-founded language. In this paper, however, due partly to the limitation of space, we will describe the semantics of Qute rather informally.

LOGLISP[l] takes a similar approach towards combination of Prolog and Lisp, but our concern centers on formalism which we slightly mentioned above. Therefore, we designed Qute so that programs can be naturally regarded as symbolic expressions. (As is explained later, we regard a 'variable' as a symbolic expression unlike usual Prolog.)

Qute has been implemented on VAX/UNIX at the Computer Centre of the University of Tokyo. The language is used to develop a programming system for proving properties of our domain of symbolic expressions. Properties of Qute will be expressed and verified in the system. See Sakurai [2] for more details of the project.

In the rest of the paper, we first review our domain S briefly and then describe the syntax and semantics of Qute. Many interesting features of the language will be described along the way.

1. Symbolic Expression

1.1. definition of sexp

Symbolic expressions *(sexps,* for short) are constructed by the following clauses:

1. 0 is a sexp.
2. If s and *t* are sexps then *snoc(s, t)* is a sexp.
3. If s and *t* are sexps and at least one of them is not 0 then *cons(s, t)* is a sexp.

All the sexps are constructed only by means of the iterated applications of the above three clauses, and sexps constructed differently are distinct. We denote the set of all the sexps by S. Note that *snoc* is a total function on SxS while *cons* is partial since it is undefined for the argument (O, *0).* We make *cons* total by stipulating that *cons(0, 0)* - 0. We also put *snoc(0, 0)* - 1.

We denote the image of the function *cons* by **M** and that of *snoc* by **A**, so that we have two bijective functions:

$$cons: \mathbf{S}\times\mathbf{S}\rightarrow\mathbf{M}$$
$$snoc: \mathbf{S}\times\mathbf{S}\rightarrow\mathbf{A}$$

Moreover, we have $\mathbf{S} = \mathbf{M}\cup\mathbf{A}$ and $\mathbf{M}\cap\mathbf{A} = \phi$; i.e., **S** satisfies the domain equation

$$\mathbf{S} = \mathbf{S}\times\mathbf{S} + \mathbf{S}\times\mathbf{S}$$

Elements in **M** are called *molecules* and those in **A** are called *atoms*. By the above discussion, we can define total functions *car* and *cdr* on **S** by the equations

$$car(cons(x,y)) = car(snoc(x,y)) = x$$
$$cdr(cons(x,y)) = cdr(snoc(x,y)) = y$$

1.2. dot notation and list notation

We introduce dot notation and list notation as notations for sexps. A sexp is also called a *list* when it is written in dot notation or list notation.

$$[x\ .\ y] = snoc(x,y)$$
$$(x\ .\ y) = cons(x,y)$$
$$[.\ x] = x$$
$$[x_1,\ \cdots,\ x_n\ .\ x_{n+1}] = [x_1\ .\ [x_2,\ \cdots,\ x_n\ .\ x_{n+1}]]$$
$$[x_1,\ \cdots,\ x_n] = [x_1,\ \cdots,\ x_n\ .\ 0]$$
$$(.\ x) = x$$
$$(x_1,\ \cdots,\ x_n\ .\ x_{n+1}) = (x_1\ .\ (x_2,\ \cdots,\ x_n\ .\ x_{n+1}))$$
$$(x_1,\ \cdots,\ x_n) = (x_1,\ \cdots,\ x_n\ .\ 0)$$
$$[] = () = 0$$

A list which begins with (is a *cons list* and a list which begins with [is a *snoc list*. (. *x*) is a cons list and [. *x*] is a snoc list, though they denote the same sexp.

1.3. name

Let **L** be the set of ASCII graphic characters. We define a function $\rho : \mathbf{L}\rightarrow\mathbf{A}$ by using 7 bit ASCII codes. We explain by examples.

$$\rho(a) = [1, 1, 0, 0, 0, 0, 1]$$
$$\rho(A) = [1, 0, 0, 0, 0, 0, 1]$$
$$\rho(1) = [0, 1, 1, 0, 0, 0, 1]$$
$$\rho(*) = [0, 1, 0, 1, 0, 1, 0]$$

Note that the ASCII code of 'a' is 1100001 in binary.

A *name* is a string of alphanumeric characters whose length is longer than 1 and which begins with a lowercase.

A name denotes a sexp as follows:
Let $n = l_1 \cdots l_k$ be a name, then n denotes

$$[\rho(l_1),\ \cdots,\ \rho(l_k)]$$

2. The Lisp-part of Qute

The top level read routine of Qute reads in an expression and processes it. An expression is either a Lisp expression or a Prolog expression. A Lisp (Prolog) expression will be processed by the Lisp (Prolog, resp.) part of Qute. We will explain the Lisp-part of Qute in this section. Evaluation mechanism of Lisp expressions in Qute mostly follows that of usual Lisp. However, our treatment of variables and function applications radically differs from usual Lisp.

We give some examples in 2.1, give informal explanation of *eval* in 2.2-2.6 and summarize definition of *eval* in 2.7.

2.1. examples

```
(L1)  0;
      = 0
(L2)  [apple, orange];
      = [apple, orange]
(L3)  > cons(X, Y) = (X . Y);
      cons defined
(L4)  cons(apple, orange);
      = (apple . orange)
(L5)  > snoc(X, Y) = [X . Y];
      snoc defined
(L6)  '[X, cons(apple, orange)];
      = [X, cons(apple, orange)]
(L7)  `[cons(left, right), X, \/Y, '/snoc(left, right)];
      = [cons(left, right), X, /Y, '[left . right]]
(L8)  > car((X . Y)) = X;
      car defined
(L9)  > cdr((X . Y)) = Y;
      cdr defined
(L10) car(snoc(left, right));
      = left
(L11) > atom(Z = (X . Y)) = eq(Z, [X . Y]);
      atom defined
(L12) eq(apple, orange);
      = 1
(L13) eq(apple, apple);
      = 0
(L14) atom(snoc(apple, orange));
      = 0
(L15) > append(X = (X1 . X2), Y)
      = Cond[ eq(X, 0) -> Y,
              0 -> [X1 . append(X2, Y)] ];
      append defined
(L16) append([aa, bb], [cc, dd, ee]);
      = [aa, bb, cc, dd, ee]
(L17) > apply(F, X) := ('APPLY, [F . X]);
      apply defined
(L18) apply(cons, [lisp, prolog]);
      (lisp . prolog)
(L19) > and(. x = (x1 . x2))
      := `Cond[ eq('/x, 0) -> 0,
                /x1 -> And[. /x2]];
      and defined
(L20) And[eq(aa, aa), 0];
      = 0
```

2.2. constant and special sexp

A molecule whose car-part is

[[0, 1, 0, 1, 0, 1, 0]],	(= [ρ(*)])
[[1, 0, 1, 1, 1, 1, 0]],	(= [ρ(^)])
[[0, 1, 0, 0, 1, 1, 1]],	(= [ρ(')])
[[1, 1, 0, 0, 0, 0, 0]],	(= [ρ(`)])
[[1, 0, 1, 1, 1, 0, 0]] or	(= [ρ(\)])
[[0, 1, 0, 1, 1, 1, 1]]	(= [ρ(/)])

is called a *special sexp.* We use VAR, QUOTE, QQUOTE, ESC, EVAL respectively to denote the above atoms. A sexp which does not contain a special sexp as its sub-sexp is called a *constant sexp* or simply a *constant.*

We now explain the function *eval* that is used to evaluate Lisp expressions. The function *eval* is defined so that it preserves *cons* and *snoc* for non-special sexps and hence it becomes an identity function on constants. Therefore, we can make use of *snoc* (and *cons* which satisfies (r3)) as a pattern constructor. This advantage comes from the fact that we have two constructors *snoc* and *cons.*

(*r*1) *eval*(0) = 0
(*r*2) *eval*([*x* . *y*]) = [*eval*(*x*) . *eval*(*y*)]
(*r*3) *eval*((*x* . *y*)) = (*eval*(*x*) . *eval*(*y*))
where $x \neq$ **VAR, APPLY, QUOTE, QQUOTE**

The expressions (L1), (L2) in 2.1 are evaluated by these rules. Note that a name is a constant.

We explain evaluation rules for special sexps in the following.

2.3. variable and environment

A special sexp (**VAR** . *x*) is called a *variable*. We introduce a syntax sugaring for a variable. A single lowercase character followed by a string of digits or a nonempty string of alphanumeric characters which begins with an uppercase character denotes a variable. Let the string be $l_1 \cdots l_n$. It denotes

$$(\mathbf{VAR}\ .\ [\rho(l_1),\ \cdots,\ \rho(l_n)])$$

Example 2.1.

Var = ([[0,1,0,1,0,1,0]] .
[[1,0,1,0,1,1,0],
[1,1,0,0,0,0,1], [1,1,1,0,0,1,0]]) □

The value of a variable is determined relative to an *environment*. An environment is a list of pairs of a variable and its value. It is created when a function or a macro is called.

Besides this environment, there is a global environment, though we do not go into details in this paper. If a variable is not found in an environment, a global environment is searched. A global environment is preserved even after evaluation.

2.4. quote and quasi-quote

For a sexp *t*, each of ´*t*, \`*t*, \\*t* and /*t* denotes a special sexp (**QUOTE** . *t*), (**QQUOTE** . *t*), (**ESC** . *t*) and (**EVAL** . *t*) respectively. We say *t* is in the *scope* of ´, \`, \\ and / respectively. ´ and \` plays a similar role in *eval* as that in usual Lisp. ´ plays the role of quote.

(*r*4) *eval*(´*t*) = *t*

In the scope of ´, a special sexp loses its special meaning. \` plays the role of backquote in Maclisp, but our \` is not a read-macro.

(*r*5) *eval*(\`*t*) = *qeval*(*t*)
(*r*6) *qeval*(0) = 0
(*r*7) *qeval*([*x* . *y*]) = [*qeval*(*x*) . *qeval*(*y*)]
(*r*8) *qeval*((*x* . *y*)) = (*qeval*(*x*) . *qeval*(*y*))
where $x \neq$ **ESC, EVAL**
(*r*9) *qeval*(*t*) = *t*
(*r*10) *qeval*(/*t*) = *eval*(*t*)

In the scope of \`, only / and \\ have a special meaning. / evaluates a sexp in its scope and \\ plays the role of quotation. For examples, see (L6), (L7), (L19). Note that quasi-quotation is useful if we want to suspend evaluation of applications.

2.5. definition of function and macro

A *function definition* is of the form

> *func fml* = *body*;

where *func* is a name, *fml* is a formal parameter which is a cons list and *body* is a sexp. Similarly, a *macro definition* is of the form

> *mac fml* := *body*;

where *mac* is a name. We cannot associate a function and a macro to the same name. For examples, see (L3), (L8), (L9), (L11), (L15), (L17), (L19).

A *formal parameter* is defined as follows:

(i) a variable is a formal parameter.
(ii) 0 is a formal parameter.
(iii) if f_1 and f_2 are formal parameters, so are $(f_1\ .\ f_2)$ and $[f_1\ .\ f_2]$.

In a formal parameter, $f_1 = f_2$ denotes $[f_1\ .\ f_2]$.

2.6. apply

A special sexp (**APPLY**, *x*) is called an application. We introduce syntax sugarings for an application. They are

$$fun(arg_1,\ \cdots,\ arg_n) \quad (1)$$

$$Fun[arg_1,\ \cdots,\ arg_n] \quad (2)$$

where *fun* is a name, *Fun* is a nonempty string of alphanumeric characters whose length is longer than 1 and which begins with an uppercase character and arg_i is a sexp. *Fun* is distinguished syntactically from a variable by the following '['.

(1), (2) denotes respectively

$$(\mathbf{APPLY},\ (fun\ .\ [arg_1,\ \cdots,\ arg_n]))$$

$$(\mathbf{APPLY},\ [fun',\ arg_1,\ \cdots,\ arg_n])$$

where *fun'* is a name obtained by replacing the leading uppercase character of *Fun* by the corresponding lowercase character.

Example 2.2.

cons(apple, orange)
= (**APPLY**, (cons . [apple, orange]))
Cons[apple, orange]
= (**APPLY**, [cons, apple, orange]) □

(1) and (2) are evaluated by the following rules.

(*r*11) $eval(fun(arg_1,\ \cdots,\ arg_n))$
$= apply(fun,\ eval([arg_1,\ \cdots,\ arg_n]))$
(*r*12) $eval(Fun[arg_1,\ \cdots,\ arg_n])$
$= apply(fun',\ [arg_1,\ \cdots,\ arg_n])$

Note that in $(r11)$ *eval* also plays the role of *evlis* in ordinary Lisp because of the rules $(r1)$, $(r2)$. Whether an argument list is evaluated or not is decided not by a function or macro but by the form of a function or macro call.

apply(fun, argl) is computed as follows. If *fun* is an atom other than eq or cond, it is regarded as a function or macro name and its definition is searched. If found, a new environment is created by the following rules from the argument list *argl* and the formal parameter *fml* of the definition.

pairup(0, *argl*) = []
pairup(*v*, *argl*) = [[*v* . *argl*]]
where *v* is a variable
pairup((f_1 . f_2), ($argl_1$. $argl_2$))
= *append*(*pairup*(f_1, $argl_1$), *pairup*(f_2, $argl_2$))
pairup((f_1 . f_2), [$argl_1$. $argl_2$])
= *append*(*pairup*(f_1, $argl_1$), *pairup*(f_2, $argl_2$))
pairup([f_1 . f_2], *argl*)
= *append*(*pairup*(f_1, *argl*), *pairup*(f_2, *argl*))

where *append* concatenates two lists.

Example 2.3.

pairup((Z = (X, Y)), [[aa, bb]])
= [[Z . [aa, bb]], [X . aa], [Y . bb]]

Recall that $f_1 = f_2$ denotes [f_1 . f_2]. □

Note that a formal parameter is used as a skeleton that is matched with the argument list and that a formal parameter matches any sexp because of the totality of *car* and *cdr*.

If *fun* is a function, its body is evaluated under the new environment.

Example 2.4.

See (L3) and (L4). cons(apple, orange) is evaluated as follows:

Evaluating the argument list [apple, orange] results in [apple, orange]. The formal parameter of cons is (X, Y) and the body is (X . Y). Pairup of (X, Y) and [apple, orange] creates a new environment

[[X . apple], [Y . orange]]

Under this environment, (X . Y) is evaluated and results in

(apple . orange) □

If *fun* is a macro, its body is evaluated under the new environment and the result is evaluated again under the environment that was current when the macro was called.

Example 2.5.

See (L17) and (L18). apply(cons, [lisp, prolog]) is evaluated as follows:

Evaluating the argument list [cons, [lisp, prolog]] results in [cons, [lisp, prolog]]. The formal parameter of apply is (F, X) and the body is ('**APPLY**, [F . X]). Pairup of (F, X) and [cons, [lisp, prolog]] creates a new environment

[[F . cons], [X . [lisp, prolog]]]

Under this environment, ('**APPLY**, [F . X]) is evaluated and results in

(**APPLY**, [cons, lisp, prolog])

This sexp, i.e., Cons[lisp, prolog], is evaluated under the previous environment yielding the result (lisp . prolog). □

Qute has two built-in functions eq and cond. eq returns 0 if its two arguments are equal, 1 otherwise. The definition of cond follows that of usual Lisp, except that 0 represents *truth* and other sexps *falsity*. We have a syntax sugaring for an argument of cond. That is,

c -> b denotes (c, b)

Example 2.6.

See (L19) and (L20). And[eq(aa, aa), 0] is evaluated as follows:

The argument of And is not evaluated. Pairup creates an environment

[[x . [eq(aa, aa), 0]], [x1 . eq(aa, aa)], [x2 . [0]]]

Under this environment, the body of and is evaluated and the result is

Cond[eq('[eq(aa, aa), 0], 0) -> 0,
eq(aa, aa) -> And[. [0]]]

This result is evaluated again. As *eval*(eq('[eq(aa, aa), 0], 0)) = 1 and *eval*(eq(aa, aa)) = 0, And[0] is evaluated. And[0] is evaluated similarly and the result is 0. □

In addition to a function and macro call, Qute also has a *lambda expression*, which we explain in 2.7.

2.7. summary

We summarize the definition of *eval* in this section. Here we define *eval* as a function from S×S to S, that is, *eval(x, env)* = *y* means that evaluating *x* under the environment *env* results in *y*.

eval(x, env)
= **if** *x* = 0 **then** 0
elif *atom(x)* **then**
snoc(eval(car(x), env), eval(cdr(x), env))
elif *car(x)* = **VAR then** *get(x, env)*
elif *car(x)* = **APPLY then**
if *mole(cdr(x))* **then**
if *atom(appl(x))* **then** *apply(fn(x), arg(x), env)*
else *apply(fn(x), eval(arg(x), env), env)* **fi**
else *true(value(x), prd(x), lvars(x), env)* **fi**
elif *car(x)* = **QUOTE then** *cdr(x)*
elif *car(x)* = **QQUOTE then** *qeval(cdr(x), env)*
else *cons(eval(car(x), env), eval(cdr(x), env))* **fi**

(For the explanation of *true*, see section 4.)

qeval(x, env)
= **if** *x* = 0 **then** 0
elif *atom(x)* **then**
snoc(qeval(car(x), env), qeval(cdr(x), env))
elif *car(x)* = **ESC then** *cdr(x)*
elif *car(x)* = **EVAL then** *eval(cdr(x), env)*
else *cons(qeval(car(x), env), qeval(cdr(x), env))* **fi**

where *appl* is *cadr*, *fn* is *caadr*, *arg* is *cdadr*, *value* is *cadr*, *prd* is *caddr* and *lvars* is *cadddr*.

```
get(v, env)
= if v = var(env) then val(env)
else get(v, rest(env)) fi
```

where *var* is *caar*, *val* is *cdar* and *rest* is *cdr*.

```
apply(f, arg, env)
= if atom(f) then
    if f = eq then eq(car(arg), cadr(arg))
    elif f = cond then evcon(arg, env)
    elif func(f) then
        eval(body(f), pairup(formal(f), arg))
    elif macro(f) then
        eval(eval(body(f), pairup(formal(f), arg)), env)
    fi
else eval(bdy(f), append(pairup(fml(f), arg), env))) fi
```

where *func(f)* and *macro(f)* decides whether *f* is a function or a macro, *body(f)* is the body of a definition of *f*, *formal(f)* is the formal parameter, *bdy* is *cadr* and *fml* is *car*. (else-part corresponds to lambda expression.)

```
evcon(cls, env)
= if cls = [] then 1
elif eval(prem(cls), env) = 0 then eval(ant(cls), env)
else evcon(rest(cls), env) fi
```

where *prem* is *caar* and *ant* is *cadar*.

This describes only the pure part of *eval*. Qute has a built-in function 'set' which can change the environment. (We omit the explanation in this paper.)

3. The Prolog-part of Qute

The Prolog-part of Qute is similar to an ordinary Prolog, but there is an important difference, i.e., the argument list of the predicate and the parameter list of the assertion are evaluated before they are unified with the assertions.

We give examples in 3.1, explain syntax of Qute in 3.2 and mechanism of unification in 3.3, 3.4.

3.1. examples

```
(P1)  + cons | X, Y, (X . Y);
      cons defined
(P2)  - cons[apple, orange, X];
      X = (apple . orange)
(P3)  + cadr | X, Y - eq[Y, car(cdr(X))];
      cadr defined
(P4)  - cadr[cons(left, (right)), X];
      X = right
(P5)  + append1
             | 0, Y, Y
             | [X1 . X2], Y, [X1 . Z2]
                     - append1[X2, Y, Z2];
      append1 defined
(P6)  - append1[[aa, bb], X, [aa, bb, cc, dd]];
      X = [cc, dd]
(P7)  + append2 | X, Y, append(X, Y);
      append2 defined
(P8)  - append2[[prolog, lisp], [qute], X];
      X = [prolog, lisp, qute]
```

3.2. definition of predicate

A *predicate* is of the form *prd arglist* where *prd* is a name and *arglist* is a snoc list. Its denotation is [*prd* . *arglist*]. *prd* is called the *predicate name* of the predicate and *arglist* is called the *argument list* of the predicate.

A *predicate definition* (*assertion*) is of the form

$$
\begin{aligned}
+\ prd\ &|\ param_1\ body_1\\
&|\ param_2\ body_2\\
&\ \cdots\\
&|\ param_n\ body_n\\
&;
\end{aligned}
$$

where *prd* is a name, $param_i$ is of the form

$$p_i^1,\ \cdots,\ p_i^{k_i} \quad \text{or} \quad .p_i^1 \quad \text{or}$$
$$p_i^1,\ \cdots,\ p_i^{k_i}\ .\ p_{i+1}^{k_{i+1}}$$

where p_i^j is a sexp, and $body_i$ is empty or of the form

$$-\ predicate_i^1 - predicate_i^2\ \cdots\ - predicate_i^{m_i}$$

where $predicate_i^j$ is a predicate.

The corresponding Marseille notation is

$$+prd[param_1]-predicate_1^1-\ \cdots\ -predicate_1^{m_1};$$
$$\cdots$$
$$+prd[param_n]-predicate_n^1-\ \cdots\ -predicate_n^{m_n};$$

We call $[param_i]$ a *parameter list* of the assertion.

A *goal* is of the form

$$-\ predicate_1 - predicate_2\ \cdots\ - predicate_m;$$

where $predicate_i$ is a predicate.

3.3. variable and its value

First, we define the notion of *free variable*. The following function *vars(f)* is used to define the set of free variables in *f*.

```
vars(f)
= if f = 0 then φ
elif f = [f1 . f2] then vars(f1) ∪ vars(f2)
elif f = (VAR . t) then {f}
elif f = fun(arg1, ···, argn) then
    vars([arg1, ···, argn])
elif f = Fun[arg1, ···, argn] then φ
elif f = Epsilon(val; body) then
    vars(body) - vars(val)
elif f = 't then φ
elif f = `t then qvars(t)
elif f = (f1 . f2) then vars(f1) ∪ vars(f2) fi
```

(For simplicity, we omit the case of lambda expression.)

```
qvars(f)
= if f = 0 then φ
elif f = [f1 . f2] then qvars(f1) ∪ qvars(f2)
elif f = \t then φ
elif f = /t then vars(t)
elif f = (f1 . f2) then qvars(f1) ∪ qvars(f2) fi
```

When a sexp / is evaluated, it is necessary to know the values of the free variables in /. However, the intended meaning of a free variable in a predicate is an unknown sexp which may be known after evaluation. We introduce the notion of an *undefined value* (it is an imaginary element outside of S). We suppose that each free variable has a different undefined value.

3.4. unification

Before evaluating the predicates in a goal, an environment which is a list of pairs of a free variable and an undefined value is set up. Before an unification is made with a predicate definition, an environment is set up similarly, using free variables in the parameter and the body of the predicate definition. An undefined value plays the role of a 'variable' in unification.

Example 3.1.

— appendll [prolog, lisp], [qute], X];

creates an environment

[[X . $undf_1$]]

before evaluation, where *undf* is an undefined value. Evaluating [[prolog, lisp], [qute], X] results in

[[prolog, lisp], [qute], $undf_1$ (1)

According to the definition of appendl in (P5), the first parameter list [0, Y, Y] is evaluated first and its result is [0, $undf_2$, $undf_2$]. This is not unifiable with (1). So the second parameter list [[XI . X2], Y, [XI . Z]] is evaluated and its result

[[undfs . $undf_A$], $undf_5$, [$undf_3$. $undf_6$]]

is unified with (1). At this time, the environment is

[[XI . prolog], [X2 . [lisp]], [Y . [qute]],
[Z . undfi)]

and *undf* is instantiated to [$undf_3$. $undf_6$. In this way, execution goes on. After execution, the first environment is instantiated to

[[X . [prolog, lisp, qute]]] D

Since no restriction is imposed on the parameters of an assertion, they may contain any special sexp as is seen in (P7). Unification with such an assertion goes like the following.

Example 3.2.

(P8) is executed under the definition (P7), where append is a function defined in (LI5). [[prolog, lisp], [qute], X] is evaluated with the result

[[prolog, lisp], [qute], $undf_1$]. (1)

A parameter list [X, Y, append(X, Y)] of (P7) is evaluated and its result is

[$undf_2$ $undf_3$ $undf_4$ (2)

with the condition

$undf_4$ - append($undf_2$,$undf_3$).

That is, since we cannot evaluate append (X, Y) with free variables X and Y, we assume that its value is $undf_4$ and impose the above condition. (1) and (2) are unified and as a result undefined values are instantiated, i.e., $undf_2$ — [prolog, lisp], $undf_3$ — [qute] and $undf_4$ — *undf*. The condition is instantiated to *undf* = append ([prolog, lisp], [qute]) and it is checked, append ([prolog, lisp], [qute]) is evaluated and *undf* is instantiated to [prolog, lisp, qute]. D

4. Connecting Lisp and Prolog

One of the most important features of Qute is that the Prolog-part can be called from the Lisp-part. It is a mechanism similar to Hilberfs epsilon symbol, that is, a mechanism to find a value which makes a certain predicate to hold.

4.1. examples

```
(EI)  > append3(X, Y)
       - Epsilon(V; append3[X, Y, V]);
      append3 defined
(E2)  4- append3
           [] [], Y, Y
            [x . X], Y, [x . Z]
               .   - append3[X, Y, Z]

      append3 defined
(E3)  append3([lisp, prolog], [qute]);
      — [lisp, prolog, qute]
(E4)  > append4(X, Y)
      = Epsilon(V; append4[X, Y, V]);
      append4 defined
(E5)   + append4
           I [], Y, Y
           I [x . X], Y, [x . append4(X, Y)]

      append4 defined
(E6)   + member
           | x, [x . X]
           I x, [y . X]
                 — member[x, X]

      member defined
(E7)  Epsilon(x; membertx, [apple, orange]]);
      — apple
(E8)  — eq[orange,
          Epsilon(x; memberlx, [apple, orange]])];
      yes
```

4.2. epsilon expression

A special sexp (APPLY . *[val, prds, vars])* is called an *epsilon expression* where *val* is a sexp, *prds* is a predicate or a cons list of predicates (i.e., conjunction of predicates) and *vars* is a snoc list of variables that are local in the epsilon expression. We introduce a syntax sugaring for an epsilon expression. It is

Epsilon*(val; body)*

where *val* and *body* are sexps. It denotes

(APPLY . *[val, body, vars])*

where *vars* is the snoc list of the free variables in *val.*

According to the definition of *eval* in 2.7,

eva/((APPLY . *[val, body, vars]), env)*
= *trueival, body, vars, env)*

It is computed as follows. A new environment *E* is set up by appending to the head of the current environment *env* a list consisting of pairs of a free variable in *vars* and an undefined value, *body* is executed by the Prolog-part under this environment *E* with the result that the environment is instantiated to an environment *E'* that makes *body true.* According to the formal specification of Qute, any *E'* that makes *body true* is accepted, but the actual implementation finds *E'* in a depth-first way. Then *val* is evaluated under the instantiated environment E" and the result is the value of (APPLY . *[val, body, vars]).* When *E'* is created a marker to this frame is also made, so that a later backtrack will return to this point.

Epsilon expression is therefore a multi-valued function, however only one value is returned at a time and further values may be obtained by using backtrack.

Example 4.1.

(E3) is evaluated under the definitions (El), (E2). append3 is called and the environment

[[X . [lisp, prolog]], [Y . [qute]]]

is created by *pairup.* Epsilon expression which is the body of append3 is evaluated. First, a new environment

[[V . *undf*$_1$*],* [X . [lisp, prolog]], [Y . [qute]]]

is created. A goal append3[X, Y, V] is executed under this environment according to the definition (E2). The environment is instantiated to

[[V . [lisp, prolog, qute]], [X . [lisp, prolog]],
[Y . [qute]]]

and V is evaluated under this environment with the result [lisp, prolog, qute]. It is the value of the epsilon expression and of append3([lisp, prolog], [qute]). □

Example 4.2.

(E6) defines an ordinary membership relation on a list. In (E7), the Lisp-part of Qute sets up an environment

E - [[x . *undf*$_1$*]]*

and calls the Prolog-part. The Prolog-part tries to find an instance of *E* that makes membertx, [apple, orage]] *true.* The following two instances of *E* both give a correct instance:

E_1 - [[x . apple]], E_2 - [lx. orange]]

However, the actual implementation does a depth-first search and returns E_1 as a new environment. The Lisp-part evaluates x in this environment and returns apple as the value of (E7).

In executing (E8), the two arguments of the predicate eq are evaluated first. The second argument, which is the same epsilon expression as (E7), is evaluated similarly as above and apple is returned as its value. Since orange and apple are not 'eq' (equal), a backtrack occurs. This forces Qute to find a second value of the epsilon expression and the value orange. will be returned this time. Since orange and orange are 'eq' 'yes' is returned as the answer to the question (E8). □

5. Conclusions and Future Plans

We have shown that it is possible to amalgamate Prolog and Lisp in a natural way. A comparison of the evaluation of a Qute predicate with that of an atomic formula in a first order language will make this naturalness clear. Consider a first order language that includes:

a binary predicate symbol < (for less than),
a binary function symbol + (for plus) and
constants for natural numbers

with their usual interpretations. Then the truth value of the atomic formula

$2+3 < 6$

is evaluated as follows. First evaluating the terms 2+3 and 6, we get 5 and 6. Then by the meaning of < we see that 5 is less than 6, which implies the truth of the formula in question. The evaluation in Qute is completely analogous. With appropriate definitions of "less_than" and "plus", the question:

— less_than[plus(2, 3), 6];

is evaluated by Qute resulting in the answer "yes".

According to this analogy, the evaluation of a sexp by the Lisp part of Qute corresponds to the evaluation of a term. Here, an epsilon expression corresponds to Hilbert's e-term.

We have defined the semantics of Qute informally in this paper. We wish to give a formal definition of Qute in a forthcoming paper. (See also Sakurai[2].) This will be done as follows. First, we will define a formal intuitionistic theory of symbolic expressions called SA which is proof theoretically equivalent to Heyting's arithmetic HA. It will then become possible to define Qute within SA. Moreover, to mechanize these processes, we will implement a proof checking system for SA using Qute. In this way, we will be able to formally reason about the properties of Qute within Qute itself.

REFERENCES

[1] Robinson, J. A., and Sibert, E. E., 1982: LOGLISP: an alternative to PROLOG, *Machine Intelligence 10.*

[2] Sakurai, T., 1983: Formalism for Logic Programming, *Master Thesis, Department of information Science Faculty of Science University of Tokyo.*

[3] Sato, M, 1983: Theory of Symbolic Expressions, I, *Theoretical Computer Science,* 22,19-55.

[4] Sato, M. and Hagiya, M., 1981: Hyperlisp, *Algorithmic Languages, Proceedings of the International Symposium on Algorithmic Languages, (eds. J. W- de Bakker andJ.C van Vliet),* North-Holland.