PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1984,
edited by ICOT.

QUTE: A FUNCTIONAL LANGUAGE BASED ON UNIFICATION

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 programming language called Qute is introduced. Qute is a functional programming language which permits parallel evaluation.

While most functional programming languages use pattern matching as basic variable-value binding mechanism, Qute uses unification as its binding mechanism. Since unification is bidirectional, as opposed to pattern match which is unidirectional, Qute becomes a more powerful functional programming language than most of existing functional languages.

This approach enables the natural *unification* of logic programming language and functional programming language. In Qute it is possible to write a program which is very much like one written in conventional logic programming language, say, Prolog. At the same time, it is possible to write a Qute program which looks like an ML (which is a functional language) program.

A Qute program can be evaluated in parallel (and-parallelism) and the same result is obtained irrespective of the particular order of evaluation. This is guaranteed by the Church-Rosser property enjoyed by the evaluation algorithm.

A completely formal semantics of Qute is given in this paper.

1 INTRODUCTION

In this paper, we will introduce a new programming language called Qute, and will define its semantics formally. The name Qute may be confusing to some readers, since we have already reported about previous versions of Qute in [7], [8]. The new Qute, which we will describe in this paper is rather different from the previous ones although it inherits many things from them. In spite of this, we have decided to call the new language also Qute.

Qute is a functional programming language which permits parallel evaluation. While most functional languages uses pattern matching as basic variable-value binding mechanism, Qute uses unification as its binding mechanism. Since unification is bidirectional, as opposed to pattern match which is unidirectional, Qute becomes a more powerful functional language than most of existing functional languages.

This approach enables the natural *unification* of logic programming language and functional language. In Qute it is possible to write a program which is very much like one written in conventional logic programming language, say, Prolog. At the same time, it is possible to write a Qute program which looks like an ML (which is a functional language) program.

In the design of Qute we tried to minimize the number of basic concepts, so that the language becomes easy to learn and specify. These concepts were selected from logical considerations, and as a result some of them were inherited from Concurrent Prolog ([9], [10]) and ML ([2]). In particular, we imported the concepts of 'parallel and' and 'sequential and' from Concurrent Prolog. Furthermore, Qute has 'negation' and 'if then else' as two of the basic programming constructs. Though they also are selected from the logical consideration, they act as synchronization mechanism in 'parallel and'.

A Qute program can be evaluated in parallel (and-parallelism only) and the same result is obtained irrespective of the particular order of evaluation. This is guaranteed by the Church-Rosser property enjoyed by the evaluation algorithm. Although it is possible to add a nondeterministic feature to Qute orthogonally, this point is not discussed in this paper.

Qute is a programming language that evaluates an expression under a certain environment. The evaluation process can be considered as a reduction process of the given expression, and the

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

evaluation stops when the expression has been reduced to a *normal expression* for which no more reduction is possible. Through the process of evaluation the given environment is also changed to another environment by unification. Therefore the result of an evaluation can be considered as a pair of normal expression and an environment. We explain these points in the following order. In 2 we define the syntax of Qute, and in 3 we define the fundamental concepts of unification and environment. In 4 we explain the semantics of Qute informally and give some examples, and in 5 we give a completely formal semantics of Qute.

2 SYNTAX OF QUTE

2.1 symbolic expressions

We define the domain of symbolic expressions (*sexps*, for short), which is used to define the semantics of Qute. Namely, later in this paper we give the formal semantics of Qute by interpreting the Qute programs in the domain of sexps. Symbolic expressions are constructed by the following clauses:

1. $*$ (*nil*) is a sexp.
2. If s and t are sexps then $[s \,.\, t]$ is a sexp.
3. If s and t are sexps then $(s \,.\, t)$ 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. A sexp can be considered as a binary tree whose leaf is $*$ and whose node contains a one bit information.

We introduce dot notation and list notation as notations for sexps. (We will use the symbol '$\equiv$' as informal equality symbol, and will reserve the symbol '$=$' as formal equality sign used in the programming language Qute.)

$$[.\ x] \equiv x$$
$$[x_1, \cdots, x_n \,.\, x_{n+1}] \equiv [x_1 \,.\, [x_2, \cdots, x_n \,.\, x_{n+1}]] \quad (n \geqq 1)$$
$$[x_1, \cdots, x_n] \equiv [x_1, \cdots, x_n \,.\, *] \quad (n \geqq 0)$$

2.2 symbol and variable

Although it is possible to define the concept of an expression as a string of characters defined by a certain set of grammatical rules, we will define the concept of an expression as a certain sexp. We take this approach only because the semantics of Qute can be conveniently given for an expression represented as a sexp. We first define two auxiliary concepts of a symbol and a variable.

A sexp of the form $(* \,.\, t)$ is called a *symbol* Symbols are used to represent basic data objects, and for this paper, we assume that the domain of integers and the domain of strings of ASCII characters are represented by two disjoint sets of symbols. (How they are actually represented is not important.) We use usual decimal notation for integers, and a string of characters will be represented by enclosing it between a pair of '"' signs. Thus, e.g.,

"apple"

is a symbol which represents a word of length 5. We also choose a certain symbol (which is neither an integer nor a string) and call it *unit* We use the notation '()' to denote the unit. From now on we will consider only those symbols which are either a unit, an integer or a string.

A sexp of the form ("var" . s) where s is a string, is called a *pure variable*. We define a *variable* by the following inductive clauses.

(i) A pure variable is a variable.

(ii) If x is a variable then ("free" . x) is a variable.

For a string "$\cdots$" we use $\cdots$ to denote the pure variable ("var" . "$\cdots$"). Thus, e.g.,

X

denotes the variable ("var" . "X").

A sexp of the form ("gvar" . s) where s is a string, is called a *global variable*. We use a similar convention to denote global variable. In order to avoid notational ambiguity, we reserve the strings whose length is at least 2 and which begins with a lower case character for global variables. Thus 'foo' becomes a global variable, while 'x' is a variable.

In the following, we will call both variables and global variables simply variables and we will use x, y, z etc. to denote variables.

2.3 expression

We define an *expression* by the following inductive definition.

(E1) A variable is an expression.

(E2) A symbol is an expression.

(E3) $*$ is an expression.

(E4) If a, b are expressions then $[a \,.\, b]$ is an expression.

(E5) If a, b are expressions then ("and" . $(a \,.\, b)$) is an expression.

(E6) If a, b are expressions then ("lambda" . $(a \,.\, b)$) is an expression.

(E7) If a, b are expressions then ("seqand" . (a . b)) is an expression.

(E8) If a, b, c are expressions then ("if" . (a . (b . c))) is an expression.

(E9) If a is an expression then ("not" . a) is an expression.

(E10) If a, b are expressions then ("apply" . (a . b)) is an expression.

(E11) If a, b are expressions then ("equal" . (a . b)) is an expression.

We use a, b, c, d etc. to denote expressions. We will call an expression defined by (E6) a *function*, (E7) a *sequential-and*, (E8) an *if-then-else*, (E9) a *negation*, (E10) an *application*, and (E11) a *unification*.

2.4 abbreviations for expressions

We introduce following abbreviations for expressions.

(a) for a
$\#x$ for ("free" . x)
$\#^0 x$ for x
$\#^{n+1} x$ for ("free" . $\#^n x$) $(n \geq 0)$
a, b for ("and" . (a . b))
a, b, c for a, (b, c) etc.
$\lambda a.b$ for ("lambda" . (a . b))
$a; b$ for ("seqand" . (a . b))
$a; b; c$ for a; (b; c) etc.
if a **then** b **else** c for ("if" . (a . (b . c)))
$\neg a$ for ("not" . a)
$a\ b$ for ("apply" . (a . b))
$a\ b\ c$ for ($a\ b$) c etc.
$<a>$ for $(\lambda().a)()$
$a = b$ for ("equal" . (a . b))
fail for $0 = 1$

2.5 free variable

For an expression a, we define the set $FV(a)$ of *free variables in* a. $FV(a)$ is defined by $FV_0(a)$ where $FV_n(a)$ $(n \geq 0)$ is defined as follows:

$a \equiv \#^k x$ where x is a pure variable $\Rightarrow$

$$FV_n(a) \equiv \begin{cases} \phi & \text{if } k < n \\ \{\#^{k-n} x\} & \text{if } k \geq n \end{cases}$$

a is a global variable $\Rightarrow$ $FV_n(a) \equiv \{a\}$
a is a symbol $\Rightarrow$ $FV_n(a) \equiv \phi$
$FV_n(\bullet) \equiv \phi$
$FV_n([a\ .\ b]) \equiv FV_n(a) \cup FV_n(b)$
$FV_n(a,\ b) \equiv FV_n(a) \cup FV_n(b)$
$FV_n(\lambda a.b) \equiv FV_{n+1}(a) \cup FV_{n+1}(b)$
$FV_n(a;\ b) \equiv FV_n(a) \cup FV_n(b)$
FV_n(**if** a **then** b **else** c)
$\equiv FV_{n+1}(a) \cup FV_{n+1}(b) \cup FV_n(c)$
$FV_n(\neg a) \equiv FV_n(a)$
$FV_n(a\ b) \equiv FV_n(a) \cup FV_n(b)$
$FV_n(a = b) \equiv FV_n(a) \cup FV_n(b)$

This definition means that variables appearing in a function or an if/then-part of if-then-else are 'localized' but the effect can be canceled by #s preceding the variables.

Example

Consider the expressions:

$a \equiv$ **if** $\#x = []$ **then** $\#y$ **else** b
$b \equiv$ **if** $\#x = [X1.X2]$ **then** c **else fail**
$c \equiv (\lambda u.\,foo(u,\ \#X1,\ \#\#y))(\#z, X2)$

Then we have:

$FV(a)$
$\equiv FV_1(\#x = []) \cup FV_1(\#y) \cup FV(b)$
$\equiv \{x\} \cup \{y\} \cup FV_1(\#x = [X1.X2]) \cup FV_1(c) \cup FV(0 = 1)$
$\equiv \{x\} \cup \{y\} \cup \{x\} \cup FV_2(u) \cup FV_2(foo(u,\ \#X1,\ \#\#y)) \cup FV_1((\#z, X2)) \cup \phi$
$\equiv \{x\} \cup \{y\} \cup \{x\} \cup \phi \cup \{foo, y\} \cup \{z\} \cup \phi$
$\equiv \{x, y, z, foo\}$

□

3 UNIFICATION AND ENVIRONMENT

As we explained in the introduction, unification plays the fundamental role in our programming language Qute. Especially, the fact that unification enjoys a kind of Church-Rosser property is the key to make Qute a functional language which permits parallel evaluation. Martelli and Montanari [5] introduced a nondeterministic unification algorithm and proved the Church-Rosser theorem for the algorithm. Lassez and Maher [4] utilized the Church-Rosser property for unification, and proved the equivalence of various resolution strategies elegantly. Jaffar [3] considered the unification problem for the domain over regular infinite trees and proved the Church-Rosser theorem for his algorithm.

Our unification algorithm, which we are about to explain, constitutes a conceptual simplification of the algorithms by Martelli and Montanari [5] and by Jaffar [3].

3.1 unification

We first define a *pattern* by the following inductive definition.

(P1) A variable is a pattern.

(P2) A symbol is a pattern.

(P3) $\bullet$ is a pattern.

(P4) If p, q are patterns then $[p\ .\ q]$ is a pattern.

(P5) If p, q are patterns then p, q is a pattern.

(P6) If a, b are expressions then $\lambda a.b$ is a pattern.

We use p, q, r to denote patterns. We note that a pattern can be classified into 6 mutually disjoint categories (P1 - P6).

We define an *equation* as follows. There are two types of equations, that is, *marked* and *unmarked*. An *unmarked equation* is a finite nonempty set of patterns. An unmarked equation is *simple* if all of its members are variables. For a simple unmarked equation A, the singleton set $\{A\}$ is called a *mark*. A finite set such that (i) its members are patterns or marks and (ii) it contains at least one mark is called a *marked equation*. We use A, B, C to denote equations. Informally an unmarked equation means that all the members of the equation should be equated. The meaning of a marked equation will be explained in section 5.

Consider two distinct patterns which are not variables. They are defined to be *incompatible* if (i) they are two distinct symbols, (ii) they are two different functions or (iii) they belong to different categories. An equation is said to be *inconsistent* if it contains two patterns which are incompatible.

A finite (possible empty) set of equations is called a *system of equations* or simply a *system*. A system is said to be *inconsistent* if it contains an inconsistent equation; otherwise it is said to be *consistent*. We use Γ, Δ, Π etc. to denote systems. For a system Γ we define a relation $\rightarrow_\Gamma$ on Γ as follows. Let A and B be equations in Γ. Then $A \rightarrow_\Gamma B$ if and only if there exists a pattern p and a variable x such that (i) $p \in A$, (ii) $x \in B$, (iii) p belongs to the category (P4) or (P5) and (iv) $x \in \mathrm{FV}(p)$. A system Γ is said to *contain a loop* if there exist a sequence of equations $A_1, \cdots, A_n$ $(n \geqq 2)$ in Γ such that

$$A_1 \rightarrow_\Gamma A_2 \rightarrow_\Gamma \cdots \rightarrow_\Gamma A_n$$

and $A_1 = A_n$; otherwise Γ is said to be *loop free*.

We now define a binary relation $\Gamma < \Delta$ between systems by the following three clauses.

(1) If A, B are members of Γ and A, B have a variable in common then $\Gamma < (\Gamma - \{A\} - \{B\}) \cup \{A \cup B\}$.

(2) If $\{[p\ .\ q], [p'\ .\ q']\} \subseteq A \in \Gamma$ then $\Gamma < \Gamma \cup \{\{p, p'\}, \{q, q'\}\}$.

(3) If $\{(p, q), (p', q')\} \subseteq A \in \Gamma$ then $\Gamma < \Gamma \cup \{\{p, p'\}, \{q, q'\}\}$.

If $\Gamma < \Delta$ and $\Gamma \neq \Delta$ we say that Γ *reduces* to Δ. In this case Γ is said to be *reducible*; otherwise Γ is said to be *irreducible*. We let $\leqq$ be the reflexive and transitive closure of the relation $<$. We then have the following theorems.

Theorem 1.

If $\Gamma \leqq \Delta_1$ *and* $\Gamma \leqq \Delta_2$ *then* $\Delta_1 \leqq \Pi$ *and* $\Delta_2 \leqq \Pi$ *for some* Π.

Theorem 2.

There is no infinite sequence of systems Δ_i $(i \geqq 0)$ *such that* Δ_i *reduces to* Δ_{i+1} *for all* i.

Theorem 3.

For any system Γ *there uniquely exists an irreducible* Δ *such that* $\Gamma \leqq \Delta$.

We will denote the Δ in Theorem 3 by Γ^* and we will call it the *solution* of Γ.

Theorem 1 states that the reduction process (or unification process) has the Church-Rosser property and Theorem 2 states that the reduction process always terminates. Theorem 3 is just a consequence of Theorems 1 and 2. We do not prove these theorems here, and we refer to Sato [6] for the proofs of them.

3.2 environment

By the results of the previous subsection, the unification process of the given system Γ ends up with the unique solution Γ^*. We then have one of the following three mutually disjoint conditions.

(i) Γ^* is inconsistent.

(ii) Γ^* is consistent and contains a loop.

(iii) Γ^* is consistent and is loop free.

Condition (i) means the failure of unification. Condition (ii) corresponds to the failure by occur-check of the usual unification algorithm, so that in this case the solution must be regarded as unacceptable if we were to solve the equation in the *finite* sexps. (This also means that it is possible to separate occur-check from the unification process.) On the other hand the same solution becomes acceptable if we interpret it in the domain of *regular infinite* sexps. In this paper we take the latter position, and consider that the condition (ii) gives an acceptable solution. (The story goes almost in parallel and is even simpler if we consider (ii) as unacceptable.)

We, therefore, define an *environment* as a consistent irreducible system. So, for any system Γ, Γ^* becomes an environment if and only if it is consistent. We use E, F, G to denote environments. An environment E is said to be *suspended* if E contains an equation A such that (i) it contains two distinct marks or (ii) it contains a mark and a non-variable pattern.

The value of a pattern is determined relative to an environment. For instance, if

$E = \{\{x, y, 1\}, \{z, 2\}\}$ then the value of $[x, y, z]$ is $[1, 1, 2]$. In general, the *value of* a pattern p *in* the environment E, which we will denote by p_E, is defined as follows.

(i) If p is a variable and there exists no equation in E which contains p then $p_E = p$.

(ii) If p is a variable and there exists an equation A in E which contains p then:

(ii.i) if A is simple then $p_E = p$,

(ii.ii) else if there is a loop containing A then $p_E = p$,

(ii.iii) else $p_E = q_E$ where q is a non-variable pattern in A.

(iii) If p is a symbol then $p_E = p$.

(iv) $*_E = *$.

(v) $[p \cdot q]_E = [p_E \cdot q_E]$.

(vi) $(p, q)_E = (p_E, q_E)$.

(vii) $(\lambda a \cdot b)_E = \lambda a \cdot b$.

If x is a variable and satisfies (i) or (ii.i), we say that x is an undefined variable in E.

4 SEMANTICS OF QUTE

Here, we explain the semantics of Qute informally.

4.1 evaluable subexpression

For an expression a, we define the set Σa of all the *evaluable subexpressions* of a as follows.

If p is $*$, a symbol, a variable or a function then $\Sigma p = \phi$

$\Sigma[a \cdot b] = \Sigma a \cup \Sigma b$

$\Sigma(a, b) = \Sigma a \cup \Sigma b$

$$\Sigma(a; b) = \begin{cases} \Sigma a & \text{if } \Sigma a \neq \phi \\ \Sigma b & \text{if } \Sigma a \equiv \phi \text{ and } \Sigma b \neq \phi \\ \{a; b\} & \text{if } \Sigma a \equiv \Sigma b \equiv \phi \end{cases}$$

$\Sigma(\textbf{if } a \textbf{ then } b \textbf{ else } c) = \{\textbf{if } a \textbf{ then } b \textbf{ else } c\}$

$\Sigma \neg a = \{\neg a\}$

$$\Sigma(a\ b) = \begin{cases} \{a\ b\} & \text{if } \Sigma a \equiv \Sigma b \equiv \phi \\ \Sigma a \cup \Sigma b & \text{otherwise} \end{cases}$$

$$\Sigma(a = b) = \begin{cases} \{a = b\} & \text{if } \Sigma a \equiv \Sigma b \equiv \phi \\ \Sigma a \cup \Sigma b & \text{otherwise} \end{cases}$$

It is easy to check that for any expression a, $\Sigma a \equiv \phi$ if and only if a is a pattern. If $b \in \Sigma a$, we say that b is an *evaluable subexpression* of a.

Example

$$x = 1, f(g(x), y = h(x)), (z = [u, v]; k(z))$$

The evaluable subexpressions of this expression are

$$x = 1,\ g(x),\ h(x),\ z = [u, v]$$

□

4.2 semantics

The evaluation of an expression proceeds by selecting an evaluable subexpression nondeterministically and by reducing it following the rules which we explain in the following. If the reduced evaluable subexpression has evaluable subexpressions, they also are the candidates for the selection. If the reduction of an evaluable subexpression of a given expression fails, then the evaluation of the whole expression fails. The evaluation of an expression terminates when:

(i) there is no more evaluable subexpression, i.e. the expression is reduced to a pattern. (In this case, we say that the evaluation succeeds.)

(ii) the reduction of an evaluable subexpression fails.

(iii) every evaluable subexpression is suspended. (The condition of suspension is explained in the following.)

Although an evaluable subexpression is selected nondeterministically, the result of the evaluation is unique so long as it terminates because of the Church-Rosser property.

Qute evaluates an expression relative to an environment and the environment is changed as the evaluation proceeds.

As is known from the definition of an evaluable subexpression, unification, sequential-and, application, negation, and if-then-else are candidates for an evaluable subexpression. We explain when they become evaluable subexpressions and how they are reduced.

We say that an expression is *renamed* if each free pure variable of the expression is renamed to a new global variable and one # of each free non-pure and non-global variable is stripped.

In the following examples, we use $\triangleright$ to represent one step of reduction.

1. unification

$a = b$ becomes an evaluable subexpression after a and b are reduced to patterns p and q. If E is an environment at that time, p and q are unified under E, i.e. $F = (E \cup \{\{p, q\}\})^*$ is computed. If F is inconsistent, the reduction of $p = q$ fails. Otherwise, $p = q$ is reduced to p and further reductions proceed under F. The environment is changed only by unification.

Example

$$z = [x \cdot y], x = 1, y = 2 \quad \text{in } E_0$$
$$\triangleright\ z, x = 1, y = 2 \quad \text{in } E_1$$
$$\triangleright\ z, x, y = 2 \quad \text{in } E_2$$

$\triangleright$ z, x, y in E_3

where

$E_0 = \phi$
$E_1 = \{\{z, [x\ .\ y]\}\}$
$E_2 = \{\{z, [x\ .\ y]\}, \{x, 1\}\}$
$E_3 = \{\{z, [x\ .\ y]\}, \{x, 1\}, \{y, 2\}\}$

The value of z, x, y in E_3 is [1 . 2], 1, 2.

Though this shows only one of the possible order of the reduction (z = [x . y], x = 1, and y = 2 are evaluable subexpressions), the result is the same irrespective of the order of the reduction. This holds because of the Church-Rosser property of the unification algorithm. □

2. sequential-and

In reducing $a; b$, a is reduced to a pattern p first and then b is reduced. If b is reduced to a pattern q, $p; q$ becomes an evaluable subexpression and is reduced to q.

Example

z = [x . y]; x = 1; y = 2 in E_0
$\triangleright$ z; x = 1; y = 2 in E_1
$\triangleright$ z; x; y = 2 in E_2
$\triangleright$ z; x; y in E_3
$\triangleright$ z; y in E_3
$\triangleright$ y in E_3

where E_0, E_1, E_2 and E_3 are the same as those of the above example.

Note that in this example the order of the reduction is unique. (See the definition of the evaluable subexpression.) □

3. application

$a\ b$ becomes an evaluable subexpression after a and b are reduced to patterns p and q. This means application is computed by call-by-value. If the value of p is $\lambda a\ .\ b$, a, b is renamed to a', b' to avoid the collision of names. $p\ q$ is reduced to $a' = q$; b', i.e. the formal parameter a' is bound to the actual parameter q by $a' = q$ and then the body b' is reduced.

Example

cons(1,2) in F_0
$\triangleright$ (x',y') = (1,2); [x' . y'] in F_0
$\triangleright$ (x',y'); [x' . y'] in F_1
$\triangleright$ [x' . y'] in F_1

where

$F_0 = \{\{\text{cons}, \lambda x,y\ .\ [x\ .\ y]\}\}$
$F_1 = F_0 \cup \{\{(x',y'), (1,2)\}, \{x', 1\}, \{y', 2\}\}$

The value of [x' . y'] in F_1 is [1 . 2]. □

4. negation

Roughly speaking, $\neg a$ is reduced to () if the evaluation of a fails, and the reduction of $\neg a$ fails if a is reduced to a pattern. This, however, is a problematic definition. We point out the problem by an example. Consider the evaluation of the expression:

$\neg(x = 0)$, $x = 1$

in the empty environment. Both $\neg(x = 0)$ and $x = 1$ are evaluable subexpressions. If $\neg(x = 0)$ is reduced first, the reduction fails because the evaluation of $x = 0$ succeeds and the evaluation of the whole expression fails. If $x = 1$ is reduced first, $\neg(x = 0)$ is reduced under the environment $\{\{x, 1\}\}$. Therefore the reduction of $x = 0$ fails and $\neg(x = 0)$ is reduced to (). This contradicts Church-Rosser property which we are going to establish. Reconsider the former case. Intuitively, failure of $\neg(x = 0)$ means x *is not* 0. However, since x is an undefined variable at that time, the value of x may become 0, 1, or some other value later. That is, it is too early to decide that x is not 0. (The condition that x is not 0 cannot be explained in the environment.) Therefore the decision should be suspended until x is instantiated enough to decide whether x is 0 or not, i.e. in this case until $x = 1$ is reduced.

In general, $\neg a$ is reduced as follows. Let E be the environment, V be the set of free variables of a, FVL be the list of free variables in $\{x_E \mid x \in V\}$, and a', E' and FVL' be the copies of a, E and FVL. a' is evaluated under E'.

(1) If the evaluation of a' succeeds, the state of FVL' is divided into two cases:
  (1.1) FVL' is not instantiated. In this case, a can be evaluated under the original environment E without affecting it, i.e. the evaluation of a will succeed whatever instantiation may be done on FVL. Therefore the reduction of $\neg a$ immediately fails.
  (1.2) FVL' is instantiated. In this case, FVL may be instantiated later and the instantiation may be incompatible with FVL'. Therefore the reduction of $\neg a$ should be suspended until later.

(2) If the evaluation of a' fails, it will fail whatever instantiation may be done on FVL. Therefore $\neg a$ is immediately reduced to () and further reductions proceed under E. (The environment is not changed.)

(3) If the evaluation of a' is suspended, the reduction of $\neg a$ is suspended and will be tried again later.

Negation can be defined by if-then-else. It is left as a primitive only for convenience.

5. if-then-else

In reducing **if** a **then** b **else** c, a, b is renamed to a', b'. **if** a **then** b **else** c is reduced to b' if the evaluation of a' succeeds and **if** a **then** b **else** c is reduced to c if the evaluation of a' fails. However, just as in the case of negation, the decision whether the evaluation of a' succeeds or not should be suspended as needed. Therefore, the reduction mechanism is the same as that of negation except that the set V in the explanation of the negation is the set of free variables of $<a>$ (not a').

Example (Eratosthenes' sieve)

We use the abbreviation $a\ b \Leftarrow c$ for $a = \lambda b.c$.

```
primes(j) <= integers(2, i), sift(i, j).
sift(i, j) <=
   if #i = [p . I] then
      #j = [p . J], filter(I, p, r), sift(r, J)
   else
      fail.
filter(i, p, r) <=
   if #i = [n . I] then
      if mod(#n, ##p) = 0 then
         filter(#I, ##p, ##r)
      else
         #r = [n . R], filter(I, #p, R)
   else
      fail.
integers(x, [x . l]) <= integers(x + 1, l).
outstream(l) <=
   if #l = [x . L] then
      write(x); outstream(L)
   else
      fail.
```

Let E be the environment after the above expressions are evaluated, i.e. the above functions are defined. If the expression

```
primes(i), outstream(i)
```

is evaluated under E and the computation is fair (i.e. every evaluable subexpression is reduced after a finite steps of the reduction), a sequence of prime numbers is printed without termination. □

Note that if-part of if-then-else works as the synchronization mechanism, which is similar to that of Parlog [1] and that of Concurrent Prolog in a sense. Though our first motivation was not to invent such a synchronization mechanism but to find the natural semantics, we obtained both as a result.

5 FORMAL SEMANTICS

Here, we define a completely formal semantics of Qute. First we give some definitions and then define the semantics.

5.1 definitions

An element of the free monoid $\{0, 1\}^*$ is called a *path*. We use σ, τ etc. to denote paths. The *empty path* (i.e., empty word) will be denoted by Λ. For a path σ and sexps s, u, we define $s_\sigma[u]$ as follows:

$$
\begin{aligned}
&s_\Lambda[u] \equiv u\\
&(s\ .\ t)_{0\sigma}[u] \equiv (s_\sigma[u]\ .\ t)\\
&[s\ .\ t]_{0\sigma}[u] \equiv [s_\sigma[u]\ .\ t]\\
&(s\ .\ t)_{1\sigma}[u] \equiv (s\ .\ t_\sigma[u])\\
&[s\ .\ t]_{1\sigma}[u] \equiv [s\ .\ t_\sigma[u]]
\end{aligned}
$$

Informally, $s_\sigma[u]$ means the result of substituting u for the subsexp of s which can be reached from the root of s by following the path σ. (Here, the character 0 (1) in the path σ means to take the left (right, resp.) subtree.)

We will sometimes regard a path σ as a sexp by the following identification:

Λ is identified with $*$
0σ is identified with $[0\ .\ \sigma]$
1σ is identified with $[1\ .\ \sigma]$

A sexp of the form ("gvar" . (s . σ)) where s is a string and σ is a path, is also called a *global variable*. For a variable $x \equiv$ ("var" . s) and a path σ, x_σ will denote the global variable ("gvar" . (s . σ)).

For an expression a and a path σ, we define an expression $a{\downarrow}\sigma$ by $a{\downarrow_0}\sigma$ where $a{\downarrow_n}\sigma$ $(n \geqq 0)$ is defined as follows:

$a \equiv \#^k x$ where x is a pure variable $\Rightarrow$

$$
a{\downarrow_n}\sigma \equiv \begin{cases} a & \text{if } k<n \\ x_\sigma & \text{if } k=n \\ \#^{k-1}x & \text{if } k>n \end{cases}
$$

a is a global variable $\Rightarrow$ $a{\downarrow_n}\sigma \equiv a$
a is a symbol $\Rightarrow$ $a{\downarrow_n}\sigma = a$
$*{\downarrow_n}\sigma \equiv *$
$[a\ .\ b]{\downarrow_n}\sigma = [a{\downarrow_n}\sigma\ .\ b{\downarrow_n}\sigma]$
$(a,\ b){\downarrow_n}\sigma = (a{\downarrow_n}\sigma),\ (b{\downarrow_n}\sigma)$
$(\lambda a\ .\ b){\downarrow_n}\sigma = \lambda(a{\downarrow_{n+1}}\sigma).(b{\downarrow_{n+1}}\sigma)$
$(a;\ b){\downarrow_n}\sigma = (a{\downarrow_n}\sigma);\ (b{\downarrow_n}\sigma)$
(if a **then** b **else** c**)**$\downarrow_n\sigma$
$\equiv$ **(if** $a{\downarrow_{n+1}}\sigma$ **then** $b{\downarrow_{n+1}}\sigma$ **else** $c{\downarrow_n}\sigma$**)**
$(\neg a){\downarrow_n}\sigma \equiv \neg\ (a{\downarrow_n}\sigma)$
$(a\ b){\downarrow_n}\sigma = (a{\downarrow_n}\sigma)\ (b{\downarrow_n}\sigma)$
$(a = b){\downarrow_n}\sigma = (a{\downarrow_n}\sigma) = (b{\downarrow_n}\sigma)$

$\downarrow$ is used in reducing application. (See the rule below.) $\downarrow$ is a formal definition of the renaming of an expression which we explained in section 4.2.

Example

$$((\lambda x.x = \#y)y)\downarrow_{10} \equiv (\lambda x.x = y_{10})y_{10}$$
$$(\textbf{if } \#x = [u.v] \textbf{ then } foo(\#x, \#y.v) \textbf{ else } y)\downarrow_{11}$$
$$\equiv (\textbf{if } x_{11} = [u.v] \textbf{ then } foo(x_{11}, y_{11}, v) \textbf{ else } y_{11})$$

□

We redefine Σa as a set of paths so that the occurrence of a subexpression can be indicated explicitly.

If p is $*$, a symbol, a variable or a function then $\Sigma p = \phi$

$$\Sigma[a\ .\ b] \equiv 0\Sigma a \cup 1\Sigma b$$
$$\Sigma(a, b) \equiv 10\Sigma a \cup 11\Sigma b$$
$$\Sigma(a; b) \equiv \begin{cases} 10\Sigma a & \text{if } \Sigma a \neq \phi \\ 11\Sigma b & \text{if } \Sigma a \equiv \phi \text{ and } \Sigma b \neq \phi \\ \{\Lambda\} & \text{if } \Sigma a \equiv \Sigma b \equiv \phi \end{cases}$$
$$\Sigma(\textbf{if } a \textbf{ then } b \textbf{ else } c) \equiv \{\Lambda\}$$
$$\Sigma\neg a \equiv \{\Lambda\}$$
$$\Sigma(a\ b) \equiv \begin{cases} \{\Lambda\} & \text{if } \Sigma a \equiv \Sigma b \equiv \phi \\ 10\Sigma a \cup 11\Sigma b & \text{otherwise} \end{cases}$$
$$\Sigma(a = b) \equiv \begin{cases} \{\Lambda\} & \text{if } \Sigma a \equiv \Sigma b \equiv \phi \\ 10\Sigma a \cup 11\Sigma b & \text{otherwise} \end{cases}$$

It is again easy to check that for any expression a, $\Sigma a \equiv \phi$ if and only if a is a pattern. We will call an expression a *redex* if $\Sigma a \equiv \{\Lambda\}$.

Henceforth, we will use the notation $a_\sigma[b]$ only when $\sigma \in \Sigma a$. Furthermore, we will use the notation $a_\sigma(b)$ to denote an expression a such that $a_\sigma[b] \equiv a$. In this case we say that b is an *evaluable subexpression* of a. We note that an evaluable subexpression is a redex.

5.2 semantics

We will call a pair of an environment E and an expression a a *form* and will use the notation $E[\![\ a\ ]\!]$ for it. Moreover we include $\perp$ and ! (which we call *fail* and *suspension*, respectively) as forms. We will use e, f, g etc. to denote forms.

We now define a binary relation $\rhd$ on forms, which represents the reduction step of forms. Actually we define the relation $\rhd_\sigma$ for each path σ. Then the relation $\rhd$ will be defined as $\rhd_\Lambda$. In the following, $\trianglerighteq_\sigma$ denotes the reflexive, transitive closure of $\rhd_\sigma$.

We will call a form e *normal* if $e \rhd f$ holds for no f.

In the following rules, it can be seen that environments are changed only by unification, and the rules of subexpression show how such change of environments affects the evaluation of other subexpressions.

For an environment E and an expression a, we define the environment $E(a)$ as follows. First we let V be the collection of all the free variables appearing either in a or in some equations in E, and we put $F \equiv (E \cup \{\{x\} \mid x \in V\})^*$. Then we define $E(a)$ by:

$$E(a) \equiv (F - \{A \mid A \text{ is simple and } A \in F\}) \cup \{\{A\} \cup A \mid A \text{ is simple and } A \in F\}$$

This kind of an environment is used in the rules of if-then-else and negation so that the marked equations detect suspension. (Recall the definition of the suspended environment in section 3.2.) However, unlike the informal explanation in section 4, the state of suspension cannot be recovered. Therefore, e.g. in evaluating $\neg(x = 0)$, $x = 1$ in the empty environment,

$$\phi[\![\ \neg(x = 0),\ x = 1\ ]\!]$$
$$\rhd_\Lambda \{\{x, 1\}\}[\![\ \neg(x = 0),\ x\ ]\!]$$
$$\rhd_\Lambda \{\{x, 1\}\}[\![\ (),\ x\ ]\!]$$

is the only possible order of the reduction. The rules of suspension appearing in 2, 3, 4, 7 and 8 below show when and how the suspension of a subexpression causes the suspension of the whole expression.

1. subexpression

$$E[\![\ a\ ]\!] \rhd_{\tau\sigma} F[\![\ b\ ]\!] \longrightarrow E[\![\ c_\sigma(a)\ ]\!] \rhd_\tau F[\![\ c_\sigma[b]\ ]\!]$$
$$E[\![\ a\ ]\!] \rhd_{\tau\sigma} \perp \longrightarrow E[\![\ c_\sigma(a)\ ]\!] \rhd_\tau \perp$$

2. list

$$E[\![\ a\ ]\!] \rhd_{\sigma 0} ! \longrightarrow E[\![\ [a\ .\ q]\ ]\!] \rhd_\sigma !$$
$$E[\![\ b\ ]\!] \rhd_{\sigma 1} ! \longrightarrow E[\![\ [p\ .\ b]\ ]\!] \rhd_\sigma !$$
$$E[\![\ a\ ]\!] \rhd_{\sigma 0} !,\ E[\![\ b\ ]\!] \rhd_{\sigma 1} ! \longrightarrow E[\![\ [a\ .\ b]\ ]\!] \rhd_\sigma !$$

3. and

$$E[\![\ a\ ]\!] \rhd_{\sigma 10} ! \longrightarrow E[\![\ a, q\ ]\!] \rhd_\sigma !$$
$$E[\![\ b\ ]\!] \rhd_{\sigma 11} ! \longrightarrow E[\![\ p, b\ ]\!] \rhd_\sigma !$$
$$E[\![\ a\ ]\!] \rhd_{\sigma 10} !,\ E[\![\ b\ ]\!] \rhd_{\sigma 11} ! \longrightarrow E[\![\ a, b\ ]\!] \rhd_\sigma !$$

4. sequential-and

$$E[\![\ p; q\ ]\!] \rhd_\sigma E[\![\ q\ ]\!]$$
$$E[\![\ a\ ]\!] \rhd_{\sigma 10} ! \longrightarrow E[\![\ a; b\ ]\!] \rhd_\sigma !$$
$$E[\![\ b\ ]\!] \rhd_{\sigma 11} ! \longrightarrow E[\![\ p; b\ ]\!] \rhd_\sigma !$$

5. if-then-else

$$E(<a>)[\![\ <a>\ ]\!] \trianglerighteq_\sigma F[\![\ p\ ]\!] \longrightarrow E[\![\ \textbf{if } a \textbf{ then } b \textbf{ else } c\ ]\!] \rhd_\sigma \begin{cases} F[\![\ <b>\ ]\!] & \text{if } F \text{ is not suspended} \\ ! & \text{if } F \text{ is suspended} \end{cases}$$
$$E(<a>)[\![\ <a>\ ]\!] \trianglerighteq_\sigma \perp \longrightarrow$$

$$E[\![\ \textbf{if}\ a\ \textbf{then}\ b\ \textbf{else}\ c\ ]\!] \rhd_\sigma E[\![\ c\ ]\!]$$
$$E(<a>)[\![\ <a>\ ]\!] \cong_e\ ! \Rightarrow E[\![\ \textbf{if}\ a\ \textbf{then}\ b\ \textbf{else}\ c\ ]\!] \rhd_\sigma\ !$$

6. negation

$$E(a)[\![\ a\ ]\!] \cong_\sigma F[\![\ p\ ]\!] \Rightarrow E[\![\ \neg a\ ]\!] \rhd_\sigma \begin{cases} \bot & \text{if } F \text{ is not suspended} \\ ! & \text{if } F \text{ is suspended} \end{cases}$$
$$E(a)[\![\ a\ ]\!] \cong_\sigma \bot \Rightarrow E[\![\ \neg a\ ]\!] \rhd_\sigma E[\![\ ()\ ]\!]$$
$$E(a)[\![\ a\ ]\!] \cong_\sigma\ ! \Rightarrow E[\![\ \neg a\ ]\!] \rhd_\sigma\ !$$

7. application

$$E[\![\ p\ q\ ]\!] \rhd_\sigma \begin{cases} E[\![\ a\downarrow\sigma = q;\ b\downarrow\sigma\ ]\!] & \text{if } p_E \equiv (\lambda a\,.\,b) \\ ! & \text{if } p_E \text{ is a variable} \\ \bot & \text{otherwise} \end{cases}$$
$$E[\![\ a\ ]\!] \rhd_{\sigma 10}\ ! \Rightarrow E[\![\ a\ q\ ]\!] \rhd_\sigma\ !$$
$$E[\![\ b\ ]\!] \rhd_{\sigma 11}\ ! \Rightarrow E[\![\ p\ b\ ]\!] \rhd_\sigma\ !$$
$$E[\![\ a\ ]\!] \rhd_{\sigma 10}\ !,\ E[\![\ b\ ]\!] \rhd_{\sigma 11}\ ! \Rightarrow E[\![\ a\ b\ ]\!] \rhd_\sigma\ !\ .$$

8. unification

$$E[\![\ p = q\ ]\!] \rhd_\sigma \begin{cases} \bot & \text{if } F \text{ is inconsistent} \\ F[\![\ p\ ]\!] & \text{otherwise} \end{cases}$$
where $F \equiv (E \cup \{\{p, q\}\})^*$
$$E[\![\ a\ ]\!] \rhd_{\sigma 10}\ ! \Rightarrow E[\![\ a = q\ ]\!] \rhd_\sigma\ !$$
$$E[\![\ b\ ]\!] \rhd_{\sigma 11}\ ! \Rightarrow E[\![\ p = b\ ]\!] \rhd_\sigma\ !$$
$$E[\![\ a\ ]\!] \rhd_{\sigma 10}\ !,\ E[\![\ b\ ]\!] \rhd_{\sigma 11}\ ! \Rightarrow E[\![\ a = b\ ]\!] \rhd_\sigma\ !$$

We now have the following theorems which capture basic properties of reduction processes.

Theorem 4.

If $e \cong f_1$ *and* $e \cong f_2$ *then* $f_1 \cong g$ *and* $f_2 \cong g$ *for some* g.

Theorem 5.

If $e \cong f_1$ *and* $e \cong f_2$ *and* f_1 *and* f_2 *are both normal then* $f_1 \equiv f_2$.

Theorem 6.

A form e *is normal if and only if*

(i) $e \equiv \bot$,

(ii) $e \equiv\ !$ *or* $'$

(iii) $e \equiv E[\![\ p\ ]\!]$ *for some environment* E *and pattern* p.

The meanings of these theorems should be clear. We refer to Sato [6] for the proofs of these theorems.

REFERENCES

[1] Clark, K. and Gregory, S., 1984: PARLOG: Parallel Programming in Logic, Research Report DOC 84/4, Imperial College.

[2] Gordon, M., Milner, R., and Wadsworth, C., 1979: *Edinburgh LCF, Lecture Notes in Computer Science 78*, Springer-Verlag.

[3] Jaffar, J., 1984: Efficient Unification over Infinite Terms, *preprint*, Monash University.

[4] Lassez, J-L. and Maher, M.J., 1984: *The Semantics of Logic Programs*, Oxford University Press, *in preparation*.

[5] Martelli, A. and Montanari, U., 1982: An Efficient Unification Algorithm, *ACM Transaction on Programming Language and System*, 4, 258-282.

[6] Sato, M., 1984: Theory of Symbolic Expressions, III, *in preparation*.

[7] Sato, M. and Sakurai, T., 1983: Qute: A Prolog/Lisp type language for logic programming, *Proceedings of the Eighth International Joint Conference on Artificial Intelligence*, 507-513.

[8] Sato, M. and Sakurai, T., 1984: Qute: A Natural Amalgamation of Prolog and Lisp, *Journal of Future Generation Computer Systems*, North Holland, *to appear*.

[9] Shapiro, E.Y., 1983: A Subset of Concurrent Prolog and Its Interpreter, ICOT Technical Report, TR-003.

[10] Shapiro, E.Y. and Takeuchi, A., 1983: Object Oriented Programming in Concurrent Prolog, *New Generation Computing*, 1, 25-48.