ICOT Technical Report: TR-245

TR-245

Proof Parameterization Method in Constructive Logic

by

Y. Takayama

March, 1987

ICOT

Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan

(03) 456-3191～5
Telex ICOT J32964

Institute for New Generation Computer Technology

January 27, 1987

Proof Parameterization Method in Constructive Logic

Yukihide Takayama

Institute for New Generation Computer Technology
21F, Mita Kokusai Building
1-4-28 Mita, Minato-ku, Tokyo 108 Japan
takayama%icot.jp@csnet-relay
[inria,kddlab,mit-eddie,ukc]!icot!takayama

1. Introduction

The idea of programming in constructive logic [Beeson 86] is as follows: Given a theorem with the following form,

For all X : TYPE1 there exists Y : TYPE2

such that F(X,Y)

The proof of this theorem should be given by human beings, and must be a CONSTRUCTIVE proof. 'Constructive' means, roughly speaking, that one should not use the refutation to prove the existential property: one must show how the required object can be built up by finite steps of fundamental operation that can be implemented on computers. After the constructive proof is given, the program that has the theorem as its SPECIFICATION will be derived through the procedure called 'the proof compilation algorithm based on the realizability interpretation' [Takayama 87]. The code extracted through the proof compiler is called 'realizer' or 'realizer code'. The correctness of the derived programs can be assured by the proof checker system that automatically checks the proofs of the theorems [Sakai 86].

Constructive proof strategies reflect the logical structure of algorithms well, so that constructive proofs of theorems can be referred to as good program specifications, and theorems can be referred to as good specifications of software functions. Thinking on the proof strategy for a theorem corresponds to the design stage of the algorithm which satisfies a specification so that the proof strategy depends mainly on the creativity of

programmers. For that reason, a full automatic theorem proving technique is not necessary. What is necessary is a proof checking technique which has theorem proving facilities to fill the trivial inference step in the proofs automatically.

This paper extends the programming in constructive logic by introducing some metalogical control mechanisms. The motivation is to make writing the generate and test type, or exhaustive search type, programs easier in the paradigm of 'writing programs as proof of formal specifications'. Section 2 outlines the proof compilation algorithm. Section 3 investigates a prime number generator program. Some difficulties in writing prime number generator programs in constructive logic are pointed out, and the reason why PROLOG but not constructive logic makes it easy to write such programs is discussed. Section 4 introduces a method to parameterize proof trees and modified proof compilation procedure to overcome the problem pointed out in section 3. Section 5 again investigates the prime number generator program in the extended framework defined in section 4. OJ [Sato 86], typed logical system, is used throughout this paper as the background formal system of constructive logic.

2. Outline of Proof Compilation

The proof compiler analyzes a given proof tree from bottom to top, extracting the code step by step for the inference rule attached to each node of the proof tree.

The code generating algorithms for every inference rule of Gentzen type natural deduction system are shown below. Algorithms for inference rules of induction and if-then-else Elimination rule in OJ are omitted since they are not used in this article. The same algorithm is shown in [Takayama 87].

<< Notations >>

Ext(Conclusion, Rule) is the procedure which extracts realizer codes from the proof tree whose conclusion is 'Conclusion', and it is concluded by

'Rule'. Ext(Formula1/Formula2, *) means that when the realizer code of Formula2 is needed in the procedure, Ext(Formula1,*), then the realizing variable sequence of Formula2 will be used instead of the realizer code.

If a:o1 (this means "term a has type o1") and b:o2, then a, b is a term of type o1 x o2. p0 and p1 are the projection function; p0(a, b) = a and p1(a, b) = b. For every provable formula, there is a realizer sequence that realizes the formula. The length of the realizer sequence can be determined systematically by the structure of the formula. Realizing variable of the formula is the variable for which the realizer code of the formula is to be substituted. Rv(Formula) denotes the realizing variable sequence of 'Formula'. A[Rv(Formula) <- ***] means substitution of code sequence *** for Rv(Formula). Apply(Term1, Term2) means the application of lambda term Term1 to Term2, i.e., Term1(Term2).

```
  .       .
  .       .
----(*1)  ----(*2)
  A         B
-----------------(/\-I)
    A /\ B
```

Ext(A/\B, /\-I) == Ext(A,*1), Ext(B,*2)

```
    .                                   .
    .                                   .
--------(*)                         ---------(*)
 A /\ B                              A /\ B
-----------(/\-E)                   --------(/\-E)
   A                                   B
```

Ext(A,/\-E) == p0(Ext(A/\B,*)) Ext(B,/\-E) == p1(Ext(A/\B,*))

```
   .                                   .
   .                                   .
 ------(*)                         --------(*)
   A                                   B
--------(\/-I)                     ---------(\/-I)
 A \/ B                              A \/ B
```

Ext(A\/B,\/-I) ==
    Index of A, Ext(A,*)

Ext(A\/B,\/-I) ==
    Index of B, Ext(B,*)

Caution: 'Index of A', for example, is a suitable term which indicates the position of A in A\/B. The index may be one of terms 'L' and 'R' of type '2', or more generally, natural numbers can be informally used.

```
          [ A ]     [ B ]
  .         .         .
  .         .         .
----(*1)  ---(*2)   ---(*3)
A\/B        C         C
-----------------------(\/-E)
            C
```

```
Ext(C,\/-E) == if p0( Ext(A\/B,*1) ) = Index of A
                    then Ext(C/A,*2) [ Rv(A) <- p1( Ext(A\/B,*1) ) ]
                    else Ext(C/B,*3) [ Rv(B) <- p1( Ext(A\/B,*1) ) ]
```

```
       [ A ]
         .
         .
       -----(*)
         B
     ----------(->-I)
       A -> B
```

```
Ext(A->B,->-I) == lambda [Rv(A)]. Ext(B/A,*)
```

```
  .           .
  .           .
---(*1)    ------(*2)
 A         A -> B
-----------------(->-E)
          B
```

```
Ext(B,->-E) == Apply( Ext(A->B,*2), Ext(A,*1) )
```

```
   [ X : Type ]
       .
       .
     -----(*)
     A(X)
 -----------------(All-I)
 All X:Type. A(X)
```

```
Ext(All X:Type. A(X), All-I)
     == All X:Type. Apply( lambda [X]. Ext(A(X),*) , X)
```

```
   .             .
   .             .
 ---(*1)    -----------(*2)
   t          All X.A(X)
------------------------(All-E)
          A(t)                          where 't' is term
```

```
Ext(A(t), All-E) == Ext(All X.A(X),*2) [ X <- t ]
```

```
  .               .
  .               .
```

```
---(*1)      --------(*2)
 t             A(t)
---------------------(Exist-I)
   Exist X.A(X)
```

Ext(Exist X.A(X), Exist-I) == t, Ext(A(t), *2)

```
                      [ t, A(t) ]
     .                   .
     .                   .
------------(*1)       -----(*2)
Exist X.A(X)             C
--------------------------------(Exist-E)
                C
```

Ext(C, Exist-E) == Ext(C/[A(t)],*2) [Rv(A(t)) <- pl(Ext(Exist X.A(X),*1)]

<< Other Inference Rules >>

```
     -----(*)
       A
```

Ext(A,*) == {nil}

3. Generator Programs

3.1 Prime Number Generator in Constructive Mathematics

Realizability interpretation is effective in the program derivation when the constructive existence proof is given. The meaning of this feature will be made clear if the modified 'Eratosthenes Algorithm' that find prime number less than 100 is investigated. The constructive proof which corresponds to this algorithm will be shown later, but this proof cannot be referred to as a 'proof of existence of prime number'.

In the following, 'N:nat' is a type declaration that means 'N has a natural number type', and type declaration will be often omitted for brevity.

Theorem:

Exist N:nat. two_digit_prime(N)

where two_digit_prime(N) == two_digit_nat(N) /\ sieve(N)

two_digit_nat(N) == digit(N1) /\ digit(N2) /\ N = 10*N1 + N2

```
sieve(N) == N > 1 /\
            All K:nat, All L:nat.( N = K*L -> K = 1 \/ K = N)

digit(N) == N = 0 \/ N = 1 \/ N = 2 \/ N = 3 \/ N = 4
          \/ N = 5 \/ N = 6 \/ N = 7 \/ N = 8 \/ N = 9
```

Proof:

In the proof tree below, inference rules on arithmetic/terms [Sato 86] are abbreviated as (*), (**), (-) and ($$). (=) denotes the rules of equalities. This convention will be held throughout subsequent descriptions of proof trees.

```
                                                        [t=K*L, K:nat, L:nat]
                                                       ------------------
                                                            .......
                                                             @@@
                                                       ------------------(\/-I)
      ------(*)    -------(*)                             K=1 \/ K=t
        n1           n2                                 ------------------(->-I)
      ------(=)    -------(=)                            t=K*L
       n1=n1        n2=n2                                 -> K=1\/K=t
      ------(\/-I) -------(\/-I)                         ---------------------(All-I)
       n1=0         n2=0                                 All L.(t=K*L
       \/           \/                                         -> K=1\/K=t)
       ..           ..          --------(*)              ---------------------(All-I)
       \/           \/           t=10*n1        ----(*)  All K,L.(t=K*L
       n1=9         n2=9           + n2          t>1           -> K=1\/K=t)
---(**) ----------------------------(/\-I)      ------------------------(/\-I)
 t              two_digit_nat(t)                        sieve(t)
-------------------------------------------------------------------------(Exist-I)
                Exist N. two_digit_prime(N)
```

Note: @@@ is 'K=1' or 'K=t'.

The code sequence extracted through the proof compilation is;

```
t, Index of n1=n1 in n1=0\/...\/n1=9 ,
   Index of n2=n2 in n2=0\/...\/n2=9 ,
   All K:nat. All L:nat.
       Apply( lambda [K,L]. Index of @@@ in K=1\/K=t , (K,L) )
```

't' is the prime number extracted from this proof. Other codes are the information which indicates that 't is actually a prime number'. Notice that in the framework of constructive logic, 't' must be an individual prime number, for example '31', and in this case, n1 and n2 are 3 and 1. Index of n1=n1 in n1=0\/..\/n1=9 is 3, and Index of n2=n2 in n2=0\/..\/n2=9 is 1. Consequently the prime number generator program cannot be extracted, although this proof seems to reflect the Eratosthenes algorithm well. This

can be seen as follows:

(1) From the point of view of constructive logic, the proof of existence of prime number is to give one individual natural number and to show that it is actually a prime number; otherwise to show how to construct any prime number - in other words, how to construct the set of prime numbers. The first case corresponds to the proof given above, and the existence of a given prime number. Moreover it note that there is no description in the proof on how to find the prime number. In other words, an algorithm which generates prime numbers cannot be extracted from this proof. For the second case, one must show the general procedure to construct prime numbers above the bar '---(**)', and then realizability interpretation will extract a program that generates prime numbers, however, that proof is likely to be difficult.

(2) Extracting generator type programs requires some kind of mechanism that can treat 'families of proofs' or 'parameterized proof' explicitly. If 't' can be regarded as a parameter in the proof given above it can be referred to as a 'schema of proof' which, if 't' is instantiated to some individual prime number, can become a proof of the theorem. This problem is clear when a prime number generator program written in PROLOG is investigated.

(3) The constructive meaning of logical sentence Exist X:Type. A(X) is a pair 't, p' where 't' is an element of Type, satisfying A(t) and 'p' is a proof of |- A(t). Note that one such 't' is sufficient. In other words, there is no need to obtain all the elements of Type which satisfy A(X). However, generator type programs are the procedure which finds all such elements. Consequently, some extension is necessary in order to treat generator type algorithms naturally in the framework of constructive logic.

3.2 Prime Number Generator in PROLOG

3.2.1 Program Transformation Technique in Logic Programming Language

In logic programming languages like PROLOG [Bowen 83], the prime number generator can be written in a style very similar to that of the theorem given in 3.1. The only difference is the definition of the 'sieve' predicate. Sentences such as that defined in 3.2 cannot be written explicitly in PROLOG mainly because it should be an executable logical formula so that some restriction in description is posed for the purpose of runtime efficiency. To be a PROLOG program, a technique of program transformation such as that seen in [Sato & Tamaki 84] and [Lloyd & Topor 84] must be applied.

```
sieve(N) == N > 1 /\ All K:nat. All L:nat.(N = K*L -> K = 1 \/ K = N)
```

The defining equation symbol '==' is rewritten into the symbol of Horn Clause ':-', and the implication formula is translated into a disjunction formula by the logical equivalence in the classical logic: A -> B <=> ~A \/ B . The conjunction symbol '/\' is rewritten into ','.

```
==>  sieve(N) :- N > 1 , All K:nat. All L:nat. ( ~ N = K*L \/ K = 1 \/ K = N)
```

Use the logical equivalence ~A \/ B \/ C <=> ~(A /\ ~B /\ ~C) in the classical logic and the definition of negation ~A <=> A -> $bottom$.

```
==>  sieve(N) :- N > 1 ,
          All K:nat. All L:nat.( N = K*L , ~ K = 1 , ~ K = N -> $bottom$ )
```

Use the definition of inequation: A =\= B <=> ~ A = B.

```
==>  sieve(N) :- N > 1 ,
       All K:nat. All L:nat.( N = K*L , K =\= 1 , K =\= N -> $bottom$ ) (**)
```

Replace All K:nat.... part by '~q(N)', defined as follows. Note that it is sufficient that K and L be two digit natural numbers because two digit prime numbers are now being investigated.

```
q(N) :- two_digit_nat(K),
        ~ All L:nat. ( N = K*L, K=\=1, K=\= N -> $bottom$ )
```

Using logical equivalences in the classical logic,

```
        ~ All X.A(X) <==> Exist X.~ A(X)
```

and

```
        Exist X.A(X) -> B  <==> All X ( A(X) -> B )
```

q(N) can be transformed as follows:

```
q(N) :- two_digit_nat(K), N = K*L , K=\=1, K=\=N.
```

```
==> sieve(N) :- N > 1, ~q(N).
```

Consequently, the two digit prime number generator program in PROLOG is as follows:

```
sieve(N) :- N > 1, \+ q(N).

q(N) :- two_digit_nat(K), Z is N mod K, Z = 0, K =\= 1, K =\= N.

two_digit_nat(K) :- digit(A),digit(B),
                    X is 10*A , K is X + B.

digit(0).
digit(1).
digit(2).
digit(3).
digit(4).
digit(5).
digit(6).
digit(7).
digit(8).
digit(9).
```

3.2.2 Interpretation of PROLOG Execution in Proof Trees

Execution of PROLOG programs can be interpreted in the proof tree of Gentzen type natural deduction easily. This does not apply to programs with metalogical control mechanisms and 'negation as failure' programs. The interpretation given here is slightly different from the ordinary one in that the execution of PROLOG is interpreted as the refutation procedures [Lloyd 84].

For inference rules, the constructive version of natural deduction like that of QJ [Sato 86] is used here.

The above program is translated to the set of axioms as follows. In this procedure, all free variables are interpreted as being universally quantified. 'A :- B', 'A,B,...' and 'A ; B ;..' are interpreted as 'B -> A', 'A/\B..' and 'A $\vee$ B $\vee$ ..' respectively as in the transformation shown in the previous section. For 'sieve', (**) in 3.2.1 is used instead of the actual PROLOG definition to avoid difficulties in 'negation as failure'.

```
Ax1: All N:nat. ( two_digit_nat(N) /\ sieve(N)  -> prime(N) )

Ax2: All N:nat. All N1:nat. All N2:nat.
```

```
              digit(N1) /\ digit(N2) /\ N = 10*N1 + N2
              -> two_digit_nat(N)

Ax3: All N:nat. All K:nat. All L:nat.
              N > 1 /\
              ( N = K*L /\ K =\= 1 /\ K =\= N -> $bottom$ )
              -> sieve(N)

Ax4: All N:nat.( N=0 \/ ... \/ N=9 -> digit(N) )
```

When the query ':- prime(N).' is given with variable 'N' free, 'N' is interpreted as being existentially quantified, and execution of the program can be seen as the proof of 'Exist N:nat.prime(N)' using the axioms given above.

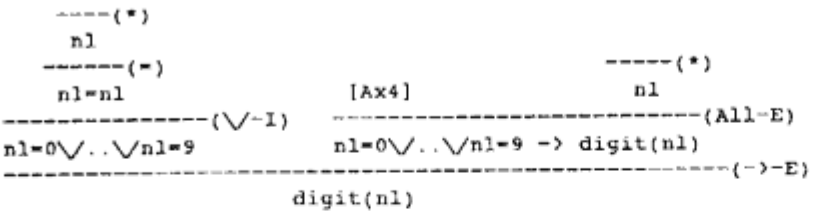
```
                                                          ---(*)
       TREE-1          TREE-2            [Ax1]             t
      ------------------------(/\-I)    ------------------------------(All-E)
        two_digit_nat(t)                 two_digit_nat(t) /\ sieve(t)
        /\ sieve(t)                      -> prime(t)
---(*)  ---------------------------------------------------------------(->-E)
 t                  prime(t)
  ------------------------------------------(Exist-I)
                Exist N:nat. prime(N)
```

<< TREE-1 >>

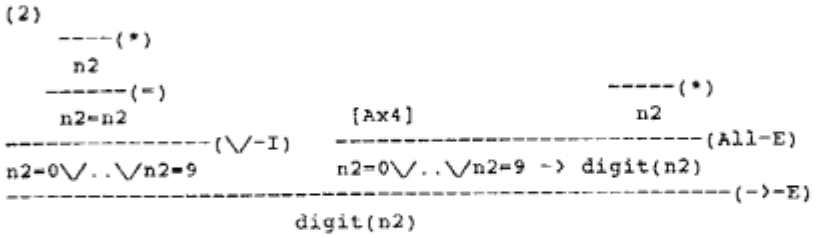
```
                                                    ---(*)
                                          [Ax 2]      t
                                          ----------------(All-E)
                                          All N1.
                                          All N2.
                                            (digit(N1)
                                               /\
                                             digit(N2)
                                               /\
                                             t=10*N1+N2              ---(*)
                                             -> two_digit_nat(t))     n1
                                          ------------------------------(All-E)
                                          All N2.
                                            (digit(n1)
                                               /\
                                             digit(N2)
                                               /\
                                             t=10*n1+N2              ---(*)
                                             -> two_digit_nat(t))     n2
                                          ------------------------------(All-E)
                                                      digit(n1)
                                                         /\
                                  ----------(*)       digit(n2)
   (1)              (2)           t=10*n1+n2             /\
 ------------------------------------(/\-I)             t=10*n1+n2
       digit(n1)/\digit(n2)/\t=10*n1+n2                 -> two_digit_nat(t)
       ---------------------------------------------------------------------(->-E)
                          two_digit_nat(t)
```

where subtree (1) and (2) are as follows:

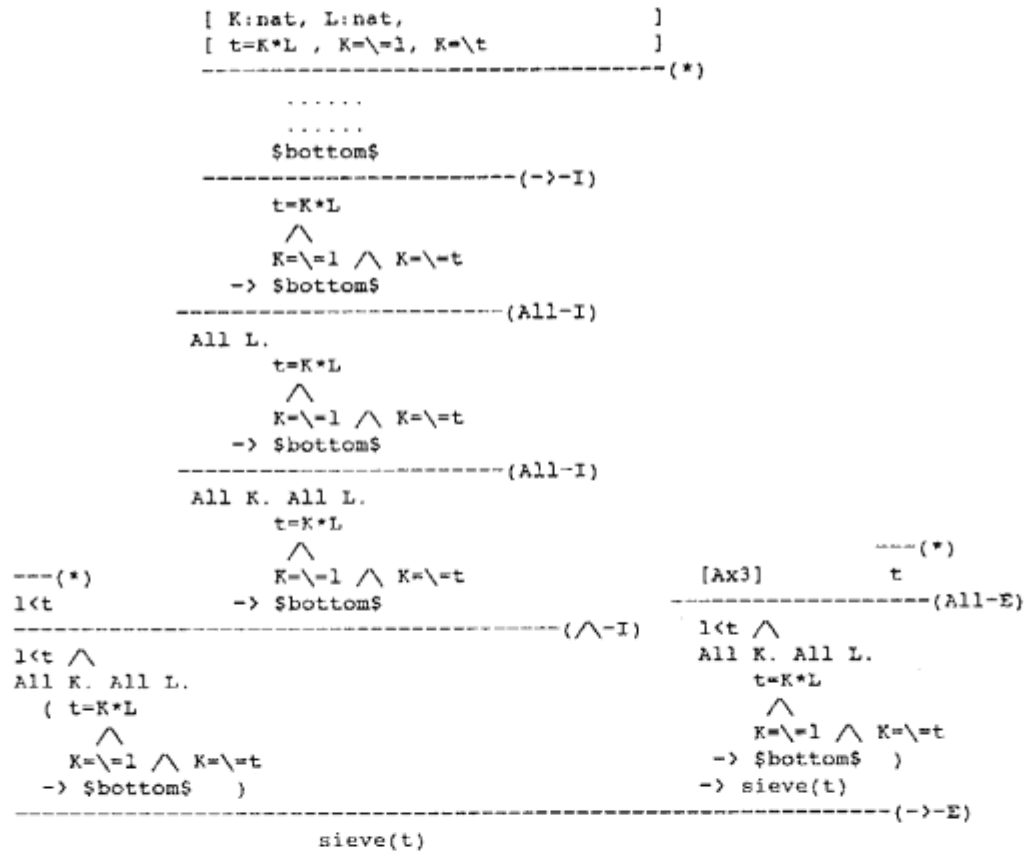
(1)



(2)

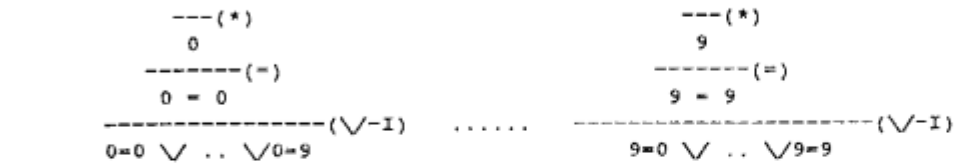


<< TREE-2 >>

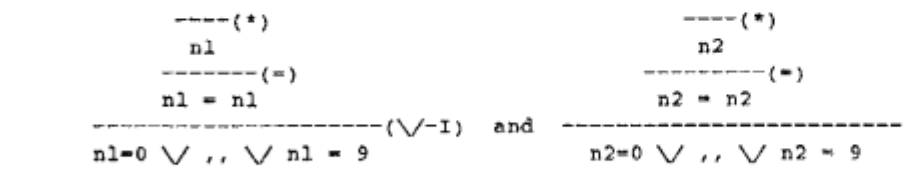


As in the proof in 3.1, t must be a individual natural number. This proof tree is basically the same as that given in 3.2. The main difference is that propositions such as 'sieve' and 'two_digit_nat' are inferred by the (->-I) rule.

PROLOG system performs Gentzen style natural deduction automatically using specification sentences as axioms. For inference of (⋁-I) as in (1) and (2) in the above proof tree, the system finds alternatives to the proofs



as



part subtree by backtracking as in sequential PROLOG, or concurrently as in parallel PROLOG. In any case, more than one proof will be obtained when the specification is executed by the PROLOG system. What PROLOG programmers give is, in a way, not a specification and its proof but a specification and the schema of its proofs. This allows the system to find as many proofs as possible, and consequently allows programmers to write the prime number generator program.

4. Parameterized Proof in Constructive Logic

If incomplete constructive proof that shows the schema or 'guideline' to make complete proof is allowed in the restricted situations and some modification is given to the proof compilation procedure, it is possible to extract generator programs from the incomplete constructive proofs. This section gives a method of parameterizing proof trees and the modified proof compilation technique.

4.1 Proof Parameterization Method

This section describes the notion of proof parameter, parameterization by free variable and extended proof compilation algorithm.

4.1.1 Proof Parameter

Assume there is an application of (\/-I) rule of the following form in the proof tree:

```
P |
--+
                #[ ti ]#
                 X = ti
        ---------------------------------------(\/-I)
        X = t1 \/ .. \/ X = ti \/ .. \/ X = tn
                  .....
```

where X is a variable of the natural number type, and t1, .. ,tn are natural numbers. Let all occurrences of ti above the (\/-I) rule application be rewritten by a variable $$. Then the proof tree will change as follows:

```
P'|
--+
                #[ $$ ]#
                 X = $$
        ---------------------------------------(\/-I)
        X = t1 \/ .. \/ X = ti \/ .. \/ X = tn
                  .....
```

Proof tree P' is called a "parameterized version of proof tree, P". $$ can take the value t1, ..., or tn.

4.1.2 Parameterization by Free Variables

Assume some constructive proof is obtained by the application of the (Exist-I) rule.

```
                          #[ t ]#
        --------(*)
         t:Type             A(t)
        ----------------------------(Exist-I)
            Exist X:Type. A(X)
```

The method of parameterization by free variables is to replace all the terms 't' occurring above the application of (Exist-I) by a free variable, for example N, and omit the application of (Exist-I). The above proof tree is translated by this method as follows:

```
                #[ N ]#

                  A(N)
```

4.1.3 Extended Proof Compilation

Extended proof compilation is almost the same as that given in 2, except in the parameterized application of (\/-I) rule. To distinguish it from ordinal (\/-I) rule, it will be written as (\/-I)* in the following description.

```
          ....
        ----------(*)
          X = $$
   -------------------------(\/-I)*
    X = t1  \/ .. \/ X = tn
```

```
Ext(X=t1 \/ .. \/ X = tn , (\/-I)* )

        == if $$ = t1 then  Index of X = t1 in X=t1\/..\/X=tn   else
           if $$ = t2 then  Index of X = t2 in X=t1\/..\/X=tn   else
                ..........
           if $$ = tn then  Index of X = tn in X=t1\/..\/X=tn
            else            $bottom$
           ,
           Ext(X=$$, *)
```

Sometimes case-sentence will be used in the following description instead of if-then-else terms:

```
        case(X, A1;B1, ... , An;Bn )      (1=<n)
```

which has the same meaning as

```
        if X = A1 then B1  else
           ........
        if X = An then Bn  else    $bottom$
```

4.2 Operational Semantics

In [Sato 86], the operational semantics of Quty are given. Quty is the typed logical language which has both logic and functional programming features. Quty is also designed so as to contain realizer codes extracted from constructive proofs written in QJ as its subset.
This section outlines the operational semantics of Quty following [Sato 86], and gives its extension in order to explain the computational meaning of the realizer code in the extended framework defined in 4.1.

4.2.1 Outline of Operational Semantics of Quty

Program and term are regarded as the same in QJ.

Definition 1: [Program/Term]

1) (typed) variable

2) lambda expression

3) if-then-else term

4) T (top), $bottom$ (bottom)

5) equality of terms

6) application of lambda expressions to terms : TERM1(TERM2)

7) conjunction of terms : TERM1, TERM2 and TERM1 ∧ TERM2

8) inl(TERM) and inr(TERM)

This terms are used to express natural numbers, list and so on.

9) Exist X:Type. TERM

10) Other

The execution of programs is defined by the reduction model. Programs on which reduction procedure can no longer be applied are called canonical programs.

Definition 2: [Canonical Program]

1. Variable X is a canonical program.

2. T (top) is a canonical program.

3. Conjunction of terms: TERM1, TERM2 is a canonical program

where both TERM1 and TERM2 are canonical.

4. inl(TERM) and inr(TERM) are canonical programs

where TERM is canonical.

5. lambda [X].A is a canonical program

where A is not always canonical.

Definition 3: [Environment]

1. T is an environment.

2. X ∧ E is an environment

where E is an environment, and X is a variable.

In this case, variable X is called a 'member of the environment'.

3. p = q ⋀ E is an environment

where E is an environment, both 'p' and 'q' are canonical programs, and every free variable of 'p' and 'q' is a member of E.

"Environment" means the execution environment of programs defined by variables and equations. Correspondence between programs and their execution environment is given as the notion of 'covering', and execution of program within the environment is defined by the rewrite rules of 'e-forms'.

Definition 4: [Covering]

Let E be an environment, and p be a program (not necessarily canonical). Then

E covers p <=[def]=> every free variable of p is a member of E

Definition 5: [e-form]

Let E be an environment, and 'a' be a program covered by E. Then

1) Environment E/⋀a is called 'e-form' and denoted as E[a]

2) If E[a] is an e-form, then Exist X1, .. Xn. E[a] (n>=0) is an e-form

3) If Exist X1, .. , Xn.E[a] is an e-form, variable X is a member of E and X is distinct from X1, .., Xn, then Exist X,X1, .. Xn. E[a] is an e-form.

Variables that are members of the environment must be existentially quantified in the e-form if the variables are existentially quantified in 'a'. The definition of rewrite rules of e-forms is omitted here. Note that e-forms are program in the meaning of QJ.

4.2.2 Operational Semantics of Realizer Code

There are three problems in applying the operational semantics of Quty to that of the realizer code in the extended framework.

(1) How or when is the environment of the realizer code made?

(2) How should the computational meaning of the realizer code extracted form

(All-I) rule application, that is, a universally quantified function closure be explained? (See 2)

(3) How should the computational meaning of the realizer code extracted from the parameterized proof be explained?

The following subsections give solutions to these problems.

4.2.2.1 Making the Environment

'Environment' in the operational semantics of Quty can be seen as the constraint of all free variables occurring in programs to be executed. The environment of the realizer code is built up during the proof compilation procedure. At the initial stage, the environment is T, and when the code containing free variables is generated, the environment is updated, adding suitable terms into it. Free variables can be introduced in the proof compilation procedure when it works on applications of (All-E) and (Exist-I) and, in the extended framework of parameterized proofs, (\/-I)".

(1) Application of (All-E) and (Exist-I)

```
      .
      .
     ---(*)
      t          All X. A(X)
    ------------------------(All-E)
             A(t)

               Ext( A(t) , All-E ) == Ext( All X.A(X) )[ X <- t ]

      .
      .
    ----(*)         ------(#)
      t               A(t)
    -------------------------(Exist-I)
          Exist X. A(X)

               Ext( Exist X.A(X) , Exist-I ) == t, Ext( A(t), # )
```

Free variables are contained in the realizer code if 't' has free variables. If so, add all the equations and inequations above

```
                    .
                    .
                  -----(*)
                    t
```

which contain 't' should be added to the environment.

(2) Application of (⋁-I)"

```
          .
          .
       ----------(*)
          X = $$
     -----------------(⋁-I)"
       X = a ⋁ X = b

          Ext( X=a⋁X=b , (⋁-I)" )
               == if $$ = a then 'L'
                  else if $$ = b then 'R'
                       else '$bottom$'    ,  Ext( X = $$ , * )
```

The proof parameter $$ is free in the realizer code. In this case, all the equations and inequations above the node

```
          .
          .
       ----------(*)
          X = $$
```

containing X = $$ should be added into the environment.

(3) Proof parameterized by free variables

If the proof is parameterized by free variables, all the equations and inequations containing the free variables occurring in the proof tree should be added to the environment.

(4) 'Colored' sub-environment

During the procedure defined in (1), (2), and (3), equations and inequations to be added to the environment should be 'colored' if they occur above the application of (->-I) rules. Set of equations and inequations with the same color makes a sub-environment.

In general, if the evaluation of an environment derives '$bottom$', the computation of the realizer code should be regarded as 'fail'. However, if '$bottom$' comes from within one particular 'colored' sub-environment, the computation of the realizer code should be regarded as 'successful'. This corresponds to the semantics of A -> B formula in which A -> B is interpreted as true if A is false.

4.2.2.2 Computational Meaning of Universally Quantified Code

All X:Type. Apply(lambda [x]. Ext(A(x),*) , X) is extracted from the application of the (All-I) rule:

```
          [ X : Type ]
             ...
            ------(*)
             A(X)
        ------------------(All-I)
          All X:Type. A(X)
```

The computational meaning of this code is as follows;

1. Apply beta-reduction to Apply(lambda [x]. Ext(A(x),*) , X) obtaining Ext(A(x) , *) [x <- X]
2. Generate an arbitrary element of type Type 't', and evaluate Ext(A(x) , *) [x <- t] If this code is reduced to $bottom$, the whole computation is regarded as failure.
3. Repeat 2. for all the elements of Type.
4. If procedure 3. succeeds, the whole evaluation of the above code succeeds.

Note that this code can also be seen as a generate and test type program.

4.2.2.3 Operational Semantics around Proof Parameters

Proof parameters are contained both in the extended realizer code and its environment. In the environment, proof parameters are interpreted as existentially quantified.

(1) If the values of proof parameters are determined through the execution of the environment, then the realizer code is ready to be executed.

(2) If both the code and its environment are not ready to be executed, the proof parameters will be 'forced to be instantiated' to possible values to execute the environment. Alternative values must be generated until execution of the environment succeeds.

5. Prime Number Generator in the Extended Framework

5.1 'sieve'

As stated in section 3.1, specification of the program which checks whether a given natural number is prime or not is as follows. Notice that 'N' is a free variable.

```
sieve(N) == 1 < N /\ All K, L.( N = K*L -> K = N \/ K = 1)
```

Proof of the above specification can be written as follows.

```
                    [ N = K*L, K , L ]
                    ------------------(*)   ----------(*)
                          K = N/L             N/L = $$
                         ------------------------------(=)
                                K = $$
                              ---------------(\/-I)
                               K = N \/ K = 1
                  ---------------------------------------------(->-I)
                               N = K*L -> K = N \/ K = 1
                  ---------------------------------------------(All-I)
                      All L.( N = K*L -> K = N \/ K = 1 )
---~~(*)          ---------------------------------------------(All-I)
1< N                All K,L.( N = K*L -> K = N \/ K = 1 )
-------------------------------------------------------------(/\-I)
                  sieve(N)
```

$$ is a proof parameter that can be instantiated to 1 or N, and N is a free variable, so that this is a parameterized proof of the specification.

The extracted realizer code is

```
    All K:nat. All L:nat.
        Apply( lambda [K,L].
                    if $$ = N then 'L'
                    else if $$ = 1 then 'R' else $bottom$ ,
               [K,L] )
```

Environment;

[1 < N , ...

[K = $$, K = N/L , N/L = $$, N=K*L, ...]]

where [K = $$, ..] is a 'colored' sub-environment, N/L is the symbolic expression of quotient of natural number, and '...' denotes other equations and inequations from inference steps abbreviated to (*).

Operational Semantics

* The above codes are executed as follows:

1) Let the value of N be the natural number 'n' to be checked to see whether it is prime or not.

2) Generate new natural numbers k and l, and let them be the values of K and L respectively. If there are no other natural numbers k, l to be newly generated, the whole computation terminates.

3) Evaluate the realizer code in its environment.

3-1) The environment is now as follows:

[1 < n, ...

{ k = $$, k = n/l, n/l = $$, n = k*l,...]]

1 < n and k = n/l can be evaluated at this stage, and then

<< Case 1 >> evaluation of 1 < n fails

The whole computation fails, that is, 'n' is proved to be non-prime.

<< Case 2 >> evaluation of 1 < n succeeds

< case 2-1 > If the evaluation of

[k=$$, k=n/l, n/l=$$, n=k*l, ..] fails,

computation succeeds. Then return to 2).

< case 2-2 > If the evaluation of

[K=$$, k=n/l, n/l=$$, n=k*l, ..] succeeds,

proceed to 3-2).

3-2) Again the environment is

[1 < n, ... ,

[k = $$, k = n/l, n/l = $$, n = k*l, ..)]

and the realizer code is reduced as follows:

if $$ = n then 'L' else if $$ = 1 then 'R' else $bottom$

Now, the if-parts of this code must be evaluated in the environment.

<< Case 1 >> Add '$$ = n' to the environment, and evaluate it.

If no contradiction occurs, then '$$ = n' is true and the realizer code is rewritten to L, then return to 2).

As L is the canonical form in QJ, then the evaluation

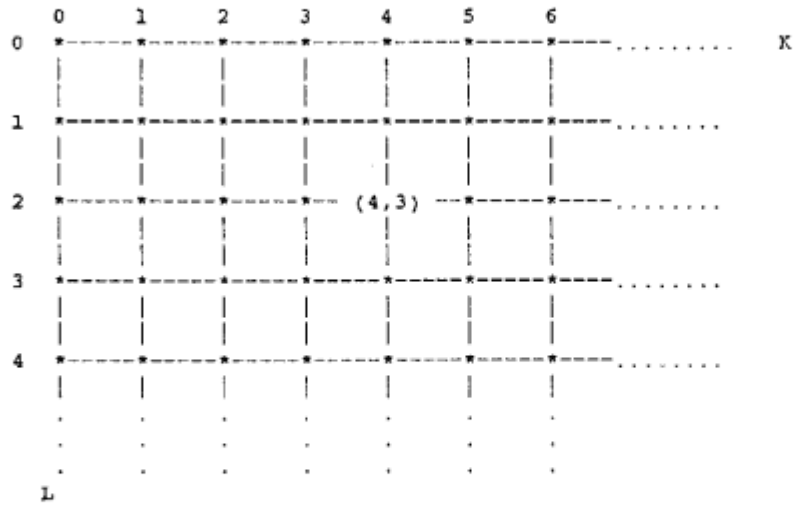
is successful. Otherwise, go to << Case 2 >>.

<< Case 2 >> Perform a similar procedure by adding '$$ = 1' to the environment. If no contradiction occurs when the environment is evaluated, and the evaluation is successful, return to 2). Otherwise, the code is rewritten to $bottom$ and whole computation fails, that is, 'n' is proved to be non-prime number.

As determined from the operational semantics given above, the realizer code extracted from the constructive proof of the 'sieve(N)' specification is the term that, when it is evaluated in the environment created through the proof compilation, will be reduced to '$bottom$ if 'n' is not prime, ~~or~~ otherwise reduced to 'L' or 'R', or the evaluation of its environment will fail in a particular colored sub-environment. The fact that 'n' is not prime is proved also when the environment evaluation fails.

5.2 Parallel Execution of Realizer Code of 'sieve'

The code given in 5.1 can also be executed in parallel naturally. In this case, as K and L are any two natural numbers, it is natural to explain the operational semantics on the lattice as follows:



Each node (K,L) = (k,l) has its code and environment, which can be

executed independently. For example, for node (4,3), its code and environment are as follows:

```
<< Code >>

        Apply( lambda [K,L], if $$ = N then 'L'
                             else if $$ = 1 then 'R' else $bottom$ ,
               [4,3] )

<< Environment >>

       { 1 < n,..,[ 4 = $$, 4 = n/3, n/3 = $$, n = 4*3, ..] }

<< Execution >>
```

In the following, execution is described with the pair

[[Code, Environment]] that has the same meaning as the e-form defined in 4.2.1 and all free variable are interpreted as existentially quantified.

Assume here that N is 2. Then,

```
[[ Apply( lambda [K,L]. if $$ = 2 then 'L' else if $$ = 1 then 'R'
   else $bottom$ ,
   [ 1 < 2,..,[ 4 = $$, 4 = 2/3, 2/3 = $$, 2 = 4*3, ..] ] ]]

-->

[[ if $$ = 2 then 'L' else if $$ = 1 then 'R' else $bottom$ ,
   [ 1 < 2,..,[ 4 = $$, 4 = 2/3, 2/3 = $$, 2 = 4*3, ..] ] ]]

-->

[[ --, [ 1 < 2, .. [ $bottom$ ]] ]]
```

In this case, the environment evaluation fails. Hence this is a meaningless execution although the computation itself is regarded as 'successful'.

Assume that a new N is instantiated to 12. Then,

```
[[ Apply( lambda [K,L]. if $$ = 2 then 'L' else if $$ = 1 then 'R'
   else $bottom$ ,
   [ 1 < 2,..,[ 4 = $$, 4 = 2/3, 2/3 = $$, 2 = 4*3, ..] ]      ]]

-->

[[ if $$ = 12 then 'L' else if $$ = 1 then 'R' else $bottom$ ,
   [ 1 < 12,..,[ 4 = $$, 4 = 12/3, 12/3 = $$, 12 = 4*3, ..] ]      ]]

-->

[[ if $$ = 12 then 'L' else if $$ = 1 then 'R' else $bottom$ ,
   [ 1 < 12, .. [ 4 = $$, 4 = 4, 4 = $$, 12 = 12, ..] ]        ]]
```

At this stage, $$ is determined as 4, so proceed to the computation

of code part.

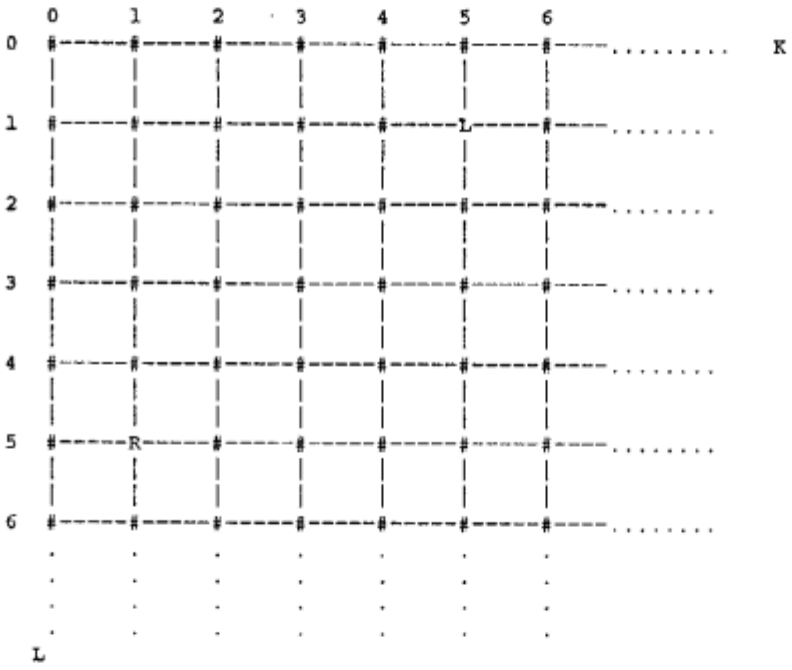
-->

```
[[ $bottom$,
   [ 1 < 12, .. [ 4 = $$, 4 = 4, 4 = $$, 12 = 12, ..] ]          ]]
```

In this case, execution succeeds in $bottom$, so that 12 is proved to be non-prime.

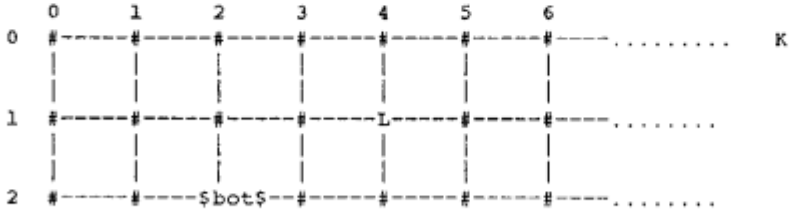
The figures shown below are sample results of parallel execution.
In the following figures, each node is marked as R, L, bot or #.
They indicate the results of local execution at each (k,l) node.
bot means that execution of the code ended in $bottom$; # means that evaluation of the environment ended in '$bottom$'.

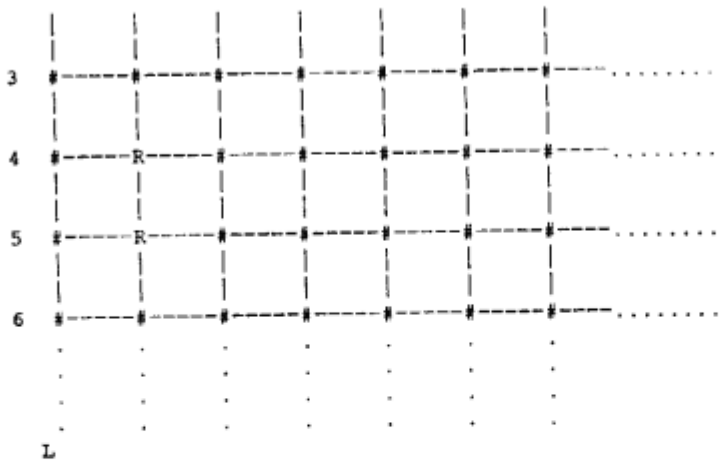
<< Case 1 >> n is 5



As every execution on local nodes ends in L, R or #, it is proved that 5 is a prime number.

<< Case 2 >> N is instantiated to 4





bot is contained in the results, so a check is made, showing that 4 is not a prime number.

Caution:

The execution given so far will not stop. Practically, K and L should be restricted as K =< N and L =< N . This restriction can be written in the specification of 'sieve', and with a minor change of the operational semantics it can be manipulated in the environment. This also indicates that the operational semantics given in 4.2.2.2 are not practical unless types of universally quantified variables are finite.

5.3 'two_digit_nat'

The definition of two digit natural number is as follows:

two_digit_nat(N) == Exist N1, N2. digit(N1) ∧ digit(N2) ∧ N = 10*N1 + N2

The parameterized proof of two_digit_nat(N) is shown below:

```
  -----(*)            ----(*)
   $1                  $2
---------(=<-ref) ----------(=<-ref)
$1 = $1             $2 = $2
---------(∨-I)  ---------(∨-I) --------------(*)
digit($1)          digit($2)       N = 10*$1 + $2
-------------------------------------------------(∧-I) --(*)
      digit($1) ∧ digit($2) ∧ N = 10*$1 + $2           $2
-----------------------------------------------------------(Exit-I) --(*)
    Exist N2. ( digit($1) ∧ digit(N2) ∧ N = 10*$1 + N2              $1
-------------------------------------------------------------------------(Exist-I)
                          two_digit_nat(N)
```

The extended realizer code extracted from the proof tree is

$1, $2, case($1, 0;I_1, ... , 9;I_9), case($2, 0;I_1, ... , 9;I_9)

where I_1, .. , I_9 are indices.

Environment::

{ $1 = $1, $2 = $2, N = 10*$1 + $2, ...}

Note that in this case there is no colored sub-environment.

Operational Semantics:

* The realizer code and the environment given above have the operational semantics both as 'two digit natural number generator' and 'two digit natural number type checker'.

<< Operational semantics as generator program >>

1) Variables both in the code and the environment are not instantiated. In this case, the proof parameters are allowed to be instantiated. In this case, as there are case($1, 0;I_1, ... , 9;I_9) and case($2, 0;I_1, ... , 9;I_9) in the code, $1 and $2 can be instantiated to natural numbers 0, 1, 2, .., 9.

2) Choose any natural numbers, n1 and n2, from the finite set { 0, 1, 2, ..., 9}, and instantiate $1 and $2 with them. Then

Code:

```
n1, n2,
      case(n1, 0:I_1, .. , 9:I_9),
      case(n2, 0:I_1, .. , 9:I_9)
```

Environment:

{ n1 = n1, n2 = n2, N = 10*n1 + n2, ... }

case(...) codes can be reduced to one of indices I_1, .. ,I_9. For the environment the value of N can be obtained by evaluating 10*n1 + n2 .

3) Choose another pair of natural number between 0 and 9, then do the operation given in 2).

In this case, the output value must be that of N, which does not occur in the code part. Some kind of output mechanism is also needed.

<< Operational semantics as type checker program >>

1) First, variable N is instantiated to a individual natural number 't' . Then,

Code:

$1, $2, case($1, 0;I_1, .. ,9:I_9), case($2, 0;I_1, .. , 9;I_9)

Environment:

[$1 = $1, $2 = $2, t = 10*$1 +$2, ...]

2) Choose any natural numbers n1 and n2 from the finite set { 0, 1, 2, ..., 9}, and instantiate $1 and $2 with them.

Then

Code:

n1, n2,

case(n1, 0;I_1, .. , 9:I_9) ,

case(n2, 0:I_1, .. , 9:I_9)

Environment:

[n1 = n1, n2 = n2, t = 10*n1 + n2, ...]

case(...) codes can be evaluated to one of indices I_1, .. ,I_9, that is, reduced to canonical terms.

For the environment, first evaluate 10*n1 + n2 , and check the equality t = value of 10*n1 + n2.

If this check is successful, the whole execution is successful.

Otherwise go to 3).

3) Choose another pair of natural numbers between 0 and 9, then do the operation given in 2).

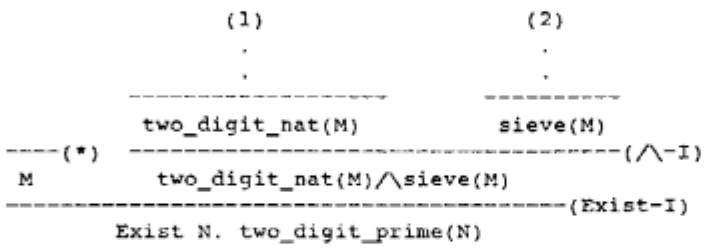
If there is no other choice of pairs, whole execution fails.

5.4 Prime Number Generator

A two digit prime number generator program can be extracted from the proof that is a combination of the proofs in 5.1 and 5.3.

The specification of two digit prime number is the same as that given in 3.1.

However, the proof is different in the meaning that it is parameterized by several proof parameters introduced in 4. In the proof tree shown below, 'M' is a free variable as in 'parameterization by free variable'. (1) and (2) are the proof trees given in 5.3 and 5.1 respectively;



The extracted realizer code is

```
M ,   ---- (a)

$1, $2, case($1, 0;I_1, .. ,9;I_9), case($2, 0;I_1, .. ,9;I_9), ---- (b)

All K:nat. All L:nat.
 Apply( lambda [K,L]. if $$ = M then 'L'
                      else if $$ = 1 then 'R'
                           else $bottom$    , [ K, L ] ) ---- (c)
```

(b) and (c) are the same as those extracted in 5.3 and 5.1.

$1, $2, and $$ are proof parameters. (a) is the code extracted by combining (1) and (2) by (/\-I) and (Exist-I) rules.

The environment of the extracted code is

```
{ $1 = $1,  $2 = $2, M = 10*$1 + $2, ...

  1 < M, [ K = $$, K = M/L, M/L = $$, M = K*L, .. }
```

This is a concatenation of the environments given in 5.1 and 5.3.

The operational semantics of the extracted code is:

(a) and (c) cannot be evaluated unless M is instantiated. Consequently (b) is evaluated first, and $1, $2, and M are instantiated.

After that the environment, (a) and (b) are evaluated. The whole code execution is successful if the evaluation of (c) is successful.

The prime number generated from this code is the value of M.

6. Subsequent Research

This paper investigated a program which generates a finite number of prime numbers for both classical and constructive logic. To allow the exhaustive search routine to be written easily, the proof parameterization method was introduced into constructive logic. If the stream structure [Goto 85] is introduced into this framework, this can be extended enough to manipulate the general prime number generator program. At the same time, several more experimental case studies are necessary to prove the power of the techniques introduced in this paper. These subjects will be the base of further research.

Acknowledgment

I am indebted to Dr.Sato at Tohoku University for reading the draft version of this paper and giving me some fruitful suggestions. This research has also benefited from discussion with Mr. Matsumoto and Mr. Okumura, researchers of the problem solving research group in ICOT, who told me how they approach to the exhaustive search problem in the area of program transformation of logic programs.

References

[Bowen 83] Bowen, D.L.,et al."DECsystem-10 PROLOG USER'S MANUAL", Department of Artificial Intelligence University of Edinburgh, 1983.

[Goto 85] Goto, S.,"Concurrency in Proof Normalization and Logic Programming", IJCAI '85, 1985.

[Lloyd 84] Lloyd, J.W.,"Foundations of Logic Programming", Springer, 1984.

[Lloyd&Topor 84] Lloyd, J.W.and Topor, R.W.,"Making PROLOG More Expressive", Journal of Logic Programming, 1 (3), pp225-240, 1984.

[Sakai 86] Sakai, K.,"Toward Mechanization of Mathematics -- Proof Checker and Term Rewriting System --", France-Japan Artificial Intelligence and Computer Science Symposium '86, 1986.

[Sato 86] Sato, M.,"QJ: A constructive logical system with types", France-Japan Artificial Intelligence and Computer Science Symposium 86, Tokyo, 1986.

[Takayama 87] Takayama, Y., "CAP-QJ : Programming System Based on QJ", 28th Programming Symposium, Hakone, Japan, 1987.

[Sato & Tamaki 84] Sato, T.,& Tamaki, H., "Transformational Logic Program Synthesis", Proceedings of the International Conference on Fifth Generation Computer Systems 1984, ICOT, Japan, 1987.