TR-328

# Proof Compiling Technique based on Realizability and Proof Normalization

by
Y. Takayama

November, 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

Proof Compiling Technique

based on

Realizability and Proof Normalization

Yukihide Takayama

Institute for New Generation Computer Technology

4-28, Mita 1-chome, Minato-ku, Tokyo 108 Japan

takayama@icot.jp

## 1. Introduction

Constructive proofs can be seen as a very high level description of algorithm, as has been discussed by many computer scientists and logicians in terms of the relationship between intuitionism and computation, and programming systems such as the Nuprl system [Constable 86] and the PX system [Hayashi 87] have been implemented. These systems have the facility to extract executable codes from constructive proofs, and are based on the notion of formulae-as-types or realizability interpretation. However, there is still room for refinement of the facility from a more practical point of view.

Generally speaking, compiler systems used in practical situations should have the following functions:

1) correctness checking of programs

2) optimization of programs

3) incremental compilation

These problems can be approached in the following way from the point of view of proofs as programs.

1) is performed by a proof checker system for mathematical reasoning. This topic is our of the range of this paper.

2) is performed at proof level by using the proof normalization method [Prawitz 65].

3) is realized by regarding subroutine calls as references to theorems already proven within proofs of other theorems.

In addition to the facilities listed above, description of user-defined rules of inference and extraction of program schemes corresponding to the rules can be realized by introducing propositional variables and introduction and elimination rules for the second order universal quantifier.

Section 2 gives our formalism of programs and logic and the algorithm of proof compilation. Section 3 gives the method of user-defined rules description and program scheme extraction. Section 4 shows the extraction of a gcd program by using user-defined course of value induction rules. Section 5 outlines the operational semantics of the extracted codes. Section 6 works on the optimization technique based on normalization of proof trees, and introduces another powerful optimization technique, the modified $\vee$-code. Section 7 deals with the method of incremental compilation, and the conclusion is given in Section 8.

## 2. Proof Compiler

This section gives the formalization and the naive version of program the extraction algorithm based on the notion of realizability. They are based on a subset of the 1985 version of QJ [Sato 85], except for the higher order features, propositional variable and second order all-I/E rules.

### 2.1 Notational Preliminaries

(1) Type expressions

1) nat

Type expression of natural number.

2) prop

Type expression of proposition. P:prop means that P is a well formed proposition.

3) Type1 -> Type2

The type of function from elements of type Type1 to elements of type Type2.

4) Type1 X .. X Typen

Cartesian product type.

(2) Term expressions

'==' is used to denote definitional equalities throughout the following description.

1) 0, 1, 2, ...

Elements of type 'nat'.

2) X, Y, Z ....

Individual variables are written in capital letters. All the variables have types, and 'X:Type' is read as 'variable X has type Type'. Type declarations are usually omitted in the following description.

3) lambda [X0,..,Xn]. A(X0,..,Xn)      (0=<n)

Lambda abstraction.

4) if A then B else C

A is a higher order equation or inequation defined in (3) 1).

5) left/right

Constants.

6) mu [Z0,..,Zn]. A(Z0,..,Zn)      (0=<n)

'mu' is the fixed point operator.

7) a(b1)(b2)...(bn)

Application. Associate to left.

8) X mod Y

Residue of fraction of X by Y.

9) (TERM1, .., TERMn) or simply TERM1,..,TERMn

Sequence of terms. If TERMi are of types Typei (0=<i=<n), then the above terms are seen to be of the Cartesian product type: Type1 X .. X Typen.

10) proj(i)

Projection function of type

Type0 X ... X Typei X ... Typen --> Typei    (n>0, 0=<i=<n)

11) l(SEQUENCE)

Length of the sequence, SEQUENCE.

12) any[N]

Sequence of arbitrary N codes. There is no notion of any[N] in QJ, but it is introduced mainly for experimental use.

13) succ/pred

Successor/predecessor function on 'nat'.

(3) Formulae

1) TERM1 = TERM2, TERM1 =< TERM2, TERM1 < TERM2

Higher order equation/inequation of terms.

2) P, Q, ....

Propositional variables. These are of type 'prop'.

3) void

Contradiction. Note that ~A == A -> void.

4) P(X), Q(X),...

Propositions that have X as free variables.

P(X) is also denoted in an abstraction formula, Abs [X] P.

Substitution of a term, t, to X that occurs free in P is formulated as a kind of beta-reduction:

(Abs [X] P)(t) --> P(t)

5) Definition of formulae

1. Higher order equation/inequation of terms are (atomic) formulae.

2. void is a formula.

3. If P and Q are formulae,

   a) P/\Q is a formula

   b) P\/Q is a formula

   c) P -> Q is a formula

   d) ~P is a formula

4. If P(X) is a formula containing X as free,

   d) all X:Type1. P(X) is a formula

   e) exist X:Type2. P(X) is a formula

where Type1 can be 'nat' or 'prop' and Type2 is 'nat'.

Formulae are regarded as of type 'prop'.

(4) Proof trees

Proof trees are written in the ordinal natural deduction style. Subtrees are often abbreviated as 'PI' or, when the free variable or individuals in the subtree should be stressed, 'PI(X)'.

## 2.2 Inference Rules on Logical Constants and Equalities

The inference rules used are listed here.

```
A     B           A/\B           A/\B
------(/\I)     ------(/\E)    ------(/\E)
 A/\B              A              B

                                              [A]       [B]
  A               B                A\/B        C         C
-------(\/I)    -------(\/I)     --------------------------(\/E)
 A\/B            A\/B                          C

 [A]
  B                  A    A -> B
--------(->I)      ------------(->E)
 A -> B                 B

 [X:Type]
   A(X)                            t:Type    all X:Type. A(X)
-----------------(all-I)           ---------------------------(all-E)
all X:Type. A(X)                          A(t)


                                                    [x, A(x)]
t:Type          A(t)             exist X:Type.A(X)      C
--------------------(exist-I)    ------------------------------(exist-E)
exist X:Type. A(X)                                 C


          [X:nat, A(X)]
A(0)        A(succ(X))
---------------------(nat-ind)
 all X:nat. A(X)


  void
--------(void-E)
   A


t = s (in Type)  A(t)          t:Type                t = s (in Type)
----------------------(=1)   -----------------(=2)   ----------------(=3)
    A(s)                     t = t  (in Type)        s = t (in Type)


p = q (in Type)    q = r (in Type)
---------------------------------(=4)
        p = r (in Type)
```

QJ also has rules of arithmetic, formation of terms and formulae, and type inferences. However, these rules are not necessary as far as program extraction is concerned. The names of these rules are abbreviated to * in the following description.

The following additional rules are of higher order logic and are not contained in original version of QJ.

```
      [P:prop]
       F(P)
    ----------------(2nd Ord. all-I)
    all P:prop. F(P)


   P:prop    all P:prop. F(P)
  ----------------------------(2nd Ord. all-E)
         F(P)
```

These two rules are allowed to use only in the following situation:

```
                       [P:prop]
                         F(P)
                     ----------------(2nd Ord. all-I)
     Q:prop          all P:prop. F(P)
     --------------------------------(2nd Ord. all-E)
                 F(Q)
```

## 2.3 Program Extraction Algorithm

The program extraction algorithm is given here. This algorithm performs q-realizability interpretation of QJ and the soundness of the extracted code, realizer, is proved in [Sato 85].

(1) Notations for algorithm description

```
       A
Ext(-------(Rule))
       B
```

Top level procedure (function) of program extraction. It is often abbreviated to Ext(B) in the situation where 'A' and 'Rule' are clear. When a conclusion, B, depends on a list of formulae, Gamma, this procedure is denoted by Ext(Gamma|-B).

Rv(A)

Realizing variable sequence. Realizing variables are sequences of variables to which realizer codes for the formula are assigned. Rv(A) is defined as follows:

1) Rv(A) == nil sequence, if A is atomic

2) $Rv(A \wedge B)$ == concatenation of Rv(A) and Rv(B)

3) $Rv(A \vee B)$ == concatenation of a new variable, Z, Rv(A), and Rv(B)

4) Rv(A->B) == Rv(B)

5) Rv(all X:type. A(X)) == Rv(A(X))

6) Rv(exist X:type. A(X)) == concatenation of a new variable, z, and Rv(A)

7) Rv(P) == Rv(P(X)), if P has X as free variables.

@

Substitution. @ is defined as {X0 <- T0, .., Xn <- Tn}, and this means to substitute Ti for Xi (0=<i=<n) that occurs free in a given expression. If A is a term, the application of @ to A is denoted as A@.

pI

Projection function on Cartesian product type.

pI: TYPE-0 X .. X TYPE-I X ... X TYPE-N --> TYPE-I

where

pI(a0, .., aI, .., aN) = aI      (0=< I =< N)

(nil)

Denotes empty code.

(2) Definition of Ext procedure

$$Ext\left(\frac{A \quad B}{A \wedge B}(\wedge I)\right) == Ext(A),\ Ext(B)$$

$$Ext\left(\frac{A \wedge B}{A}(\wedge E)\right) == p0(A \wedge B)$$

$$Ext\left(\frac{A \wedge B}{B}(\wedge E)\right) == p1(A \wedge B)$$

$$Ext\left(\frac{A}{A \vee B}(\vee I)\right) == left,\ Ext(A)$$

$$Ext\left(\frac{B}{A \vee B}(\vee I)\right) == right,\ Ext(B)$$

```
              [A]     [B]
     A\/B      C       C
Ext(------------------------(\/E))
               C

        == Ext(A|-C)[Rv(A) <- p1(Ext(A\/B))]      ; if p0(Ext(A\/B)) = left

           Ext(B|-C)[Rv(B) <- p1(Ext(A\/B))]      ; if p0(Ext(A\/B)) = right

           if left = p0(Ext(A\/B))                ; otherwise
                     then Ext(A|-C)
                     else Ext(B|-C)

      [A]
       B
Ext(--------(->I)) ==  Ext(B)                      ;if Rv(A) = nil
     A -> B
                       lambda [Rv(A)].Ext(A|-B)    ;otherwise


     A    A -> B
Ext(------------(->E)) == Ext(A->B)(Ext(A))
          B


       [X:Type]
         A(X)
Ext(-----------------(all-I)) == lambda [X]. Ext(A(X))
    all X:Type. A(X)


     t:Type    all X:Type. A(X)
Ext(---------------------------(all-E)) == Ext(all X:Type. A(X))(t)
           A(t)


      t:Type          A(t)
Ext(--------------------(exist-I)) == t, Ext(A(t))
      exist X:Type. A(X)


                          [x:Type, A(x)]
    exist X:Type.A(X)           C
Ext(----------------------------------(exist-E))
               C

        == Ext(x:Type, A(x) |- C)@


   where @ = [x <- p0(Ext(exist X:Type. A(X)),
              Rv(A(x)) <- p1(Ext(exist X:Type. A(X)) ]


           [X:nat, A(X)]
    A(0)      A(succ(X))
Ext(----------------------(nat-ind))
      all X:nat. A(X)

        == lambda [X]. if X=0 then Ext(A(0)) else Ext(X:nat,A(X)|-A(succ(X)))

                     if ZZ does not occur in Ext(X:nat,A(X)|-A(succ(X)))

           mu [ZZ]. lambda [X]. if X = 0 then Ext(A(0))
                                     else Ext(X:nat, A(X) |- A(succ(X)))@

                 where ZZ = Rv(A(X)), and @ = [ZZ <- ZZ(pred(X))]
```

```
                               otherwise
     void
Ext(--------(void-E))
       A
        ==  (nil)                ;if Rv(A) is nil sequence

            any[l(Rv(A))]        ; otherwise


     t = s (in Type)  A(t)
Ext(----------------------(=)) == Ext(A(t))
           A(s)


        t:Type
Ext(-----------------(=)) == (nil)
     t = t  (in Type)


    t = s (in Type)
Ext(---------------(=)) == (nil)
    s = t (in Type)


    p = q (in Type)   q = r (in Type)
Ext(-----------------------------------(=)) == (nil)
          p = r (in Type)


      A
Ext(------(*)) == (nil)
      B
```

(3) The Ext procedure for second Order all-I/E rules will be discussed in the next section.

# 3. Proof Schema Using Propositional Variables

## 3.1 Proof of Course of Value Induction

As is well known in mathematical logic, the course of value induction schema

```
all P:prop.
        all X:nat. (all Y:nat. (Y<X -> P(Y)) -> P(X))     [COV-IND]
        -> all Z:nat. P(Z)
```

can be proven by mathematical induction. Any proof tree of the first order theorem that uses the course of value induction rule can be transformed into one that uses the mathematical induction rules as follows. Let Q be some individual proposition. The proof of all X:nat. all Y:nat. (Y<X -> Q(Y)) -> Q(X) will be referred to as course of value proof in the following descriptions.

```
             PI_1

all X:nat. all Y:nat. (Y<X -> Q(Y)) -> Q(X)
-------------------------------------------(course of value induction)
    all Z:nat. Q(Z)
```

```
==(Proof Transformation)==>>
```

COV-TREE:

```
              PI_1

all X:nat. all Y:nat. (Y<X -> Q(Y)) -> Q(X)          SUB_TREE
-------------------------------------------------------------(->E)
                   all Z:nat. Q(Z)
```

SUB_TREE:

```
                                             .
                                             .
                                             .
                all P:prop.
                        all X:nat. (all Y:nat. (Y<X -> P(Y)) -> P(X))
                        -> all Z:nat. P(Z)
                ----------------------------------------------(2nd Ord. all-I)
Q:prop                            COV-IND
----------------------------------------------------------------(2nd Ord. all-E)
       all X:nat. (all Y:nat. (Y<X -> Q(Y)) -> Q(X))
       -> all Z:nat. Q(Z)
```

The complete proof of all X:nat. (all Y:nat. (Y<X -> P(Y)) -> P(X)) -> all Z:nat. P(Z) is shown in Appendix 1. This can be seen as a proof schema; if the free variable, P, is instantiated to some individual proposition, a proof in first order logic can be obtained.

### 3.2 Proof Compilation Algorithm for 2nd Ord. all-I/E Rules

The proposition variables and the second order all-I/E rules are used to handle user-defined rules of inference such as the course of value induction schema explained in 3.1.

The proof compilation algorithm for second order all-I/E rules is based on the idea of realizability interpretation of second order intuitionistic logic.

```
          [P:prop]
            F(P)
Ext(------------------(2nd Ord. all-I)) == LAMBDA RV(P). Ext(P/RV(P)|-F(P))
     all P:prop. F(P)
```

RV(P) is a variable as long as P is a variable. If P is instantiated to some particular proposition, RV(P) returns a value of Rv(P). The intentional meaning of LAMBDA is similar to ordinal lambda notation. LAMBDA is only used to distinguish the above case. Ext(P/RV(P) |- F(P)) means that if the realizer of P is needed in the procedure of proof compilation of F(P), meta-variable RV(P) should be used instead of the realizer code.

```
    Q:prop        all P:prop. F(P)
Ext(------------------------------(2nd Ord. all-E))
                F(Q)

                  == perform beta-reduction on

                     Ext(all P:prop. F(P))(Rv(Q))
```

Ext(all P:prop. F(P)) must be of the form LAMBDA RV(P).Ext(P/RV(P)|-F(P)). After one beta-reduction of LAMBDA-expression, the above code is Ext(F(Q)). This corresponds to the following normalization of second order logic:

```
                     [P:prop]
                      PI(P)
                       F(P)
                 ----------------(2nd Ord. all-I)
Q:prop.          all P:prop. F(P)
---------------------------------(2nd Ord. all-E)
            F(Q)
```

By applying the normalization of proofs in second order logic [Prawitz 65], the following proof can be obtained:

```
        [Q]
        PI(Q)
        F(Q)
```

## 3.3 Proof Compilation of Course of Value Schema

The following code is generated from the proof of COV-IND given in 3.1 by the proof compilation algorithm given in 2.3 and 3.2.

```
LAMBDA RV(P(X)).
        lambda RV(P(X)).lambda [Z]. RV(P(X))(Z)(A0(Z))        [COV-CODE]
```

where

```
A0 == mu RV(P(Y)).
      lambda [X].
          if X = 0 then lambda [Y]. any[1(Rv(P(Y)))]
          else lambda [Y].
                  if left=AA(X)(Y) then RV(P(Y))(pred(X))(Y)
                  else RV(P(X))(X)(RV(P(Y))(pred(X)))

AA ==  mu [Z]. lambda [X].
            if X=0 then
              lambda [Y]. if Y=0 then right else any[1]
            else
              lambda [Y].
                 if left = CODE1(Y) then left
                 else if left = Z(pred(X))(pred(Y)) then left
                      else right

CODE1 == lambda [P]. if P=0 then left else right
```

The code 'left = AA(X)(Y)' in A0 is the conditional equation which is logically equivalent to Y<X.

The meta-variable, RV(P(X)), denotes the realizing variables of all X:nat. (all Y:nat. (Y<X -> P(Y)) -> P(X)). The proof of this part must be given in course of value induction proofs, and the proof contains the computational meaning of how to construct the justification of P(X) by using the justifications of P(Y) (for all Y s.t. Y < X).

RV(P(Y)) denotes the realizing variables of all Y:nat. (Y<X -> P(Y)).

## 4. Simple Example: GCD Program

The GCD program is taken as a simple example which uses COV-IND.

### 4.1 GCD Proof

The specification of GCD program is defined as follows:

$$\text{all N:nat. all M:nat. exist D:nat. } (D|N \wedge D|M)$$

where for P:nat and Q:nat, $P|Q$ == exist R:nat. Q=R*P.

The specification and proof that the constructed natural number is actually a maximal one which satisfies the specification is omitted here for simplicity, but the natural number which satisfies this condition is constructed in the proof given below. The proof is called proof of GCD program or GCD proof in the following description.

The course of value proof of this specification is shown in Appendix 2.

## 4.2 Proof Compilation of GCD Proof

(1) The proof trees in Appendix 1 and 2 give COV-TREE for the gcd program.

The following code is obtained by proof compilation.

```
f0(g)

where

f0 == COV_CODE(Rv(Q))

g == lambda [N]. lambda [Z0, Z1, Z2]. if left = CODE1(N) then CODE2 else CODE3

        CODE2 ==  lambda [M]. M, (lambda [Q].0)(M), (lambda [R].1)(M)
        CODE3{N} ==  lambda [M]. (Z0(M mod N)(N),
                                  Z1(M mod N)(N),
                                  Z1(M mod N)(N)
                                  + (Z1(M mod N)(N))*(M-(M mod N)/N))
```

Here, Q == ABS [X]. all M:nat. exist D:nat. D|X /\ D|M.

Let Rv(Q(X)) == (W0, W1, W2), and Rv(Q(Y)) = (ZZ0,ZZ1,ZZ2).

Consequently, by beta-reduction of LAMBDA-expression, f0 can be reduced

to the following form:

```
f == lambda [W0, W1, W2]. lambda [Z]. (W0, W1, W2)(Z)(A1(Z))
        where
        A1 == mu [ZZ0,ZZ1, ZZ2].
             lambda [X].
                if X = 0 then lambda [Y]. any[3]
                else lambda [Y].
                        if left = AA(X)(Y) then (ZZ0, ZZ1, ZZ2)(pred(X))(Y)
                        else (W0, W1, W2)(X)((ZZ0, ZZ1, ZZ2)(pred(X)))
```

The obtained program is gcd0 = f(g).

# 5. Execution of the Extracted Codes

## 5.1 Tiny Quty Interpreter

Tiny Quty is a subset of Quty [Sato 87]. Tiny Quty is non-typed sequential functional language to describe executable codes extracted from constructive proofs. The syntax of Tiny Quty is given in Section 2 as the definition of term expressions. For simplicity, the language presented here has no syntax for list structure. The chief difference from ordinal functional languages is that it allows sequences of variables as parameters of the fixed point operator, mu, i.e., multi-valued function can be written. Another difference is that a function can be applied not only to first order objects such as atoms and integer but also to functions.

The interpreter of Tiny Quty is basically the call-by-value evaluator of lambda-expressions. However, the following features should be noted.

(1) f == mu [Z0,..,Zn]. Exp(Z0,..,Zn) is regarded as a sequence of single valued functions f0,..,fn, and fi (0=<i=<n) is defined as proj(i)(f) The following reduction can be performed on f:

```
mu [Z0,..,Zn]. Exp(Z0,..,Zn) ---> Exp(f0,..,fn)
```

(2) lambda [X0,..,Xn].Exp is regarded as another description of lambda [X0]. .. , lambda [Xn]. Exp.

(3) (lambda [X0, ...,Xn]. Exp1(X0,..,Xn))(Exp2) can be reduced to Exp1(A0, .., An) where (A0,..,An) = split(Exp2). The definition of the function, split, is as follows:

```
1) split( lambda [X]. Exp) ) = (lambda [X].A0, .. , lambda [X].An)
        where (A0, .. ,An) = split(Exp)
2) split(if A then B else C)
        = (if A then B0 else C0, ... , if A then Bn else Cn)
                if (B0,..,Bn) = split(B) and (C0,..,Cn) = split(C)
          (if A then B else C0, ... , if A then B else Cn)
                if split(B) = B
                   and (C0, .. ,Cn) = split(C)
          (if A then B0 else C, ... , if A then Bn else C)
                if (B0,..,Bn) = split(B),
                   and split(C) = C
3) split(A(B)) = (A0, .., An)(B)
        where (A0, .. , An) = split(A)
4) split(mu [Z0,..,Zn]. Exp(Z0,..,Zn)) = (f0, .. , fn)
        where f = mu [Z0, .., Zn]. Exp(Z0,..,Zn),
              and fi = proj(i)(f) (0=<i=<n)
5) otherwise split(Exp) = Exp, i.e., Exp cannot be split.
```

(4) No reduction is performed on any[N] term. any[N](Term) is reduced to Term.

## 5.2 Evaluation of the GCD Code

The code extracted in Section 4 is a program that takes two natural numbers as inputs and returns the triplet of three natural numbers. The first element of the pair is the gcd of the inputs, and the other two elements are verification information that can be seen as the decoded proof to show that the first element of the pair is actually the gcd. If one is interested in only the value of gcd, the extracted code should be properly transformed into a single valued function.

# 6. Optimization Technique

In the PX system [Hayashi 87], the optimization of extracted codes proceeds as follows: if the code of the form (lambda [X]. A)(B) is extracted in the process of proof compilation, then perform beta-reduction of the code immediately. The optimization technique of proof compilation can be presented more systematically.

## 6.1 Proof Normalization and Partial Evaluation of Programs

Normalization of proofs corresponds to partial evaluation of extracted codes from the proofs through realizability interpretation. The following are the normalization rules given in [Prawitz 65].

(1) all-normalization

```
        PI(a)
         P(a)
    --------------(all-I)
     all X. P(X)              ==>      PI(t)
     ------------(all-E)                P(t)
        P(t)
```

(2) -> normalization (cut elimination)

```
               [A]
               PI'                PI
                B                 [A]
    PI     -------(->I)           PI'
     A       A -> B       ==>    ------
    ---------------(->E)           B
             B
```

There are other rules of normalization such as exist, $\wedge$, $\vee$-normalization rules, but they are not effective for the optimization in proof compilation as can be seen by the definition of the Ext procedure. Rules (1) and (2) correspond to beta-reduction of lambda expressions.

Note that in terms of proof compilation, the following diagram commutes:

```
        Proof 1    -----(normalization)--------> Proof 2
           |                                        |
           |                                        |
    (proof compilation)                     (proof compilation)
           |                                        |
           |                                        |
           V                                        V
        Realizer Code 1  -----(partial eval.)--> Realizer Code 2
                               (reduction)
```

Following this diagram, optimization facilities can be realized in either of two ways:

1) By implementing a proof normalizer

Proofs are normalized first, and then compiled.

2) By implementing a partial evaluator built in the proof compiler

Proofs are compiled first, then the partial evaluation method of the functional programming language is applied to the extracted codes.

From the aesthetic point of view, both proofs and codes should be transformed simultaneously to maintain a clear correspondence between proofs and programs in terms of realizability.

The order of applying the normalization rules can be arbitrary. If the normalization rules are applied from the leaves of proof trees, this corresponds to call-by-value evaluation of the programs extracted from the proof trees. If they are applied from the bottom of proof trees, it means call-by-name evaluation. The operational semantics given in 5. 1 defines call-by-value evaluation. However, this is just for efficiency of runtime evaluation.

6.2 Example of Proof Normalization

For the GCD proof given in 4.1, first, the all-normalization rule can be applied to the proof of M|0 and M|M in TREE_1, as follows.

```
                   ------(*)
                   0:nat  [P:nat]
       ----(*)     --------------(*)
       0:nat        0=0*P                                  ----(*)
       --------------------(exist-I)                       0:nat  [M:nat]
       exist D':nat. 0=D'*P                       -----(*) ------------(*)
       --------------------(all-I)                0:nat        0=0*M
[M:nat]  all P:nat. P|0                 ===>      -------------------(exist-I)
------------------------(all-E)                       M|0
            M|0
                                                 Note: M|0 == exist D:nat.
                                                                0 = D*M

                   -----(*)
                   1:nat  [Q:nat]
       ----(*)     --------------(*)
       1:nat      Q=1*Q                                      -----(*)
       --------------------(exist-I)                          1:nat  [Q:nat]
       exist D":nat.Q=D"*Q                        -----(*)   -------------(*)
       --------------------(all-I)                1:nat       M=1*M
 [M:nat]  all Q:nat.Q|Q                 ===>      -------------------(exist-I)
 ----------------------(all-E)                        M|M
          M|M
```

By this normalization, CODE2 is translated to the CODE22:

```
        CODE22 == lambda [M]. M,0,1
```

As the (->E) rule is used in the course of value induction schema given at the end of Section 3, the -> normalization rule can be applied:

```
                                            Proof of course of value schema
                                            by mathematical induction:

Course of value proof:                      [all X. (all Y.(Y<X)->P(Y)) -> P(X)]
                                                        PI'
        PI                                          all Z. P(Z)
--------------------------------(all-I)    -------------------------------(->I)
all X. (all Y.(Y<X)->P(Y)) -> P(X)         all X. (all Y.(Y<X)->P(Y)) -> P(X)
--> all Z. P(X)                            --> all Z. P(Z)
----------------------------------------------------------------------(->I)
                all Z. P(Z)

==>
                 PI
          ----------------------------------(all-I)
          all X. (all Y. (Y<X->P(Y)) -> P(X)
          -> all Z. P(X)
                 PI'
                all Z. P(Z)
```

By this transformation, beta reduction of f(g) is performed, and the all-normalization rule can also be applied to Gamma1 and Gamma2 in Section 3 combined with the course of value proof given in 4.1, then the code is as follows:

```
gcd1 ==  lambda [Z].
        (lambda [ZZ0, ZZ1, ZZ2].
             if left = CODE1(Z) then CODE22 else CODE3[Z]
        )(A2(Z))

        where
        A2 == A1[
                 (W0,W1,W2)
                 <-
                 lambda [ZZ0, ZZ1, ZZ2].
                        if left = CODE1(X) then CODE22 else CODE3[X]
                ]
```

6.3 <u>Modified \/ Code</u>

For left = (lambda [P]. if P=0 then left else right)(N) in code AA, it corresponds to the proof of N=0\/N>0. However, none of the normalization rules in 6.1 can be applied, although this code can be partially evaluated to left = (if N=0 then left else right). As is known from the example given in 5.2, most of the execution of the gcd program is that of AA, the code extracted from the proof of Y<X+1 |- Y<X \/ Y=X, and is logically equivalent to Y<X, as explained in 4.2.

On the other hand, as given in 4.1, N=0\/N>0 (N:nat) and Y<X+1 |- Y<X\/Y=X are proved by mathematical induction. However, in practical situations, it is not efficient if we must always prove well known properties of natural number of this kind strictly by using induction. For this reason, the following modification is introduced in proof compilation:

```
                 [A]      [B]
      A\/B        C        C
Ext(---------------------------(\/E)) == if A then Ext(A|-C) else Ext(B|-C)
                  C
                                          when A and B are equations or
                                          inequations of natural numbers
```

In this case, the proof of A\/B can be omitted by declaring this formula as an axiom.

By this optimization, the gcd code obtained in 6.2 is changed as follows:

```
gcd2 ==  lambda [Z].
        (lambda [ZZ0, ZZ1, ZZ2].
             if Z=0 then CODE22 else CODE3{Z}
        )(A3(Z))

        where
        A3 = mu [ZZ0,ZZ1, ZZ2].lambda [X].
               if X = 0 then lambda [Y]. any[3]
               else lambda [Y].
                       if Y < X then (ZZ0, ZZ1, ZZ2)(pred(X))(Y)
                       else (lambda [ZZ0, ZZ1, ZZ2].
                                if X=0
                                then CODE22
                                else CODE3{X}
                            )((ZZ0, ZZ1, ZZ2)(pred(X)))
```

## 7. Incremental Compilation of Proofs

### 7.1 Referring Theorems Already Proven

Theorems already proven should be stored in a library accompanied by the code extracted from the proof. They can be referred to within proofs of theorems. The proof compiler system also refers to theorems in the library when it must extract the codes corresponding the the theorems. In this case, the compiler uses the code stored in the library. For theorems of the form all X. A(X) and A -> B, the code must be of the forms lambda [X]. T(X) and lambda [Rv(A)].T(Rv(A)). These theorems are typically used in the following situations, and correspond to the subroutine call in ordinal programming.

```
          Theorem:                               Theorem:
t         all X. A(X)                   A        A -> B
-------------------(all-E)              -----------------(->E)
      A(t)                                      B
```

Then, the extracted codes, (lambda [X].T(X))(t) and lambda [Rv(A)].T(Rv(A))(Ext(A)), can be partially evaluated to T(t) and T(Ext(A)). This procedure has the same effect on the extracted codes as that when complete proofs (proofs that do not refer to any theorems already proven) are compiled and optimized with proof normalization.

### 7.2 Example of Incremental Compilation

The proofs given in 3.1 and 4.1 use three simple theorems on arithmetic, theorems 1, 2 and 3. This kind of theorem is used frequently in the programming of proofs as programs, so it should be stored in the library system in the following forms:

```
Theorem 1: Statement => all K:nat. K=0\/K>0,
           Extracted Code => lambda [K]. if K=0 then left else right

Theorem 2: Statement => all P:nat. P|0
           Extracted Code => lambda [P]. 0

Theorem 3: Statement => all Q:nat. Q|Q
           Extracted Code => lambda [Q]. 1
```

Then, the programmer simply declares the names of the theorems in the proof of the gcd program. The code, gcd3, is extracted through the proof compilation and optimization by proof normalization:

gcd3 is obtained from gcd1 by replacing CODE1 in the identifier <Code of theorem 1>, CODE22 into

(lambda [M]. M, <Code of theorem 2>(M), <Code of theorem 3>(M)).

Then, attach the 'Extracted Code' in the library to identifiers <Code of theorem i> (i = 1,2,3) part, and perform the following partial evaluation:

```
(lambda [P].0)(M) -> 0

(lambda [Q].1)(M) -> 1
```

Note that this partial evaluation corresponds to all-normalization illustrated at the beginning of 6.2.

Then, the obtained code becomes the same as gcd1.

## 8. Conclusion

This paper presented a proof compilation technique based on the notion of realizability and proof normalization. A higher order feature was introduced to handle the description of user-defined rules of inference. Optimization and incremental compilation can be handled quite naturally with the notion of proof normalization. Modified $\vee$-code was also introduced as a powerful technique of optimization. The extracted codes can be executed as functional style programs. The syntax and the interpreter system of the language were also presented.

Acknowledgment

Thanks must go to Dr. Aiba, Dr. Murakami, and Mr. Sakai of ICOT, and to Mr. Kameyama and Professor Sato at Tohoku University, who gave me many useful suggestions.

References

[Constable 86] Constable, R. L, et al., "Implementing Mathematics with the Nuprl Proof Development System", Prentice-Hall, 1986

[Hayashi 87] Hayashi, S. and Nakano, H., "PX: A Computational Logic", RIMS-573, Research Institute for Mathematical Sciences, Kyoto University, 1987

[Prawitz 65] Prawitz, D., "Natural Deduction", Almqvist & Wiksell, 1965

[Sato 85] Sato, M., "Typed Logical Calculus", TR-85-13, Department of Computer Science, University of Tokyo, 1985

[Sato 87] Sato, M., "Quty: A Concurrent Language Based on Logic and Function", Proceedings of the Fourth International Conference on Logic Programming, Melbourne, 1987

Appendix 1: Proof Tree of COV-IND:

```
          TREE-1                          TREE-2
          ---------------------------------------(nat-ind)
[Z:nat]   all X:nat.(all Y:nat.(Y<X -> P(Y)))
-----------------------------------------------(all-E)
  all Y:nat. (Y<Z -> P(Y))                            Gamma1
-----------------------------------------------------------------(->E)
                        P(Z)
                  ----------------(all-I)
                  all Z:nat. P(Z)
         ----------------------------------------------(->I)
         all X:nat. (all Y:nat. (Y<X -> P(Y)) -> P(X))
         -> all Z:nat. P(Z)
```

Gamma1:

```
     [Z:nat] [all X:nat.(all Y:nat. (Y<X -> P(Y)) -> P(X))]
     ---------------------------------------------------------(all-E)
              all Y:nat. (Y<Z -> P(Y)) -> P(Z)
```

TREE-1:

```
                  [Y:nat]
                  ------(*)
      [y<0]       ~(Y<0)
      ----------------(->E)
             void
           ------(void-E)
            P(Y)
        ------------(->I)
        Y<0 -> P(Y)
     ------------------------(all-I)
     all Y:nat.(Y<0 -> P(Y))
```

TREE_2:

```
                           [Y:nat]  [HYP]               [HYP]    Gamma2
                           ---------------(all-E)       ----------------(->E)
                      [Y<X]  Y<X -> P(Y)            [Y=X]      P(X)
                      ------------------(->E)       ---------------(=-E)
     TREE-2-1                P(Y)                           P(Y)
     ------------------------------------------------------------(\/-E)
                              P(Y)
                      --------------------(->I)
                       Y<succ(X) -> P(Y)
                ------------------------------(all-I)
                all Y:nat.(Y<succ(X) -> P(Y))
```

```
     HYP == all Y:nat.(Y<X -> P(Y))
```

Gamma2:

```
      [X:nat] [all X:nat.(all Y:nat. (Y<X -> P(Y)) -> P(X))]
      ---------------------------------------------------------(all-E)
               all Y:nat. (Y<X -> P(Y)) -> P(X)
```

```
TREE-2-1:    X:nat, Y:nat, Y<succ(X) |- Y<X \/ Y=X

---(*)
 0
---(=)
0=0   [0<1]
------(/\I) [Y+1<1]
0=0         -----(*)
/\0<1       Y<0     [Y:nat]
------(/\E) -----------(*)
0=0          void
---(\/I)    --------(void-E)
0<0         Y+1<0
\/0=0       \/ Y+1=0
------(->I) ---------(->I)      TREE-2-2   TREE-2-3   TREE-2-4
0<1 ->      Y+1<1 ->            --------------------------------(\/E)
0<0         Y+1<0                           Y<X+1 \/ Y=X+1
\/0=0       \/ Y+1=0               -------------------------(->I)
-----------------(nat-ind)          Y<X+2 -> Y<X+1 \/ Y=X+1
         all Y:nat.               -----------------------------(all-I)
           (Y<1 ->                all Y:nat.
             Y<0 \/ Y=0)             (Y<X+2 -> Y<X+1 \/ Y=X+1)
         ------------------------------------------------------(nat-ind)
[X:nat]  all X:nat. all Y:nat. (Y < X+1 -> Y<X \/ Y=X)
-----------------------------------------------------------(all-E)
[Y:nat]                all Y:nat. (Y < X+1 -> Y<X \/ Y=X)
-----------------------------------------------------------(all-E)
[Y<X+1]                             Y < X+1 -> Y<X \/ Y=X
-----------------------------------------------------------(->E)
                     Y<X \/ Y=X


TREE-2-2:

[Y:nat]       Theorem-1
------------------------(all-E)
    Y=0 \/ Y>0


TREE-2-3:
                    [X:nat]
                    -------(*)
         [Y=0]      0 < X+1
         -------------------(=-E)
            Y < X+1
         -------------------(\/I)
         Y < X+1 \/ Y = X+1


TREE-2-4:

         [Y>0]
         ------(*)
         Y-1:nat    [HYP"]
         -------------(all-E) [Y-1<X]         [Y-1=X]
[Y<X+2]     Y-1<X+1 ->        -------(*)      -------(*)
-------(*)  Y-1<X             Y<X+1           Y=X+1
Y-1<X+1     \/ Y-1=X          --------(\/I)   -------(\/I)
-----------------(->E)        Y<X+1           Y<X+1
Y-1<X \/ Y-1 =X               \/ Y=X+1        \/ Y=X+1
-----------------------------------------------------(\/E)
Y < X+1 \/ Y = X+1

where  HYP" = all Y:nat. (Y<X+1 -> Y<X \/ Y=X)
```

Theorem-1 is a simple theorem of number theory that states 'any natural number is equal to 0 or larger than 0'. This can be proved by mathematical induction as follows:

```
                        [K=0]                [K>0]
                        ------(*) ---(*)     -----(*) ---(*)
                         K+1=1     1>0       K+1>1     1>0
  ---(*)                -------------(=)     -------------(>)
   0         [K=0\/K>0]  K+1>0                K+1>0
 ----(=)     ---------------------------------------(\/E)
  0=0               K+1>0
---------(\/I) -------------(\/I)
0=0\/0>0        K+1=0\/K+1>0
---------------------------(nat-ind)
    all K:nat. K=0\/K>0
```

Appendix 2: The course of value proof of the GCD Proof

Let Q(X) == all M:nat. exist D:nat. D|X∧D|M.

```
[N:nat]  Theorem 1
------------------(all-E)
    N=0∨N>0                     TREE_1            TREE_2
-------------------------------------------------------------(∨-E)
                                                   Q(N)
                    -----------------------------------------(->I)
                    all L:nat. (L < N -> Q(L)) -> Q(N)
        -----------------------------------------------------(all-I)
        all N:nat. (all L:nat. (L < N -> Q(L)) -> Q(N))
```

TREE_1: Proof of N=0 |- Q(N)

```
      [M:nat]          Theorem 2          [M:nat]       Theorem 3
      ------------------------(all-E)    ----------------------(all-E)
                        M|0                     M|M
                        -------------------------(∧I)
         [M:nat]                 M|0∧M|M
         -----------------------------------(exist-I)
                   exist D:nat. D|0∧D|M
                  ----------------------(all-I)
[N=0]                    Q(0)
--------------------------------------(=E)
                         Q(N)
```

Theorem 2 and 3 is simple theorems of number theory:

Theorem 2:

```
              ------(*)
               0:nat   [P:nat]
----(*)       ---------------(*)
 0:nat          0=0*P
--------------------(exist-I)
exist D':nat. 0=D'*P
--------------------(all-I)
   all P:nat. P|0
```

Theorem 3:

```
              -----(*)
               1:nat   [Q:nat]
----(*)       ---------------(*)
 1:nat          Q=1*Q
--------------------(exist-I)
exist D":nat.Q=D"*Q
--------------------(all-I)
   all Q:nat.Q|Q
```

TREE_2: Proof of N>0 |- Q(N)

```
                   [N>0]  [M:nat]
                   ------------(*)
                     (M mod N):nat    [HYP]
[N>0]  [M:nat]      ------------------(all-E)  [d|(M mod N)]
--------------(*)    (M mod N) < N              [        ∧d|N]
   (M mod N)<N       -> Q(M mod N)              -----------(∧E)
   --------------------------(->E)                d|N          TREE_2_sub
[N:nat]     Q(M mod N)                             ------------------------(∧-I)
---------------------(all-E)          [d:nat]          d|N∧d|M
   exist D:nat                        -------------------------(exist-I)
     D|(M mod N)∧D|M                   exist D:nat. D|N∧D|M
-----------------------------------------------------------------(exist-E)
           exist D:nat. D|N∧D|M
           ----------------------(all-I)
                 Q(N)
```

```
where  HYP == all L:nat. (L<N -> Q(L))


TREE_2_sub: Proof of d:nat, d|(M mod N)/\d|N, M:nat,N:nat |- d|M

[M:nat] [N:nat]            [M:nat] [N:nat]
----------------(*)   ------------------------------------(*)
(M-(M mod N))/N:nat      M=N*((M-(M mod N))/N) + (M mod N)
---------------------------------------------------------(exist-I)
exist d0:nat.
  M=N*d0 + (M mod N)                                    TREE_2_subsub
-----------------------------------------------------------------------(exist-E)
                              d|M

TREE_2_subsub: Proof of
          d|N/\d|(M mod N), M:nat, N:nat, d0:nat, M=N*d0 + (M mod N) |- d|M

                                  [N=d3*d]
                                  [d0:nat]
                    [d0:nat]      [d1:nat]
                    [d3:nat]    -----------(*)
[d|(M mod N)]     ---------(*) N*d0          [d1:nat]    [N*d0=d2*d]
[/\ d|N      ]    d3*d0:nat    =d3*d0*d      [d2:nat]    [(M mod N)=d1*d]
----------(/\E) -----------------(exist-I) ----(*)  ------------------------(*)
      d|N            d|N*d0                 d1+d2      N*d0+(M mod N)=(d1+d2)*d
      -------------------------(exist-E)    ----------------------------(exist-I)
[d|N/\d|(M mod N)]    d|N*d0                     d|N*d0+(M mod N)
---------------(/\E)  -------------------------------------(exist-E)
      d|(M mod N)              d|N*d0+(M mod N)
      ----------------------------------------(exist-E)
[M=N*d0+(M mod N)]      d|N*d0+(M mod N)
-----------------------------------------(=-Elim)
                    d|M

where
          d|(M mod N) == exist(d1:nat. (M mod N) = d1*d)
          d|N*d0 == exist(d2:nat. N*d0 = d2*d)
          d|N == exist(d3:nat. N=d3*d)
```